Información Tecnológica
Vol. 28(5), 65-74 (2017)
doi: 10.4067/S0718-07642017000500009

Desempeño de los Paradigmas Funcional y Orientado a Objetos en Arquitecturas Multicore: Caso de Estudio Haskell-Java

Jorge G. Hoyos y Alex Puertas
Facultad de Ingeniería de Sistemas, Universidad Santo Tomás, Calle 19 No. 11-64, Tunja, Colombia
(e-mail: jorge.hoyos@usantoto.edu.co; alex.puertas@usantoto.edu.co)

Recibido Mar. 7, 2017; Aceptado Abr. 26, 2017; Versión final May. 20, 2017, Publicado Oct. 2017

Resumen

En este trabajo se describe la implementación de paralelismo multicore en los lenguajes de programación Haskell y Java, utilizado dos paradigmas de programación diferentes, el funcional y el orientado a objetos. Se parte de una implementación ya existente del algoritmo solucionador de sudokus, desarrollada en Haskell con un enfoque funcional, a la que se incorporaron elementos de paralelización básica. Luego se construyó su equivalente desde la programación orientada a objetos en el lenguaje Java. Finalmente se realizaron diferentes mediciones para analizar y contrastar el desempeño de las dos implementaciones. De acuerdo al análisis realizado, se llegó a la conclusión que Haskell hace una utilización más óptima que Java de la arquitectura multicore.

Palabras clave: arquitectura multicore; Haskell; Java; programación funcional, programación orientada a objetos

Performance of Functional and Object-Oriented Paradigms in Multicore Architecture: Case Study Haskell-Java

Abstract

This paper describes how multicore parallelism is implemented in the Haskell and Java programming languages, using two different programming paradigms, functional and object oriented. The starting point is an existing implementation of the sudokus solver algorithm, developed in Haskell with a functional approach, to which elements of basic parallelization were incorporated. After that, an equivalent version was constructed from object-oriented programming in the Java language. Finally, different measurements were taken to analyze and contrast the performance of the two implementations. According to the analysis, it was concluded that Haskell makes a more optimal use than Java of multicore architecture.

Keywords: functional programming; Haskell; Java; multicore architecture; object-oriented programming

INTRODUCCIÓN

Con los avances tecnológicos se ha hecho más fácil que el común de la sociedad pueda adquirir equipos de cómputo y dispositivos móviles, que cada vez son más numerosos y se hacen más pequeños y portables (Castro et al., 2016). En los equipos de última generación ya es común encontrar dispositivos integrados y teléfonos móviles que poseen procesadores de varios núcleos, y por ende se están convirtiendo en máquinas paralelas. En la actualidad se evidencia la subutilización del hardware multicore, ya que son muy pocas las aplicaciones capaces de aprovechar esta infraestructura para mejorar su desempeño. Este nuevo escenario propicia el resurgimiento de la programación funcional como el paradigma candidato a sustituir el de orientación a objetos que ha estado vigente en los últimos 20 años (Hoyos y Puertas, 2015). La programación funcional es un paradigma de programación, que se caracteriza por el uso de expresiones y funciones, de manera tal, que un programa es en sí una función, que a su vez, está compuesta de funciones más simples. En este sentido una función puede ser invocada por otra, y el resultado de una función puede ser utilizado como parámetro de entrada de otra función (Gónzalez, 1998).

El paradigma funcional ha sido muy popular en el campo de la academia y la investigación, pero su uso en el campo comercial e industrial es aún incipiente, teniendo en cuenta que el desarrollo en este último sigue siendo dominado por un pensamiento estructurado y orientado a objetos. Algunos autores atribuyen esta situación a dos razones principales. La primera es que la programación funcional difiere bastante de la programación tradicional y por ende implicaría un gran esfuerzo por aprender y desaprender, para quienes han trabajado durante mucho tiempo con los paradigmas imperativo y orientado a objetos. La segunda razón es que el hardware actual implementa el modelo de programación imperativo de forma natural (Hinsen, 2009).

