PARALELIZACIÓN DEL PROBLEMA DE SATISFACCIÓN DE RESTRICCIONES UTILIZANDO ARCO CONSISTENCIA

Jordi Alcaraz Rodriguez

Resum– El problema de satisfacción de restricciones se utiliza en varios campos de la informática: Inteligencia Artificial, planificación de recursos, etc., y se considera NP-completo. A causa de la complejidad computacional de este problema es necesario utilizar buenas heurísticas y paralelismo para reducir el tiempo de cómputo. En este proyecto se realizarán varias implementaciones paralelas para maximizar el uso del hardware disponible y reducir el tiempo de resolución del problema.

Paraules clau– Problema de satisfacción de restricciones, computación paralela, Xeon Phi, árbol de búsqueda en profundidad

Abstract– In some fields related to Computer Science the Constraint Satisfaction problem is used, in exemple: Artificial Intelligence, resource planing, etcetera, and this problem is NP-complete. Because of it's computational complexity there is a need to use good heuristics and paralelism to reduce the time to solve it. In this project some parallel implementations will be performed to maximize the use of the available hardware and minimize the time needed to solve this problem.

Keywords– Constraint Satisfaction Problem, CSP, parallel computing, Xeon Phi, Depth-first search

1 INTRODUCCIÓN

En los últimos años los fabricantes de procesadores decidieron dejar de incrementar la velocidad de reloj para aumentar el rendimiento. Este incremento ocasiona varios problemas, el principal es el aumento del consumo energético, el cual eleva la temperatura[1][2]. La solución adoptada fue cambiar la arquitectura de los procesadores, incrementando el número de núcleos de ejecución. Este cambio afecta a los modelos de programación existentes y surge la necesidad de programación orientada a sistemas paralelos.

Al aparecer un nuevo mercado orientado al paralelismo, algunas empresas, como NVIDIA e Intel, desarrollan nuevos coprocesadores. Estos sistemas se basan en la utilización de una mayor cantidad de núcleos de cómputo, y son aptos para aplicaciones con un gran nivel de paralelismo.

Con el fin de analizar las mejoras e inconvenientes de estos nuevos sistemas, se utilizará un problema denominado Problema de Satisfacción de Restricciones, conocido con el nombre de *Constraint Satisfaction Problema* (CSP), el cual puede beneficiarse de este nuevo modelo de programación al necesitar gran capacidad de cómputo para su resolución.

1.1. Organización del documento

El resto del documento sigue la siguiente organización: En la sección 2, "Estado del Arte", se explicará el problema CSP y, también, de sistemas y modelos paralelos. En la sección 3, "Objetivos", se detallarán los objetivos de este proyecto. Y en la sección 4, "Metodología", como se llegan a cumplir los objetivos. En la sección 5, "Desarrollo", se describe el algoritmo secuencial y los modelos utilizados para paralelizarlo. En la sección 6, "Experimentación", se explicarán los experimentos realizados y el hardware utilizado. En la sección 7, "Resultados", se expondrán los resultados obtenidos y se razonarán. Finalmente, en la sección 8, "Conclusiones", se resumirán los resultados del proyecto.

2 ESTADO DEL ARTE

Este apartado se divide en dos partes. Primero se expone el estado del arte del problema CSP y, para finalizar, el estado del arte de los sistemas paralelos.

2.1. Constraint Satisfaction Problem

En inteligencia artificial, y en otras áreas de la computación, muchos problemas pueden ser vistos como proble-

E-mail de contacte: jordi.alcaraz@e-campus.uab.cat
Menció realitzada: Enginyeria de Computadors
Treball tutoritzat per: Juan Carlos Moure (CAOS)

mas de satisfacción de restricciones[3][4]. Los problemas de satisfacción de restricciones se componen de:

- Conjunto finito de variables, $X = \{x_1, x_2, ..., x_n\}$.
- Conjunto de Dominios de cada variable, D_1, D_2, ..., D_n, donde cada dominio, a su vez, es el conjunto de valores posibles de una variable.
- Conjunto de restricciones (*Constraints*) entre variables, C_1, C_2, ..., C_k.

Las restricciones son relaciones entre variables que limitan los valores válidos. Un ejemplo de restricción es Alldifferent(X_1, X_2, ..., X_n), indicando que cada valor diferente únicamente puede asociarse a una variable.

Una de las técnicas más utilizadas para la resolución de CSPs es la exploración en profundidad de un árbol de soluciones y volver a los nodos anteriores con *backtracking*. Además se utilizan técnicas de poda (*pruning*) para limitar la cantidad total de asignaciones de valores a variables a explorar [5].

Este problema se considera NP-completo debido al crecimiento exponencial de cálculo necesario para resolver el problema al aumentar el número de variables.

Un ejemplo sencillo de CSP es el problema de las N-Reinas. Este problema consiste en colocar N reinas en un tablero de tamaño NxN de manera que ninguna pueda atacar a otra. La codificación utilizada en nuestra implementación es la siguiente:

- Variables, $X = \{x_1, x_2, ..., x_n\}$. Una variable por cada reina y asignada a una fila. El valor de la variable x_i indica la columna de la reina de la fila i-ésima.
- Dominio común para todas las variable, D_1=(1, 2, ..., N). D_1=D_2=...= D_n.
- Las restricciones se pueden ver en la figura 1 y son las siguientes:
  - Diferente-Columna(X_1, X_2, ..., X_N). Todas las variables se asignan a columnas diferentes.
  - Diferente-Diagonal(X_1, X_2, ..., X_N). Todas las variables se asignan a diagonales diferentes.

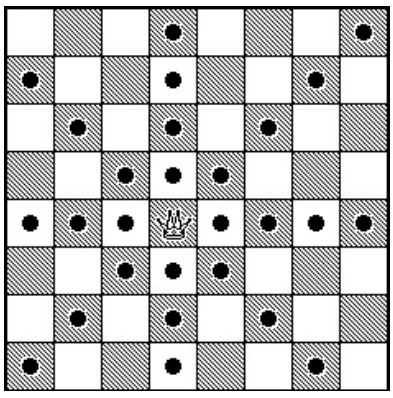

Figura 1: Reina y sus restricciones [6].

2.2. Sistemas paralelos

Actualmente la mayoría de ordenadores tienen procesadores con multiples núcleos de procesamiento. Con el fin de aprovechar de manera eficiente los procesadores, los algoritmos secuenciales deben dividir el trabajo a realizar de manera eficiente entre los diferentes núcleos del sistema, surgiendo la necesidad de nuevos modelos de programación.

2.2.1. Procesadores multicore de propósito general

La arquitectura utilizada para paralelismo entre núcleos de cómputo se denomina MIMD (*Multiple Instruction Multiple Data*). MIMD permite a cada núcleo ejecutar diferentes instrucciones a diferentes datos de forma simultánea.

Para incrementar el paralelismo dentro de cada núcleo de cómputo se creó el procesamiento SIMD (*Single Instruction Multiple Data*). El procesamiento SIMD, también conocido como vectorización, permite ejecutar la misma instrucción sobre diferentes datos de forma simultánea en un único núcleo. La cantidad de datos que pueden ser procesados de forma paralela, y que componen el vector SIMD, depende del tamaño de los elementos del vector y del tamaño de los registros SIMD del procesador [7].

De la misma manera que en un procesador se pueden tener múltiples núcleos de cómputo un ordenador puede tener más de un procesador. Los sistemas con varios procesadores se denominan sistemas *multisocket*. Debido a esto se pueden dividir los sistemas MIMD en dos grupos según su organización de memoria [8]:

- *Uniform Memory Acces* (UMA). Todas las unidades de cómputo del sistema comparten la memoria principal y tienen la misma latencia mínima (sin colisiones) para acceder a memoria.
- *Non-Uniform Memory Acces* (NUMA). En este caso la memoria se encuentra distribuida en bloques y cada bloque asociado a uno de los diferentes procesadores del sistema. La latencia mínima (sin colisiones) para acceder a memoria varía según el procesador que haga la petición y el bloque que deba responder a la petición. Es la arquitectura utilizada con más frecuencia por los sistemas multisocket.