Los lenguajes de programación funcionales mantienen características distintivas del paradigma funcional como son las funciones de orden superior, evaluación perezosa, correspondencia de ecuaciones y patrones, así como diferentes tipos de abstracción de datos. Uno de los lenguajes funcionales modernos más conocidos y utilizados es Haskell, que ha servido de base para la creación de lenguajes de programación funcional paralela como GpH y Eden (Vara, 2008). Adicionalmente varios lenguajes funcionales incluyen constructos paralelos implícitos y primitivas paralelas. Dentro de las múltiples aplicaciones de la programación funcional, se ha advertido sobre la potencialidad que tiene este paradigma para su uso en programación paralela que busca optimizar el uso de las arquitecturas multicore, presentes en el hardware moderno (Hoyos y Puertas, 2015). En cuanto a la comparación de los dos paradigmas, el funcional y el orientado a objetos, es poca la literatura existente. Se evidencio la existencia de una investigación en la que se compararon diferentes lenguajes funcionales en términos del desempeño y la programación (Loid et al., 2003) y otra en la que se comparan los dos paradigmas, pero en términos de métricas de codificación, desarrollo y pruebas (Harrison et al., 1996).

El objetivo de este proyecto fue evaluar el desempeño de aplicaciones desarrolladas mediante programación funcional sobre hardware multicore, para el logro del cual se propuso desarrollar una investigación de tipo exploratoria cuasi experimental, compuesta por cuatro fases: identificación de los diferentes enfoques y aplicaciones de la programación funcional, realización de un estudio comparativo de los lenguajes de programación funcional, caracterización del tipo de aplicación que permite aprovechar el hardware multicore y comparación de un prototipo desarrollado con programación funcional con su equivalente desarrollado en otro paradigma de programación. El documento presenta el resultado de la última fase desarrollada en la investigación. En la primera parte se hace una introducción sobre la programación funcional y su relación con la programación paralela, dado que el paradigma funcional no cuenta con el mismo nivel de conocimiento y popularidad que si tiene su contraparte, el paradigma de orientación a objetos. En un segundo apartado se procede a realizar una comparación de los dos lenguajes escogidos como representantes de los dos paradigmas, Haskell y Java, en términos de los constructos utilizados por cada lenguaje para implementar el paralelismo. A continuación se presenta el contexto y el diseño del experimento propuesto. En el siguiente apartado se da cuenta de los resultados obtenidos a partir de las mediciones realizadas con las dos implementaciones y con diferente número de núcleos.

Teniendo en cuenta la diversidad de las preferencias en el uso de los lenguajes de programación, se tomó como referencia para la selección de los lenguajes a utilizar, el índice TIOBE basado en 25 de los motores de búsqueda mejor ranqueados (TIOBE, 2016) con fecha de actualización de enero de 2016. Aquí pueden verse en la posición 1 a Java, representante de la programación orientada a objetos, y en las posiciones 38 y 44 a Haskell (Marlow, 2013) y Erlang (Armstrong, 2013) respectivamente, como representantes de la programación funcional pura y su paradigma. Así mismo en la posición 27 y 34 pueden observarse F# y Lisp (Hansen y Rischel, 2013) respectivamente, como ejemplos de lenguajes de programación con características funcionales. Los lenguajes Haskell y Java fueron escogidos por considerarse representativos de los paradigmas funcional y orientado a objetos. A continuación se describen los mecanismos incorporados en cada uno de estos lenguajes, que posibilitan la paralelización de algoritmos.

Haskell: En el caso del lenguaje Haskell, se conocen por lo menos dos extensiones para procesamiento paralelo, que utilizan diferentes enfoques para aprovechar una arquitectura multicore, GpH y Eden (Hoyos y Puertas, 2015). Sin embargo este trabajo se enfoca en los mecanismos para la paralelización incluidos en el compilador GHC, considerado el compilador de Haskell líder a nivel mundial (Berthold et al., 2009). Se aclara que el mecanismo escogido es el de una paralelización básica mediante la mónada Eval, ya que en la actualidad, existen otros enfoques que buscan mejorar el desempeño mediante el uso de mecanismos más sofisticados. La mónada Eval define varias operaciones que hacen posible la paralelización en la evaluación de expresiones. Con la operación *rpar* se establece que el argumento debe ser evaluado en forma paralela, y con la operación *rseq* se fuerza una evaluación secuencial de los argumentos. Existe una tercera operación llamada *runEval* encargada de procesar la monada (conjunto de operaciones *rpar* y *rseq*) y retornar el resultado correspondiente.

En principio se podría considerar un particionamiento estático de las operaciones a realizar, con el ánimo de distribuir el trabajo asignado a cada núcleo del procesador. Dicha solución no es la más óptima, especialmente porque se corre el riesgo de que dicha asignación produzca un desbalance que conduzca al detrimento del speedup, o nivel de balance entre los núcleos utilizados. Para solventar estos efectos no deseados, se acude a un particionamiento dinámico, consistente en distribuir pequeñas unidades de trabajo entre los procesadores en tiempo de ejecución, generando un mejor balance (Marlow, 2013). Este comportamiento lo proporciona la operación *parMap*.

Java: El paquete *java.util.concurrent* contiene un conjunto de clases útiles en la programación concurrente, que representan una evolución del soporte multihilo integrado en java desde hace mucho tiempo. Uno de los elementos que contiene este paquete es el framework *fork/join*, que permite el aprovechamiento de una arquitectura multicore, ya que está diseñado para aquellos casos en los que una tarea particular puede ser dividida recursivamente en pequeñas piezas, con el objetivo de utilizar el poder de procesamiento de todos los procesadores disponibles para mejorar el desempeño de la aplicación (Oracle, 2015)**.**

Aunque el paquete contiene un buen número de clases, las que se utilizan para esta implementación son *ForkJoinPool* y *RecursiveAction*. La clase *ForkJoinPool* proporciona el punto de entrada para las tareas entregadas por un cliente, y los mecanismos para la administración o monitoreo de las mismas. La clase abstracta *RecursiveAction* representa una tarea recursiva de tipo *ForkJoinTask* que no ha sido completada. Esta clase establece las reglas para parametrizar acciones no completadas, cuyo valor de retorno (en este caso indicador de completitud) es el valor null (Oracle, 2015).

METODOLOGÍA

En una etapa preliminar fue necesario identificar las características que hacen que un algoritmo se pueda considerar como paralelizable. Una característica fundamental es que el problema pueda ser dividido en partes, cada parte se puedan procesar por separado y finalmente los resultados puedan ser consolidados y entregados como una unidad. Algunos autores han planteado diversas formas en las que se podría realizar la descomposición de la tarea principal, como podría ser la descomposición de los datos, recursiva, exploratoria, especulativa e hibrida (Grama et al., 2003). Cada una de dichas técnicas es aplicada dependiendo del tipo de problema a resolver.

Otro elemento determinante dentro de la arquitectura usada para la etapa experimental, es el concepto de multicore. En este contexto, el término multicore hace referencia a aquellas arquitecturas en las que en un procesador se integran varios cores, usualmente entre dos y diez (Rivoire, 2010). Este tipo de procesador no mejora por sí sólo el desempeño de algoritmos secuenciales, pero si puede mejorar ostensiblemente el desempeño de algoritmos paralelos, siempre y cuando se dé una coordinación apropiada entre los diferentes cores cuando atienden tareas concurrentes (Díaz et al., 2012). Es importante mencionar que este tipo de procesadores, trabaja con un esquema de memoria compartida, teniendo en cuenta que los cores comparten un mismo espacio de memoria. Las mediciones fueron realizadas en dos arquitecturas diferentes. La primera un equipo desktop con la siguiente configuración: estación de trabajo Windows 10 con un procesador FX 8370 de 4,3 GHz (8 cores), 16 MB de caché y 8 GB de RAM. La segunda, una estación de trabajo Windows 10 con un procesador AMD FX 8150 de 3,6 GHz con cuatro procesadores físicos (ocho lógicos), 16 MB de caché y 8 GB de RAM.