En algunos procesadores multicore, Intel ha introducido la tecnología *Hyper-Threading*. Esta tecnología permite a un núcleo de cómputo ejecutar dos threads de manera simultánea. Gracias a tener más de un thread ejecutándose en el mismo núcleo se aumenta el uso de los recursos al permitir ejecutar instrucciones de threads diferentes que esconden los tiempos de espera producidos por fallos de caché o de predicción de saltos, entre otros (Figura 2) [9].

2.2.2. Coprocesador Intel Xeon Phi

Además de los procesadores se pueden utilizar coprocesadores para recudir el tiempo de ejecución de los programas paralelos. El coprocesador creado por Intel recibe el nombre de Intel Xeon Phi y se diseñó para aumentar el rendimiento en aplicaciones que obtengan una buena escalabilidad en procesarores Intel Xeon [10].

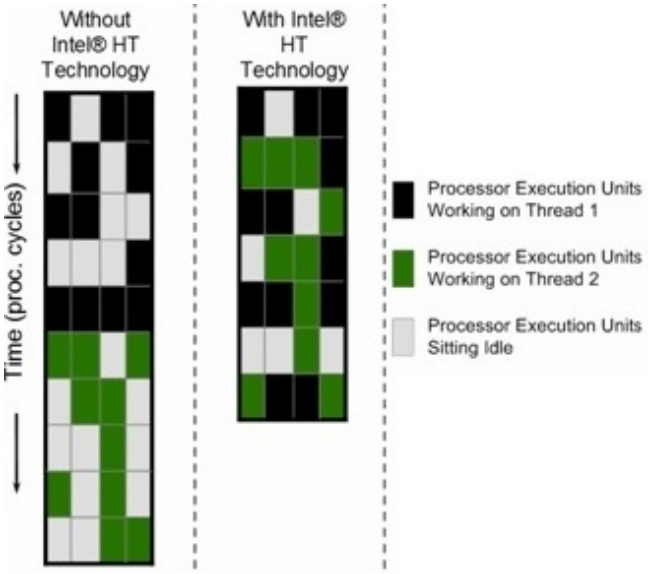

Figura 2: Sistema sin Hyper-Threading y sistema con Hyper-Threading. Se puede ver el efecto en el tiempo de ejecución de la posibilidad de esconder los tiempos de espera [9].

EL coprocesador Intel Xeon Phi dispone de una capacidad de cómputo siete veces superior a los procesadores Xeon Phi [11]. Además, esta mejora se obtiene con un mejor rendimiento energético. La capacidad de cómputo del Intel Xeon Phi 7120P es de 1208 GFLOPS en doble precisión, con una eficiencia de 3.66 GFLOPS/W. En cambio, el procesador Intel Xeon 2670 dispone de una capacidad de cómputo de 166.4 GFLOPS, con una eficiencia de 1.45 GFLOPS/W.

El coprocesador Xeon Phi cuenta con una arquitectura MIMD y utiliza los mismos modelos de programación que la CPU, facilitando así la programación y adaptación del código.

A diferencia de los procesadores Xeon, de hasta 18 núcleos de computo y 36 hilos de ejecución, los coprocesadores Phi contienen hasta 61 cores y 244 threads. Esto hace que sea muy importante la escalabilidad de las aplicaciones a ejecutar.

Otra diferencia es el tamaño de los vectores SIMD. Mientras que el Xeon Phi permite instrucciones SIMD de 512 bits, los procesadores Xeon disponen de instrucciones SIMD de 256 bits [12], siendo importante utilizar de manera eficiente las unidades SIMD del Phi para obtener un buen rendimiento.

Se dispone de dos modos diferentes de ejecución: ejecución nativa (el comando *ssh* permite conectarse al coprocesador y ver al Phi como un sistema independiente) y ejecución *offload* (modelo híbrido que permite especificar qué parte del código se ejecuta en el *host* y qué parte se ejecuta en el Phi).

El coprocesador dispone de un sistema operativo basado en Unix para poder gestionar la ejecución nativa y el sistema operativo necesita utilizar un núcleo. En ejecuciones offload no se permite utilizar ese núcleo, mientras que en las ejecuciones nativas se dispone de todos los núcleos, pero se recomienda dejar uno libre para el sistema operativo.

2.2.3. Modelos de programación paralela

Antes de hablar de modelos de programación paralela se debe pensar en elos modelos de paralelismo a utilizar.En este proyecto se utilizarán los modelos de paralelismo de bucle y de paralelismo de pareas.

En el modelo de paralelismo de bucle al llegar la ejecución a un bucle paralelo se dividen las iteraciones entre los diferentes hilos de ejecución. Este modelo utiliza el esquema de paralelismo *Fork-Join*. El esquema consiste en ejecutar el código de manera secuencial hasta llegar a un punto concreto, donde se divide el trabajo en varias partes y se distribuye entre diferentes threads. Una vez han acabado de trabajar todos los threads, la ejecución del algoritmo vuelve a continuar de manera secuencial, pudiendo haber más divisiones de trabajo en el futuro (ver Figura 3).

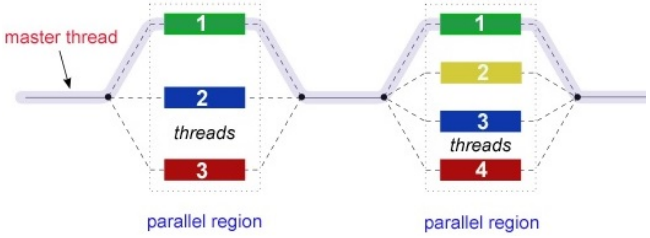

Figura 3: Esquema de paralelismo Fork-Join

En cambio, en el modelo de tareas, un hilo trabaja como generador (llamado *Master*) y el resto actúan como consumidores (llamados *Workers*). Al generarse una tarea, la nueva tarea se añade a una lista de tareas. Si los consumidores acaban de ejecutar una tarea, o están en espera, consultan la lista de tareas y, si no está vacía, se les asigna trabajo a realizar (ver Figura 4).

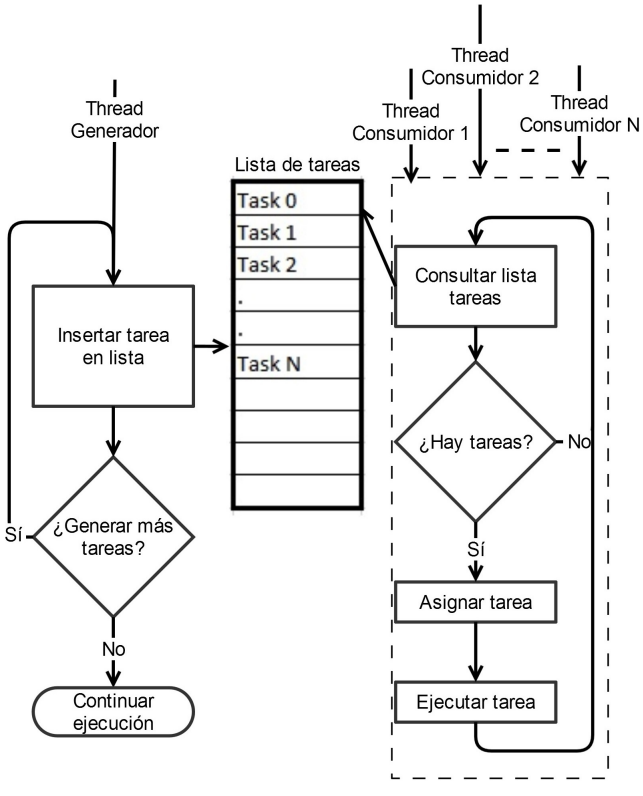

Figura 4: Esquema de paralelismo de tareas

Entre los diferentes modelos de programación paralela con soporte para tareas para el lenguaje C++ destacan por su popularidad [13]: Intel Threading Building Blocks, OpenMP y Cilk Plus. El proyecto utilizará Cilk Plus y OpenMP, que están integrados en los compiladores de las máquinas disponibles

Cilk Plus asigna una cola de tareas a cada hilo de ejecución (o *worker*) disponible, y reparte las tareas entre diferentes colas. Los *workers* creados por Cilk Plus no se encuentran ligados a un thread hardware y pueden migrar a otro núcleo de procesamiento. Cuando un hilo de ejecución se queda con la cola vacía, y los demás no, selecciona a una víctima para robarle trabajo [14].