Experimento

El algoritmo escogido fue el problema del "Solucionador de Sudokus". El sudoku es un juego muy popular consistente en rellenar un tablero cuadrado de 81 celdas (9x9), dividido en 9 subcuadrados de 9 celdas cada uno (3x3) con los números del 1 al 9, de manera que un número no se repita en la misma fila, columna o

subcuadrado. La solución a estos rompecabezas presenta diferentes niveles de dificultad, que están determinados más por la relevancia y la posición de los dígitos, que por el número de ellos que hacen parte del estado inicial.

Existen diferentes estrategias de solución que se han traducido en algoritmos que permiten la resolución automática de cualquier sudoku de este tamaño. La forma más sencilla es utilizar un mecanismo de fuerza bruta que explore todas las posibilidades hasta encontrar una solución. Sin embargo, se ha demostrado que frente a juegos de alta dificultad, esta solución resulta poco práctica por la cantidad de recursos requeridos, en especial en términos del tiempo de procesamiento. Existe una propuesta que ha demostrado mayor eficacia y eficiencia, utilizando la propagación de restricciones, que reduce rápidamente el espacio de búsqueda, lo que permite probar y descartar un número menor de posibles soluciones (Moraglio et al., 2006). La estrategia algorítmica de solución utilizada e implementada para las pruebas realizadas, es la propuesta por Peter Norving (Norving, 2016), basada en dos ideas básicas: propagación de restricciones (a lo largo de filas, columnas y vecindad) y búsqueda (exploración de caminos diferentes).

Se escoge este problema por ser representativo de un algoritmo que realiza los mismos cálculos sobre cada elemento de un conjunto de datos de entrada, en este caso un estado inicial del juego o rompecabezas, lo que lo hace un buen candidato para la paralelización. Un estado inicial, está compuesto por una cadena de 81 caracteres, que representan los valores para un tablero de tamaño 9x9. Los diez caracteres válidos se limitan a los dígitos (1..9) y el punto que representa un espacio libre al que posteriormente le será asignado un digito. A manera de ejemplo un dato de entrada sería representado por la cadena de 81 caracteres “4.....8.5.3.....7.....2.....6.....8.4.....1.....6.3.7.5..2.....1.4.....” y equivaldría al estado inicial del sudoku mostrado en la figura 1. En este caso se habla de procesamiento paralelo, y un esquema de memoria compartida, teniendo en cuenta que los 8 núcleos comparten un mismo espacio de memoria.

4 . . . 3	. . . . . . 7 . .	8 . 5 . . . . . .
. 2	. . . . 8 . . 1 .	. 6 . 4 . . . . .
. . . 5 . . 1 . 4	6 . 3 2	. 7 . . . .

Fig. 1: Ejemplo del estado inicial del sudoku

El grado de paralelización está determinado por el tamaño del conjunto de entrada, ya que a cada entrada individual de dicho conjunto, le es aplicado el algoritmo solucionador de sudokus, que para este caso es tratado como una caja negra. En el caso de las pruebas realizadas, se utilizó un conjunto de entrada de 1000 individuos, que da origen a 1000 tareas diferentes, una por cada individuo, por lo que se considera una descomposición del conjunto de datos, lo que lo hace un caso representativo de uno de los principales tipos de descomposición utilizados en la programación paralela.

Implementación

Se parte de una solución ya implementada en el lenguaje funcional Haskell. A partir de esta implementación, se construyó su equivalente en el paradigma de programación orientada a objetos utilizando el lenguaje de programación Java.