Por otro lado, OpenMP tiene una cola compartida de tareas. En el momento de asignar una tarea a un thread, se entra en una zona crítica, la tarea es asignada y eliminada de la lista. Finalmente, el thread sale de la zona crítica y ejecuta la tarea [15][16].

3 OBJETIVOS

Un objetivo principal de este proyecto es desarrollar e implementar un algoritmo para resolver el problema de las N-Reinas, codificado como CSP, para sistemas con múltiples núcleos de procesamiento.

Otro objetivo principal es evaluar y comparar los resultados obtenidos utilizando diferentes modelos de paralelismo y diferentes sistemas paralelos.

Además de los objetivos principales, también se pretende desarrollar e implementar un algoritmo paralelo equivalente apto para el coprocesador Intel Xeon Phi.

4 METODOLOGÍA

Con el fin de realizar este proyecto se utilizó la metodología Waterfall. Esta metodología se considera apta para el proyecto al ser necesario el cumplimiento estricto de cada una de las fases para poder avanzar a la siguiente. Además, esta metodología se utiliza en proyectos con objetivos bien definidos al principio y con bajas probabilidades de sufrir cambios, requerimientos que se cumplen en este proyecto.

En cada fase del proyecto con desarrollo de algoritmos se ha contado con pruebas de rendimiento y al finalizar las fases de desarrollo de los algoritmos para CPU se realizaron pruebas de escalabilidad.

Las fases de desarrollo del proyecto son las siguientes:

- Desarrollo del algoritmo paralelo con el esquema de paralelismo Fork-Join.
- Desarrollo utilizando el esquema de paralelismo de tareas. Primero para el modelo de programación paralela OpenMP y después para Cilk Plus
- Desarrollo de versiones aptas para coprocesadores Intel Xeon Phi.

5 DESARROLLO

Este apartado se divide en el análisis del algoritmo inicial y la paralelización del algoritmo.

5.1. Análisis del algoritmo secuencial

Este algoritmo se utilizará para resolver el problema de las N-Reinas, codificado como un problema de satisfacción de restricciones. El algoritmo se basa en el recorrido de un árbol de búsqueda de soluciones en profundidad con backtracking y usando arco-consistencia para podar alternativas. Las restricciones son binarias y el algoritmo utiliza operaciones a nivel de bit para reducir la cantidad de memoria necesaria.

El algoritmo al empezar selecciona la variable (var) 0 y el nivel (level) 0 del árbol de búsqueda. Una vez acabada la inicialización se llama a la función de búsqueda de soluciones. Esta función acepta de parámetros de entrada el nivel y el estado de las variables, y sigue los siquientes pasos del Algorithm 1:

3. Se comprueba si el estado (state) actual es válido. Si no es válido se devuelve 0, es decir, 0 soluciones encontradas.
4. Se comprueba si el nivel (level) es el máximo permitido. Si el nivel es el máximo permitido y el estado es válido, tenemos una solución y devuelve 1.
5. Se selecciona una variable libre a la variable (var) a utilizar.
6. Se inicializa el contador de soluciones (s) a 0.
7. Se realizan los siguientes dos puntos por cada valor de la variable (var) seleccionada:
8. Se calcula el nuevo estado (state$'$) con la función de arc_consistency.
9. Se llama recursivamente a la función de búsqueda de soluciones, con los parametros de entrada del nuevo estado (state$'$) y el nivel incrementado en uno (level+1). Y se suma al contador de coluciones (s) las soluciones encontradas por la función.
10. Se devuelve el valor del contador de soluciones (s).

Algorithm 1 Búsqueda de soluciones

```
1: procedure int search_solutions( level, state )
2:     if !feasible(state) then return 0
3:     if level == N then return 1
4:     var ← select_free_variable(state)
5:     s ← 0
6:     for val in available_values(var, state) do
7:         state' ← arc_consistency(var, val, state)
8:         s ← s + search_solutions(level + 1, state')
       return s
```

La complejidad del algoritmo es $O(N!)$, donde N es el número de reinas, esta complejidad se obtiene al tener N variables con N valores pero los valores libres al asignar un valor a una variable se reducen en uno. Pero esta complejidad se reduce considerablemente al utilizar en el algoritmo técnicas de poda.

La función de arco-consistencia utiliza una lista de restricciones binarias de tamaño N^3. Esta función, por cada

valor de cada variable, realiza N comparaciones entre la lista y los valores de las demás variables, obteniendo un resultado para la poda. Debido a las comparaciones a realizar, la arco-consistencia tiene una complejidad de N^3, y resulta ser la función con una demanda de tiempo de aproximadamente el 93 % del total.

En el algoritmo se define la estructura de datos BitVector. Esta estructura se define para poder modificar el tamaño de las estructuras de datos de cada reina, sus dominios, las restricciones binarias y los valores de arco-consistencia, según el número de reinas. Con tal de reducir la cantidad de memoria necesaria se trabaja a nivel de bit y el tamaño de Bitvector es$N + 1$ bits, donde N es el número de reinas.

La lista de restricciones binarias es un cuello de botella si el número de reinas es elevado, ya que el tamaño de la lista en bytes es $(N^3)*$ tamaño$_de_BitVector$. Si se utiliza un número pequeño como 17 reinas el tamaño de la lista es 19,2 KBytes, pero en el caso de utilizar 200 reinas se necesitan 244,14 MBytes y si se requiere de 500 reinas se necesitan 7,5 GBytes.

Para hacer *backtracking* es necesario mantener los resultados previos de arco-consistencia obtenidos durante la búsqueda en el árbol de posibles soluciones y se utilizan N vectores de bits (Bitvector) en cada nivel. Con este fin se utiliza una lista de tamaño $= (N^2+N)*sizeof(BitVector)$.

5.2. Paralelización del algoritmo

Para poder paralelizar el algoritmo se ha dividido la búsqueda de soluciones de la siguiente manera:

- Un hilo de ejecución llamado *Master* explora el árbol de búsqueda de forma secuencial hasta una profundidad determinada por el usuario y genera trabajo (continúa la búsqueda a partir del estado generado) para el resto de hilos.
- Un grupo de hilos se divide el trabajo generado por el *Master*.

Primero se explica la estrategia de utilizar un bucle paralelo, después la versión realizada mediante el uso de tareas y, para finalizar, la versión híbrida para ejecutar el algoritmo en CPU y en el acelarador Xeon Phi.

5.2.1. Paralelización mediante un bucle paralelo

En esta aproximación, al llegar la ejecución secuencial a un nivel determinado por el usuario, un bucle reparte los valores libres de la Variable seleccionada entre los diferentes hilos de ejecución. Cambiando el bucle del Algorithm 1 por un bucle paralelo en el nivel seleccionado, siguiendo el esquema de paralelismo Fork-Join (ver Figura 3). Debido a la poda irregular realizada en el árbol de búsqueda cada thread tiene cargas de trabajo diferente. La parte secuencial del algoritmo debe esperar a la finalización de todas las iteraciones del bucle paralelo para poder continuar.

Esta aproximación tiene una parte crítica: la actualización del número de soluciones. Debido a tener varios threads ejecutándose concurrentemente, la variable se podría actualizar de manera incorrecta si se lee y escribe de manera concurrente. Para evitar tener un valor incorrecto se limita el acceso de esta variable a un único thread utilizando una región crítica.

Para cada thread se necesita memoria adicional para realizar la búsqueda en el subárbol que tiene asignado, aproximadamente $(N^2+3N)*BitVector$, donde N es el número de reinas.

Las ventajas de esta implementación son las siguientes:

- Fácil implementación.
- Se necesita poca memoria extra.

Las desventajas son:

- Paralelismo de threads limitado por el número de valores libres de la variables escogida.
- Paralelismo únicamente en la zona del bucle paralelo.
- Mal balanceo de carga al tener cada iteración cargas de trabajo diferentes.

5.2.2. Paralelización mediante tareas paralelas

En esta aproximación se intenta aumentar el paralelismo y mejorar el balanceo de carga en el proceso de búsqueda.

La parte secuencial del algoritmo, una vez llega al nivel a paralelizar, guarda en una lista la información necesaria para poder continuar con la búsqueda. Cada valor de la lista puede ser utilizado K veces, donde K es el número de valores libres (ramas) de la Variable escogida a ese nivel. Después el programa crea tantas tareas como valores libres tenga la Variable seleccionada y continua explorando. A su vez, los threads disponibles en el sistema consumen las tareas creadas de manera concurrente, permitiendo un mayor uso de los recursos disponibles.

En este modelo también se limita con una región crítica el acceso de la variable que contiene el número de soluciones. Además, se ha introducido una nueva zona crítica, donde se evalúa si el valor actual de la lista se ha utilizado K veces, en caso afirmativo se utilizará la información del siguiente valor de la lista, donde K es el número de ramas.

De igual manera que la estrategia anterior necesitaba memoria extra, ésta necesita la misma memoria adicional por thread. Pero necesita más memoria para los datos que utilizarán las tareas: su tamaño dependerá de los nodos explorados en el nivel seleccionado. Al utilizar poda no es posible saber a priori el número de nodos explorados y se requiere hacer una aproximación que reserva más memoria de la necesaria.

En la aproximación realizada si se utilizan 17 reinas y se paraleliza a profundidad 3 se reservan 880 KBytes, de los cuales sólo son necesarios 732 KBytes. En el caso de paralelizar a nivel 6 la diferencia aumenta, de los 1135 MBytes reservados sólo son necesarios 311 MBytes. Los valores de la memoria necesaria se obtienen al finalizar la ejecución y mostrar cuantos elementos de la lista se han utilizado.

Las ventajas de esta implementación son:

- El thread máster no espera a los demás threads para continuar su trabajo.
- Mejor balanceo de carga al eliminar la necesidad de sincronizar en cada nodo del árbol en el nivel donde se paraleliza la búsqueda.

Las desventajas son:

- El uso de memoria extra depende de la profundidad a la cual se paraleliza.
- Se necesitan dos zonas críticas en lugar de una.

5.2.3. Paralelización híbrida

Si se quiere utilizar el Xeon Phi como coprocesador las anteriores estrategias no son aptas. En el caso del bucle paralelo, al limitar el paralelismo de threads al número de valores libres, se obtiene una cantidad insuficiente de trabajas independientes comparado con el número de threads hardware del Phi.

En cambio la estrategia de tareas resultaría viable, pero no hay soporte para consumir concurrentemente las tareas en el Phi y en el sistema host.

Finalmente, se ha optado por utilizar el algoritmo del modelo de tareas pero modificarlo de la siguiente manera:

- Se introducen los elementos en la lista de información para las tareas, pero sin generar tareas. Se recorre de manera secuencial el árbol completo limitando la profundidad máxima de exploración.
- Se determina la cantidad de elementos de la lista a ejecutar por el host y por el coprocesador con el objetivo de repartir el trabajo.
- Se crea una zona paralela, en el host y el en Phi, al finalizar la parte serie del algoritmo y se consumen los elementos de la lista.

Este modelo tiene los mismos requerimientos de memoria que la versión con el modelo de tareas.

Las ventajas de esta implementación son:

- Buen balanceo de carga en la parte paralela
- Permite aumentar la capacidad de cómputo al utilizar el coprocesador Xeon Phi.
- El uso de memoria extra depende del nivel al cual se paraleliza.
- Dos zonas criticas
- No hay paralelismo hasta finalizar la parte serie del algoritmo.

6 EXPERIMENTACIÓN

Las experimentaciones con las diferentes versiones del algoritmo se han realizado en las máquinas de la siguiente tabla (Cuadro 1):

Todas las máquinas, menos **Batman**, disponen de procesadores Intel con *HyperThreading*, permitiendo una utilizar instrucciones independientes de diferentes threads para ocultar las esperas producidas por las latencias. Si el rendimiento del algoritmo secuencial tiene problemas con esperas, por fallos de caché u otras latencias, utilizando dos threads por núcleo se puede obtener una mejora de rendimiento superior al número de núcleos de procesamiento del sistema.

La máquina **Aolin** , al únicamente disponer de un socket, es de arquitectura UMA, es decir, la latencia de los accesos de memoria es la misma para todos los núcleos de cómputo. En el caso de la máquina **Aoclsd**, que tiene de una arquitectura NUMA pero durante el periodo de experimentación estaba configurada con una emulación de UMA.

Hardware		
Aolin	CPU	Intel i7-950
	№Núcleos	1 socket x 4 = 4 cores
	№Threads H/W	4 x 2 threads = 8 threads
Aoclsd	CPU	Intel Xeon E5645
	№Núcleos	2 socket x 6 = 12 cores
	№Threads H/W	12 x 2 threads = 24 threads
Penguin	CPU	Intel Xeon E5-4620
	№Núcleos	4 socket x 8 = 32 cores
	№Threads H/W	32 x 2 threads = 64 threads
Batman	CPU	AMD Opteron 6376
	№Núcleos	4 socket x 16 = 64 cores
	№Threads H/W	64 x 1 threads = 64 threads
Sandman	CPU	Intel Xeon E5-2620
	№Núcleos	2 socket x 6 = 12 cores
	№Threads H/W	12 x 2 threads = 24 threads
	Coprocesor	Intel Xeon Phi 7120
	№Núcleos Phi	61 cores
	№Threads H/W Phi	61 x 4 threads = 244 threads

Cuadro 1: Hardware de las máquinas utilizadas

Las demás máquinas tienen una de arquitectura NUMA: cada socket tiene una memoria asociada y acceder a la memoria asociada a otro socket incrementa el tiempo de respuesta. Para aprovechar al máximo las características de estos sistemas se realiza una implementación de la versión de tareas adaptada a sistemas NUMA, tando con OpenMP como con Cilk Plus. En esta nueva versión la estructura de datos a utilizar por cada *worker* la declara el propio *worker* en lugar del thread máster.

Todas las ejecuciones realizadas utilizan 17 reinas, menos en las máquinas **Aolin**, que utilizan 16 reinas, debido a su menor capacidad de cómputo y a utilizar un sistema de colas con un menor tiempo máximo de ejecución.

7 RESULTADOS

En este apartado se explicarán los resultados obtenidos en diferentes máquinas y se extraerán conclusiones generales de los resultados.

Se debe tener en cuenta que la versión con bucle paralelo (llamada FOR en las gráficas) únicamente se utilizará en la computadora Aolin debido a sus desventajas y limitaciones. Las ejecuciones del modelo de tareas, al utilizarse en la mayoría de gráficas, se las nombrará según el modelo de programación paralela utilizado. Siendo OpenMP la ejecución de tareas con OpenMP y Cilk Plus la ejecución con tareas y Cilk Plus.

7.1. Resultados iniciales de rendimiento

A continuación se analiza el impacto de aumentar el nivel del árbol a partir del cual se distribuye la búsqueda en paralelo (ver Figura 5),. En estos experimentos se utilizan ocho threads.