Implementación en Haskell

La implementación utilizada es presentada en el libro "Parallel and Concurrent Programming in Haskell - Techniques for multicore and multithreaded programming" (Marlow, 2013). Esta solución utiliza la estrategia algorítmica de Peter Norving. En la figura 2(a) se muestra la codificación de la función *solve*, que recibe como valor de entrada una cadena representativa del estado inicial del problema (sudoku) y retorna la solución del mismo en un formato representativo del tablero de 81 celdas con sus respectivos valores.

solve :: String -> Maybe Grid *solve str = do* *grd <- parsegrid str* *search grd*	*solutions = runEval (parMap solve puzzles)*	*parMap :: (a -> b) -> [a] -> Eval [b]* *parMap f [] = return []* *parMap f (a:as) = do* *b <- rpar (f a)* *bs <- parMap f as* *return (b:bs)*
(a)	(b)	(c)

Fig. 2: Fragmentos de código de la implementación en Haskell

La función mostrada en la figura 2(a) calcula la solución para un caso en particular, pero teniendo en cuenta que la aplicación toma como entrada un conjunto de problemas, la función debe ser aplicada en forma recursiva a cada una de las entradas, cada una de las cuales es representada por una cadena. En la figura 2(b) se muestra la instrucción correspondiente. En este caso se está ejecutando la operación *runEval* encargada de procesar la monada Eval, constituída por la función *parMap* y sus parámetros: la función *solve* y el identificador *puzzle* que contiene el conjunto de casos de entrada. El resultado correspondiente es almacenado en *solutions*. La función *parMap* se define como se muestra en la figura 2(c). La función *parMap*, ejecuta la operación *rpar* sobre la función *f* y el primer elemento de la lista *a*, y se llama de forma recursiva para la función *f* y los elementos restantes de la lista inicial.

Implementación en Java

Se genera una aplicación que intenta reproducir, en la medida de lo posible, el mismo esquema de solución planteada en la implementación con Haskell. En la figura 3(a) se muestra la codificación del método *solve*, equivalente a la función *solve*, el cual hace parte de la clase *Sudoku.java*

De forma similar a la implementación en Haskell, el método mostrado en la figura 3(a) calcula la solución para un caso en particular, y por ende el método debe ser aplicado en forma recursiva a cada una de las entradas, mediante la llamada al método *invoke* que recibe como parámetro el objeto *task*, representativo de una tarea recursiva, que se traduce en un conjunto de tareas individuales generadas para cada uno de las entradas contenidas en el objeto juegos (arreglo de String), tal como se muestra en la figura 3(b) . En este caso, es el método *compute* de la clase *Task*, el encargado de crear las tareas individuales para cada una de las entradas, tal como se muestra en la la figura 3(c)

public Grid solve(String cadena){ *Grid grid = parseGrid(cadena);* *return search(grid);* *}*	*Task task = new Task(juegos);* *pool.invoke(task);*	*protected void compute() {* *Grid all = (new Sudoku()).solve(juego);* *if(enlistados<juegos.length){* *Task task2 = new Task(juegos);* *invokeAll(task2);* *}* *}*
(a)	(b)	(c)

Fig. 3: Fragmentos de código de la implementación en Java

En esta investigación se comparó el desempeño del procesamiento paralelo de un mismo algoritmo, implementado en los lenguajes de programación Haskell y Java, representativos de los paradigmas funcional y orientado a objetos, respectivamente. El algoritmo utilizado fue el solucionador de sudokus y las variables medidas fueron el tiempo de ejecución, el speedup y la eficiencia paralela. La implementación del paralelismo básico en Haskell se basó en la mónada Eval, mientras que en Java se construyó a partir de la clase ForkJoinPool. En ambos casos la solución utiliza una función recursiva encargada de generar una tarea por cada caso del conjunto de datos de entrada.

RESULTADOS

Las mediciones que se presentan en este apartado, corresponden a la ejecución de los algoritmos secuenciales en un conjunto de 1.000 problemas diferentes y los algoritmos paralelizados con el mismo conjunto pero utilizando un número diferente de núcleos (de 1 a 8). Es necesario anotar que se midieron los tiempos para 10 ejecuciones por cada una de las diferentes configuraciones, y se tomó el tiempo promedio de las mismas. En la figura 4 se presentan los tiempos de ejecución alcanzados en las pruebas, para cada configuración (número de núcleos), y para las dos arquitectura utilizadas.

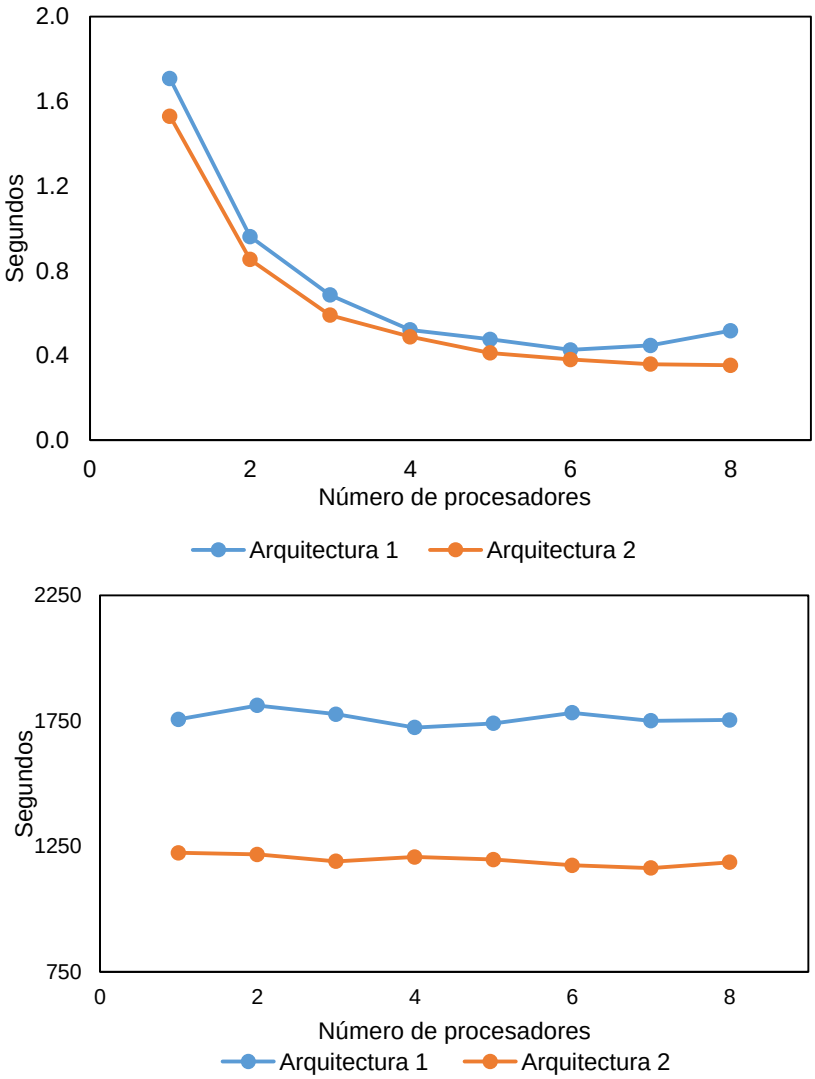

Fig. 4: Tiempos de ejecución de las implementaciones Haskell y Java, en dos arquitecturas diferentes

En la figura 4 se puede apreciar, que para el caso de Haskell el tiempo de ejecución disminuye, a medida que aumenta el número de núcleos utilizados para la ejecución del algoritmo en paralelo. Una situación similar se observa con Java con la disminución en el tiempo de ejecución a medida que aumenta el número de núcleos, aunque de una de forma más irregular. Sin embargo se debe mencionar que la diferencia en los tiempos entre Haskell y Java resulta demasiado grande. Con el fin de medir el desempeño de las dos implementaciones en el paralelismo multicore, se utilizan los indicadores de speedup y eficiencia paralela, definidos en (Phan y Hansen, 2014).