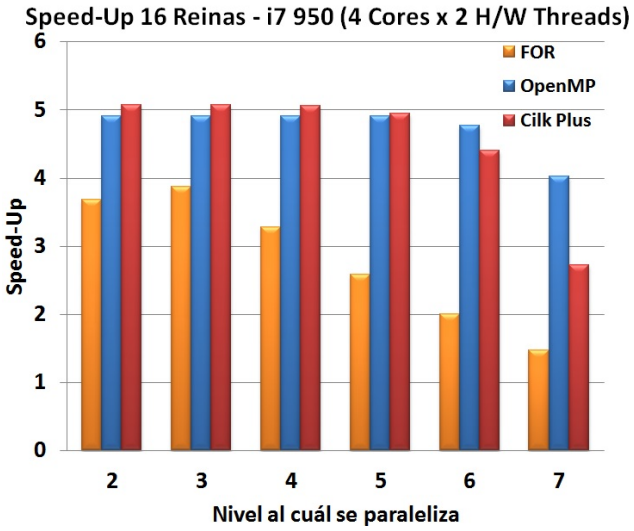

Figura 5: Speed-Up respecto a la ejecución serie. El nivel varía entre 2 y 7.

Se puede observar que la versión con bucle paralelo en ningún momento obtiene un rendimiento adecuado. El Speed-Up obtenido desde la profundidad 5 no compensa el uso de hardware adicional. Esto se debe a la mala estrategia que sigue el algoritmo al tener partes de código serie y mal balanceo de carga en las zonas paralelas, desaprovechando los recursos del procesador (ver Figura 6). Otro motivo del mal rendimiento es el incremento en el número de instrucciones para gestionar las regiones paralelas.

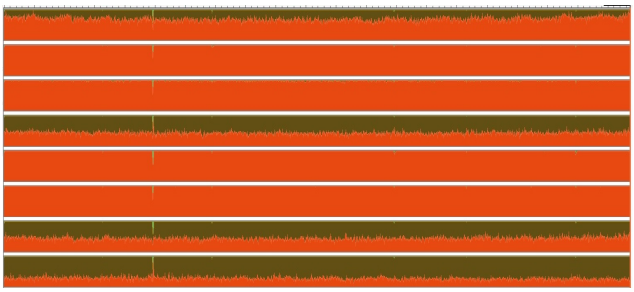

Figura 6: Profiling con VTune. Cada fila corresponde a un núcleo de cómputo. Las franjas rojas corresponden a esperas, mientras que las zonas marrones a utilización de CPU

En el caso de las versiones de tareas el rendimiento no varía de manera significativa al variar la profundidad del árbol de búsqueda entre el nivel 2 y el 5. A partir del nivel 5 Cilk Plus pierde rendimiento de manera notable. Esto se debe a un incremento en el número total de instrucciones ejecutadas por el programa, llegando a un incremento del 72.68 % en el nivel 7. Ente incremento viene generado principalmente por las funciones de Cilk Plus para gestionar las tareas. Para finalizar, la pérdida con OpenMP se centra en el nivel 7 a causa de tener menos trabajo los workers y más trabajo el hilo generador. El incremento en el número de instrucciones en este caso tiene poco impacto, al ser del 5.37 %.

En la Figura 7 se puede ver la escalabilidad del rendimiento al utilizar un número creciente de threads. En los casos de 2, 3 y 4 threads, cada uno de ellos se asigna a un núcleo diferente, y en el caso de 8 threads se habilita el *HyperThreading*, ejecutándose dos threads por núcleo (ver Figura 7). Se ha escogido paralelizar la búsqueda a partir del nivel 3 al ser el mejor para todas las versiones según la Figura 5.

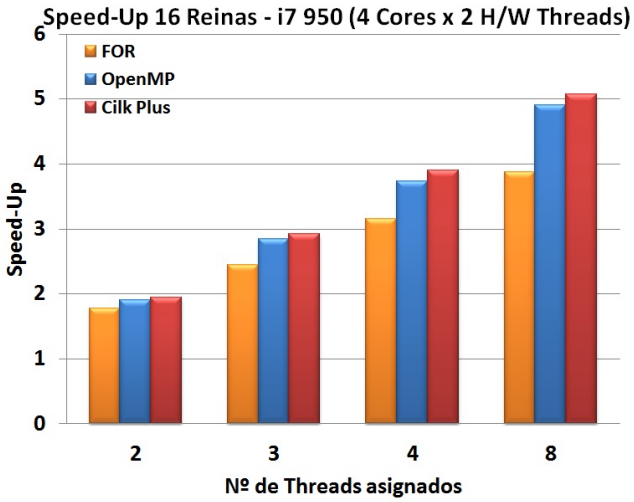

Figura 7: Escalabilidad del rendimiento incrementando el número de hilos de ejecución.

Dados estos resultados, y los de la anterior gráfica (ver Figura 5), se puede afirmar que la optimización que paraleliza con un bucle paralelo es la peor de todas sin importar los recursos utilizado. Esto es debido a carecer de suficiente paralelismo y de un buen balanceo de carga. En cambio, las versiones que utilizan un modelo de tareas son mucho más eficaces, obteniendo un Speed-Up próximo al número de cores utilizados. Con *HyperThreading* el aumento de rendimiento supera el número de núcleos del sistema al maximizar el uso de los recursos de cómputo escondiendo las latencias debidas a zonas críticas y dependencias de datos.

7.2. Resultados de escalabilidad en sistemas multisocket

En este apartado se analiza la escalabilidad en el rendimiento de la implementación con el modelo de tareas. Se utilizan sistemas con múltiples sockets y arquitecturas NUMA. Para reducir la cantidad de figuras a mostrar se analizan los resultados de la máquina **Penguin** (ver Figura 8), dado que los resultados son similares en las máquinas con procesadores Intel, y los de la máquina **Batman** (ver Figura 9).

En estas ejecuciones las computadoras **Penguin** y **Batman** tienen una arquitectura de memoria NUMA. Al utilizar dicha arquitectura se debe tener en cuenta que la localización de los datos puede incrementar la latencia para acceder a memoria. Con tal de minimizar los overheads de memoria es importante que cada núcleo de computo utilice la memoria más cercana.

En el algoritmo del modelo de tareas el thread Master crea, para cada núcleo, la estructura de datos necesaria para explorar los subárboles. En el caso de la versión *NUMA-Aware* cada núcleo crea la estructura de datos en su memoria más cercana al iniciarse cada thread.

Se puede ver como la implementación que usa Cilk Plus tiene variaciones leves de rendimiento entre la versión ini-

cial y la adaptada para NUMA. Gracias a que Cilk Plus dispone de un planificador dinámico, para balancear mejor la carga entre los diferentes núcleos, se pueden mitigar los efectos de la arquitectura NUMA de forma automática, sin que el programador deba hacer nada especial.

Por otro lado, con OpenMP siempre es mejor la versión *NUMA-Aware*. A causa de no permitir migrar threads de manera automática, según la localización de los datos, los efectos de la arquitectura NUMA son considerables, y el programador tiene que optimizar el código de forma explícita para trabajar en estas arquitecturas con OpenMP.

En las dos máquinas con procesadores Intel (Figura 8) se ha llegado a conseguir un Speed-Up mayor al número de núcleos físicos del sistema. Gracias a la tecnología *HyperThreading* que ha conseguido maximizar la utilización de los recursos al esconder latencias.

Con la arquitectura AMD, en los casos de 8, 16 y 32 threads, se ha llegado a una mejora de rendimiento superior al incremento de hardware utilizado (ver Figura 9).En cambio, al usar 64 threads, se pierde escalabilidad, y la utilización de los recursos, en la versión OpenMP NUMA-Aware, baja al 85.6 %.

Hay que destacar que la arquitectura de los procesadores AMD organiza los núcleos de cómputo en módulos de dos cores y dentro del módulo se comparten diferentes recursos. Algunos de los recursos compartidos son las unidades de vectorización, las unidades *Fetch* y *Decode*, y la caché de instrucciones, entre otras (ver Figura 10) [17][18][19][20].

Las causas exactas de esta pérdida no son fáciles de encontrar al no disponer de un profiler como VTune en arquitecturas AMD. Descartando los fallos de caché al obtener resultados similares con ejecuciones de 32 y 64 núcleos, y también problemas de vectorización al realizar pruebas deshabilitando las instrucciones SIMD y obteniendo mejoras de rendimiento similares, las causas más probables pueden ser los tiempos de espera debidos a las zonas críticas del algoritmo y también las perdidas debidas a los recursos compartidos en los módulos de los procesadores AMD.