Speedup: Relación entre el tiempo de ejecución de un programa en paralelo y su equivalente en una implementación secuencial, y está dada por:

$$S_N = \frac{T_1}{T_N} \quad (1)$$

Donde N es el número de procesadores, T_1 es el tiempo de ejecución del algoritmo secuencial, y T_N es el tiempo de ejecución del algoritmo paralelo utilizando N procesadores.

Eficiencia paralela: indica que tan eficientemente es utilizado el poder de procesamiento de N procesadores, y es definida como la relación entre el speedup y el número de procesadores utilizado para la ejecución del algoritmo paralelo, dada por:

$$E_N = \frac{S_N}{N} \quad (2)$$

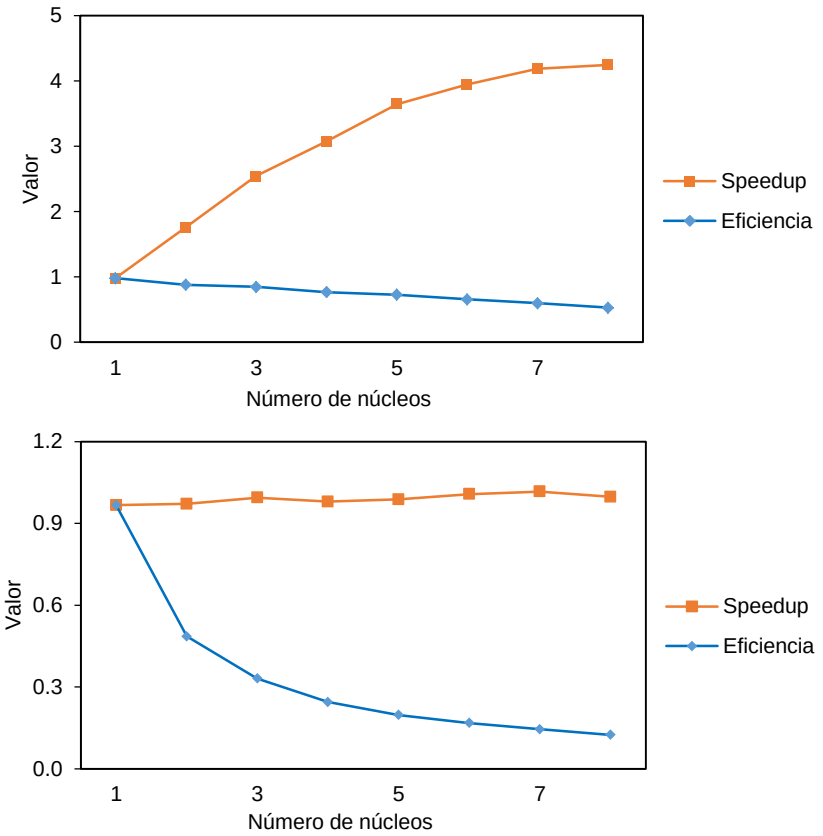

Fig. 5: Valores para speedup y eficiencia paralela de las implemen-taciones Haskell y Java, en la arquitectura 1.