Con la versión *Numa-aware* de Cilk Plus el rendimiento ha empeorado, al contrario que la misma versión con OpenMP. La causa probable es que Cilk Plus es propiedad de Intel y la arquitectura AMD puede que no sea adecuada para este modelo de programación paralela.

7.3. Resultados Xeon Phi

En este apartado primero se mostrarán los resultados obtenidos con el coprocesador Xeon Phi y el esquema de tareas con OpenMP Numa-Aware. Después se mostrarán los resultados del modelo híbrido.

Las comparaciones de aumento de rendimiento se realizan contra el algoritmo secuencial ejecutado en el host.

Se debe tener en cuenta que las ejecuciones que utilizan el coprocesador Xeon Phi han utilizado un núcleo de procesamiento menos de los disponibles debido al núcleo utilizado por el sistema operativo del Phi.

Los resultados obtenidos con la versión de tareas en el Xeon Phi (ver Figura 11) permiten observar cómo utilizando únicamente un hilo de ejecución en cada núcleo de procesamiento se obtiene un rendimiento bajo, comparado con utilizar el máximo de threads permitidos, donde casi se consigue duplicar el Speed-Up.

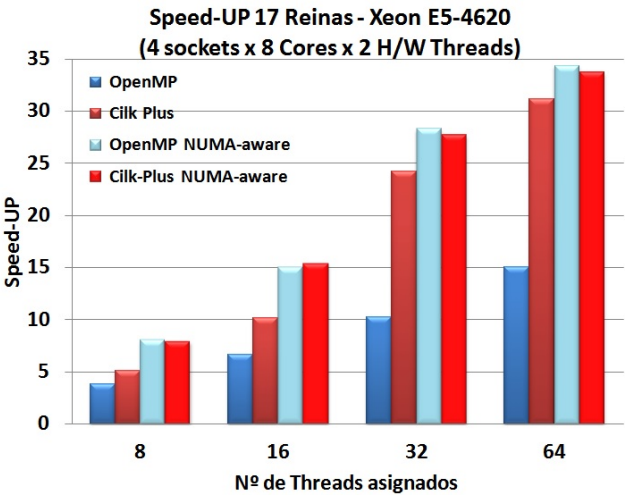

Figura 8: Escalabilidad en **Penguin**. Con 8 threads se utiliza un único socket. De 8 a 32 threads se utiliza un núcleo por thread. Con 64 hilos se utilizan 2 threads por core.

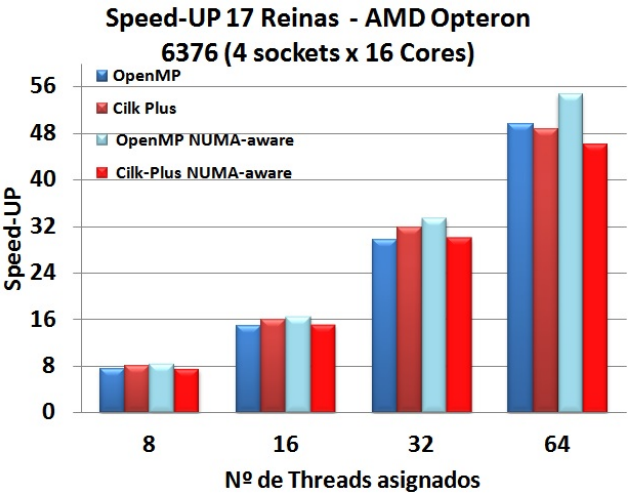

Figura 9: Escalabilidad en **Batman**. De 8 a 32 threads se utiliza un núcleo de cada módulo de dos cores. Con 64 threads se utilizan los dos núcleos de cada módulo.

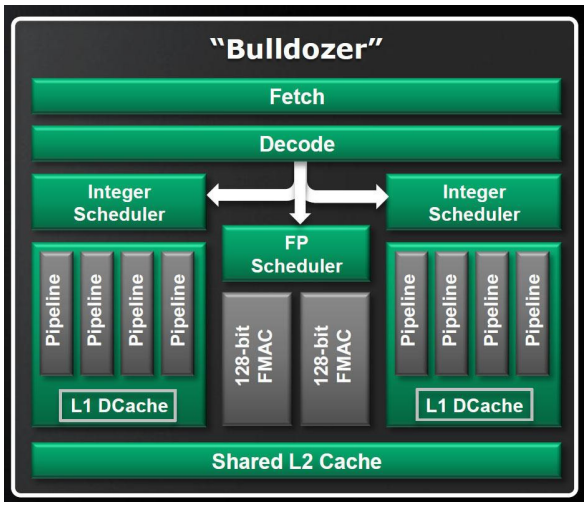

Figura 10: Un módulo de la arquitectura AMD[17]

Teóricamente, en el Xeon Phi se puede obtener un incremento de siete veces la potencia de cálculo de un Xeon de

8 núcleos. Esto se traduce en un Speed-Up de aproximadamente 56 respecto la versión serie, aunque solamente se consigue un 17.26 (un 30,8 % del rendimiento teórico). Esto se debe en parte a tener un número pequeño de reinas, se utilizan vectories SIMD de pequeño tamaño, y no se aprovecha al máximo la capacidad de SIMD del Phi, que cuenta con vectores del doble de tamaño que los procesadores Xeon.

Si se comparan los resultados de la Figura 11 y de la Figura 12 se puede ver un ligero incremento del rendimiento entre la versión de tareas y la versión híbrida ejecutándose únicamente en el Phi. Esta mejora se obtiene al ejecutar la parte serie en un núcleo de procesamiento más potente, perteneciente al host, consiguiendo recorrer toda la parte serie antes de que el Phi consiga generar un número de tareas que utilice de manera apropiada la gran cantidad de threads que contiene.

La versión híbrida en el host tiene una pequeña pérdida de rendimiento, del 2,5 %, respecto a la versión de tareas en el host. Esta pequeña diferencia se debe a tener una parte secuencial pequeña al paralelizar al nivel 3.

En la ejecución híbrida se consigue llegar al 95.7 % del aumento de rendimiento teórico, el máximo se obtiene con la suma del aumento del rendimiento en el host y en el Phi (13,53 + 17,57 = 31,1). Este resultado se puede considerar satisfactorio al no haber ninguna comunicación entre los dos sistemas para balancear carga dinámicamente.

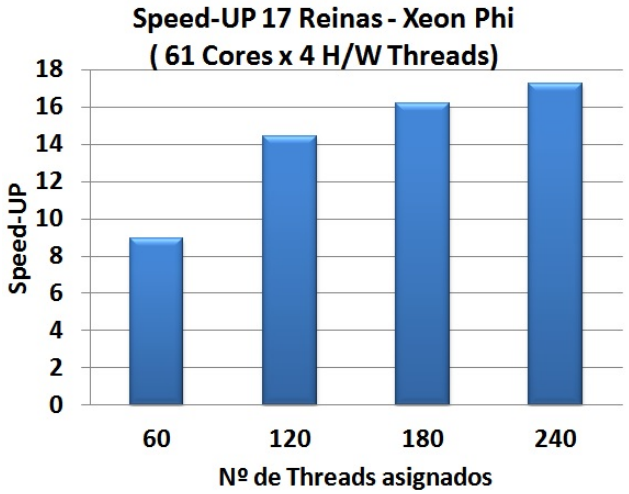

Figura 11: Rendimiento del modelo de tareas en Xeon Phi comparado con la versión secuencial en el host. El host es Sandman, con Intel Xeon E5-2620 v2. Con 60 threads se utiliza un thread por núcleo, con 120 dos por core, con 180 tres y con 240 se utilizan 4 threads por núcleo.

8 Conclusiones

Una vez finalizado el proyecto se puede comprobar que se han cumplido tanto los objetivos principales como el objetivo secundario, y, además, la planificación no se ha visto modificada a lo largo del proyecto, demostrando que la metodologia *Waterfall* es adecuada para el proyecto.

Los resultados obtenidos demuestran que los modelos de paralelización utilizados son adecuados para la resolución del problema de satisfacción de restricciones al obtener aumentos de rendimiento cercanos e incluso superiores al número de núcleos de cómputo utilizados.

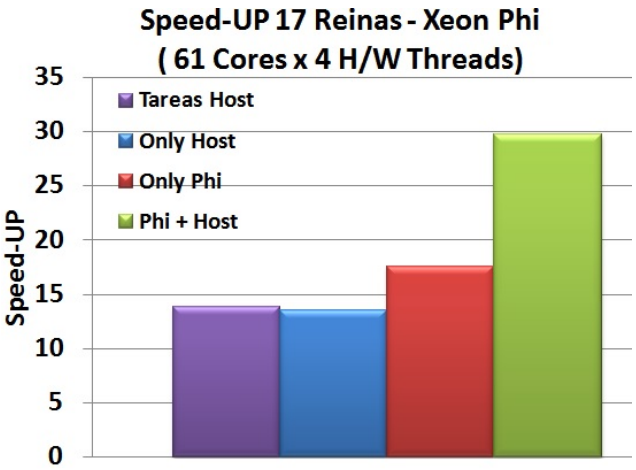

Figura 12: Rendimiento de la versión híbrida comparada con la versión secuencial en el host. El host es Sandman, con Intel Xeon E5-2620 v2. En cada caso se utiliza el máximo número de threads que permite el sistema.

En el caso de utilizar *HyperThreading* el aumento de rendimiento ha superado la cantidad de núcleos utilizados, demostrando que esta tecnología cumple con la finalidad de aumentar el rendimiento, aprovechando instrucciones de diferentes threads para minimizar tiempos de espera.

El coprocesador Intel Xeon Phi ha permitido aumentar la capacidad de cómputo del sistema pero su utilización no ha sido en la proporción esperada. Pese a ser un coprocesador fácil de utilizar y programar, se deben llevar a cabo optimizaciones en el algoritmo si se quiere obtener un rendimiento óptimo. Con los resultados obtenidos se puede afirmar que el problema de no obtener el Speed-Up esperado utilizando el Phi recae en la implementación del algoritmo y no en los modelos utilizados, que obtienen buenos resultados de escalabilidad.

9 Líneas futuras

Se pueden realizar varios cambios en el futuro. Estos cambios son los siguientes:

1. Limitar el número de tareas a ejecutar concurrentemente. Limitando el número de tareas se puede reducir el tamaño de la lista con la información de las tareas, reduciendo la memoria extra necesaria.

2. Utilizar restricciones All-Different, que son más relevantes para el campo de la Inteligencia Artificial. Este tipo de restricciones no necesitan de una lista con las restricciones, y requieren menos trabajo de cómputo y menos capacidad de memoria.

3. Realizar una implementación secuencial y una versión paralela equivalente para la búsqueda de la primera solución o de las X primeras soluciones.

Realizando las líneas futuras 2 y 3 se podría realizar una implementación adecuada para ser ejecutada en GPUs y también para mejorar el rendimiento del Xeon Phi al tener un mayor número de reinas.

AGRADECIMIENTOS

Me gustaría agradecerle a mi tutor Juan carlos Moure su ayuda durante el desarrollo del proyecto y soportarme hasta su finalización.

Quisiera también darle las gracias a Toni Espinosa y al pequeño grupito de alumnos de máster y doctorado que me han ayudado a resolver problemas relacionados con las máquinas utilizadas.

REFERENCIAS

[1] P. G. y M. F. Kowalik, *Multi-Core Processors: New Way to Achieve High System Performance*. PARELEC 2006: Proceedings of the International Symposum on Parallel Computing in Electrical Engineering, 2006.

[2] T. Spyrou, "Why parallel processing? why now? what about my legacy code?." https://software.intel.com/en-us/blogs/2009/08/31/why-parallel-processing-why-now-what-about-my-legacy-code, 2009. Último acceso: 7 Marzo 2015.

[3] V. Kumar, *Algorithms for Constraint Satisfaction Problems: A Survey*. AI Magazine, vol. 13, nº 1, 1992.

[4] E. Tsang, *Foundations of Constraint Satisfaction*. London: Academic Press Limited, 1993.

[5] S. C. B. y C. N. Potts y B. M. Smith, *Constraint satisfaction problems: Algorithms and applications*. European Journal of Operational Research 119, 1999, 557-591, 1999.

[6] H. Bodlaender, *The rules of chess*. http://www.chessvariants.org/d.chess/chess.html, 2000.

[7] H. Bodlaender, *M. McCool, A. D. Robison y J. Reinders*. Waltham: Morgan Kauffmann, 2012.

[8] J. L. H. y D. A. Patterson, *Computer Architecture: A Quantitative Approach IV*. Morgan Kauffmann, 2007.

[9] A. Valles, "Performance insights to intel hyper-threading technology." https://software.intel.com/en-us/articles/performance-34insights-to-intel-hyper-threading-technology, 2009. Último acceso: 25 Abril 2015.

[10] J. J. y J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Waltham: Morgan Kauffmann, 2013.

[11] Intel, *Introducing the Intel Xeon Phi Coprocessor, Architecture for Discovery*. 2012.

[12] S. J. P. y C. J. Hughes y M. Smelyanskiy y S. A. Jarvis, *Exproring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors*. Intel Corporation, 2013.

[13] A. L. y A. Gilman, *A Comparative Analysis of Parallel Programming Models for C++*. ICCGI 2014: The Ninth International Multi-Conference on Computing in the Global Information Technology, 2014.

[14] Intel, "Faq: Cilk plus task scheduler." https://www.cilkplus.org/faq/20, 2012. Último acceso: 25 Abril 2015.

[15] K. A. H. y A. D. Malony y S. Shende y D. W. Jacobsen, *Integrated Measurement for Cross-Platform OpenMP Performance Analysis*. Using and Improving OpenMP for Devices, Tasks, and More, Springer, 2014.

[16] X. T. y P. Unnikrishnan y X. Martorell y E. Ayduadé y R. Silvera y G. Zhang y E. Tiotto, *OpenMP Tasks in IBM XL Compilers*. CASCON '08: proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: meeting of minds, 2008.

[17] C. Angelini, "Amd bulldozer review: Fx-8150 gets tested." http://www.tomshardware.com/reviews/fx-8150-zambezi-bulldozer-990fx,3043-3.html, 2011. Último acceso: 25 Abril 2015.

[18] C. Webster, "Amd bulldozer - what's a module, what's a core?." http://www.bit-tech.net/hardware/cpus/2011/10/12/amd-fx-8150-review/2, 2011. Último acceso: 25 Abril 2015.

[19] S. Walton, "Amd fx-8350 and fx-6300 piledriver review." http://www.techspot.com/review/586-amd-fx-8350-fx-6300/, 2012. Último acceso: 25 Abril 2015.

[20] M. Pollice, "Opteron 6300 series launch: An incremental upgrade on an outdated platform." http://www.vrworld.com/2012/11/05/opteron-6300-series-launch-an-incremental-upgrade-on-an-outdated-platform/, 2012. Último acceso: 25 Abril 2015.

APÉNDICE

En las secciones del apéndice se muestran resultados numéricos de las ejecuciones. Las ejeuciones con nombre FOR se refieren al algoritmo con paralelismo de bucle. Las versiones OpenMP, Cilk Plus, y las NUMA (NUMA-Aware) se refieren al modelo de programación utilizado en el algoritmo de tareas.

En los resultados se pueden distinguir tres parámetros de ejecución:

- N. Indica el número de reinas utilizado en la ejecución.
- par_level. Indica el nivel al que se ha paralelizado. En el caso de no estar determinado se establece el nivel en tres.
- threads. Indica el número de threads utilizados. En las ejecuciones secuenciales se considera 1. En caso de no estar indicado el valor, se considera el máximo de threads hardware de la máquina.

Las métricas utilizadas en esta sección se han conseguido con el comando perf y son las siguientes:

- seconds. Segundos totales de la ejecución.
- instructions (G). Número de instrucciones en 10^9 intrucciones.
- cycles (G). Número de ciclos en 10^9 instrucciones.
- IPC. Instrucciones por ciclo de cada thread.
- Speed-UP. Indica el aumento de rendimiento frente a la versión secuencial.
- CPUs utilized. Media del número de threads utilizados durante toda la ejecución del programa.
- Work efficiency. Indica la eficiencia según el número de instrucciones de la ejecución. Se obtiene de dividir el número de instrucciones de la versión secuencial entre las instrucciones de la ejecución seleccionada.

A.1. Resultados en la computadora Aolin

Primero se muestran los resultados según el nivel al que se paraleliza y después la mejora de rendimiento según el número de hilos de ejecución utilizados.

N = 16	par_level	seconds	instructions(G)	cycles(G)	IPC	Speed-Up	CPUs utilized	Work efficiency
Secuencial	-	156,139	756,26	526,41	1,44	1,00	0,998	1,000
OpenMP	2	31,754	754,92	822,56	0,92	4,92	7,976	1,002
OpenMP	3	31,771	754,95	821,83	0,92	4,91	7,965	1,002
OpenMP	4	31,755	755,10	821,56	0,92	4,92	7,966	1,002
OpenMP	5	31,826	756,12	822,71	0,92	4,91	7,959	1,000
OpenMP	6	32,681	762,14	829,00	0,92	4,78	7,810	0,992
OpenMP	7	38,730	796,88	860,35	0,92	4,03	6,836	0,949
Cilk Plus	2	30,756	756,55	795,86	0,95	5,08	7,962	1,000
Cilk Plus	3	30,755	756,57	794,92	0,95	5,08	7,959	1,000
Cilk Plus	4	30,845	759,28	797,97	0,95	5,06	7,966	0,996
Cilk Plus	5	31,564	775,29	816,27	0,95	4,95	7,963	0,975
Cilk Plus	6	35,481	859,18	917,00	0,94	4,40	7,958	0,880
Cilk Plus	7	57,262	1.305,90	1.464,17	0,89	2,73	7,873	0,579
FOR	2	42,424	759,71	757,10	1,00	3,68	5,492	0,995
FOR	3	40,314	788,07	866,46	0,91	3,87	6,614	0,960
FOR	4	47,617	900,67	1.126,39	0,80	3,28	7,285	0,840
FOR	5	60,358	1.075,32	1.559,03	0,96	2,59	7,953	0,703
FOR	6	78,086	1.268,60	2.019,21	0,63	2,00	7,962	0,596
FOR	7	105,873	1.559,49	2.731,99	0,57	1,47	7,946	0,485

Cuadro 2: Resultados según nivel a paralelizar

N = 16	threads	seconds	instructions(G)	cycles(G)	IPC	Speed-Up	CPUs utilized	Work efficiency
Secuencial	-	156,139	756,26	526,41	1,44	1,00	0,998	1,000
OpenMP	2	81,774	754,69	529,98	1,42	1,91	1,995	1,002
OpenMP	3	54,869	754,69	533,52	1,41	2,85	2,994	1,002
OpenMP	4	41,822	754,70	541,49	1,39	3,73	3,987	1,002
OpenMP	8	31,771	754,95	821,83	0,92	4,91	7,965	1,002
Cilk Plus	2	80,498	755,98	521,52	1,45	1,94	1,995	1,000
Cilk Plus	3	53,350	756,02	518,72	1,46	2,93	2,994	1,000
Cilk Plus	4	40,055	756,08	518,77	1,46	3,90	3,988	1,000
Cilk Plus	8	30,755	756,57	794,92	0,95	5,08	7,959	1,000
FOR	2	87,539	760,77	536,70	1,42	1,78	1,884	0,994
FOR	3	63,707	765,19	548,72	1,39	2,45	2,649	0,988
FOR	4	49,504	769,71	564,19	1,36	3,15	3,506	0,983
FOR	8	40,314	788,07	866,46	0,91	3,87	6,614	0,960

Cuadro 3: Resultados según número de threads hardware utilizados

A.2. Resultados en la computadora Penguin

N = 17	Threads	seconds	instructions(G)	cycles(G)	IPC	Speed-Up	CPUs utilized	Work efficiency
Secuencial	-	1.297,752	6.103,24	3.372,85	1,81	1,00	1,00	1,000
OpenMP	8	339,109	6.107,68	7.028,47	0,87	3,83	7,99	0,999
OpenMP	16	195,042	6.108,72	7.787,21	0,79	6,65	15,98	0,999
OpenMP	32	126,788	6.111,07	9.329,16	0,66	10,24	31,96	0,999
OpenMP	64	86,273	6.115,01	12.510,44	0,49	15,04	63,03	0,998
Cilk Plus	8	252,626	6.105,72	5.230,36	1,17	5,14	7,99	1,000
Cilk Plus	16	127,792	6.106,99	5.084,23	1,20	10,16	15,98	0,999
Cilk Plus	32	53,640	6.128,69	3.948,39	1,55	24,19	31,96	0,996
Cilk Plus	64	41,616	6.155,83	5.891,85	1,04	31,18	61,66	0,991
OpenMP NUMA	8	161,286	6.103,86	3.342,31	1,83	8,05	7,99	1,000
OpenMP NUMA	16	86,446	6.104,08	3.448,05	1,77	15,01	15,98	1,000
OpenMP NUMA	32	45,871	6.104,14	3.373,02	1,81	28,29	31,95	1,000
OpenMP NUMA	64	37,782	6.106,83	5.490,15	1,11	34,35	63,34	0,999
Cilk-Plus NUMA	8	163,175	6.103,63	3.378,90	1,81	7,95	7,99	1,000
Cilk-Plus NUMA	16	84,342	6.104,79	3.357,53	1,82	15,39	15,99	1,000
Cilk-Plus NUMA	32	46,842	6.107,39	3.448,14	1,77	27,70	31,95	0,999
Cilk-Plus NUMA	64	38,529	6.114,85	5.402,15	1,14	33,68	61,05	0,998

Cuadro 4: Resultados según número de threads hardware utilizados

A.3. Resultados en la computadora Batman

N = 17	Threads	seconds	instructions(G)	cycles(G)	IPC	Speed-Up	CPUs utilized	Work efficiency
Sec	-	1.685,211	5.547,62	4.351,99	1,28	1,00	1,00	1,000
OpenMP	8	224,376	5.548,46	4.634,98	1,20	7,51	8,00	1,000
OpenMP	16	113,097	5.548,75	4.672,11	1,19	14,90	16,00	1,000
OpenMP	32	56,842	5.548,77	4.688,61	1,18	29,65	32,00	1,000
OpenMP	64	33,955	5.545,78	5.299,22	1,05	49,63	63,93	1,000
Cilk Plus	8	209,847	8.470,11	4.334,79	1,26	8,03	8,00	0,655
Cilk Plus	16	105,052	5.470,00	4.339,42	1,26	16,04	16,00	1,014
Cilk Plus	32	53,017	5.472,16	4.360,18	1,26	31,79	32,00	1,014
Cilk Plus	64	34,556	5.474,40	5.348,55	1,02	48,77	63,37	1,013
OpenMP NUMA	8	199,303	5.547,33	4.116,92	1,35	8,46	8,00	1,000
OpenMP NUMA	16	101,705	5.547,48	4.201,39	1,32	16,57	16,00	1,000
OpenMP NUMA	32	50,426	5.547,47	4.135,19	1,35	33,42	32,00	1,000
OpenMP NUMA	64	30,761	5.547,46	4.772,92	1,16	54,78	63,92	1,000
Cilk-Plus NUMA	8	223,079	5.469,28	4.608,15	1,19	7,55	8,00	1,014
Cilk-Plus NUMA	16	111,599	5.470,29	4.610,18	1,19	15,10	16,00	1,014
Cilk-Plus NUMA	32	56,222	5.472,38	4.635,49	1,18	29,97	32,00	1,014
Cilk-Plus NUMA	64	36,637	5.476,27	5.688,36	0,96	46,00	63,31	1,013

Cuadro 5: Resultados según número de threads hardware utilizados