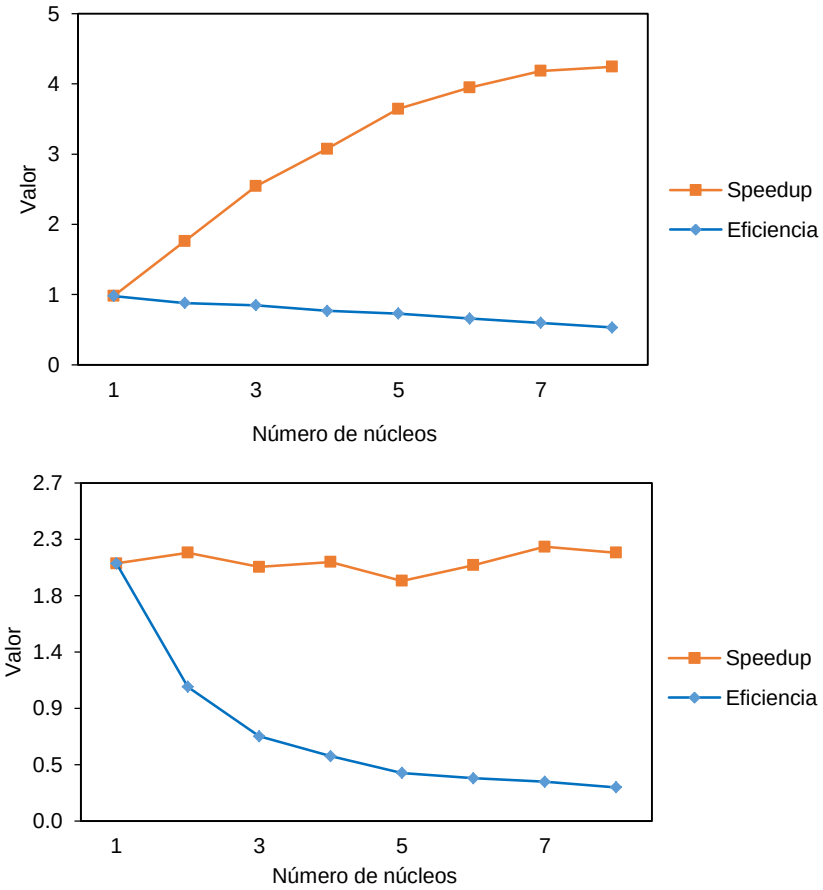

Fig. 6: Valores para speedup y eficiencia paralela de las implementa-ciones Haskell y Java, en la arquitectura 2.

En la figura 5 se presentan los valores obtenidos para los indicadores speedup y eficiencia paralela, para cada configuración (número de núcleos) en la primera arquitectura. En la figura 6 se presenta el resultado de las mismas mediciones para la segunda arquitectura. En las figuras 5 y 6 se puede apreciar, que para el caso de Haskell el indicador de speedup aumenta a medida que aumenta el número de núcleos utilizados para paralelizar el algoritmo, mientras que el indicador de eficiencia disminuye, pero en menor proporción. En el caso de Java, se muestra un leve crecimiento del indicador de speedup a medida que aumenta el número de núcleos, mientras que el indicador de eficiencia disminuye de forma más acelerada a como lo hace la ejecución en Haskell, indicando una correspondencia inversamente proporcional entre los dos indicadores.

DISCUSIÓN

En cuanto al tiempo de ejecución, se observa una diferencia muy grande entre los tiempos alcanzados con la implementación en Haskell, frente a su contraparte implementada en Java. Una posible explicación se podría encontrar en el tamaño de las tareas, teniendo en cuenta que para ambos casos el proceso es dividido en 1000 tareas, una por cada problema del conjunto de problemas representado en la entrada. Teniendo en cuenta lo anterior la diferencia radicaría en el tamaño de la tarea (en términos de tiempo de procesamiento), que como ya se mencionó presenta una diferencia demasiado grande, entre las dos implementaciones.

En vista de la diferencia tan elevada en el tiempo de ejecución entre las dos implementaciones, se examinó la implementación en java para tratar de detectar la causa de esta situación. Se pudo observar, que un solo método, el primero en ser invocado (*parseGrid*) consume en promedio el 48% del tiempo total de ejecución. Este método es el encargado de configurar el estado inicial de cada juego, ubicando en cada posición de la estructura que representa el sudoku, los dígitos que hacen parte de la entrada (cadena de texto). Lo anterior sólo puede explicarse por la cantidad de objetos creados y llamadas a métodos generados por el algoritmo, dada la naturaleza recursiva del método *parseGrid*. Sin embargo se pudo establecer que ésta implementación no presenta dependencias de datos que fuercen la secuencialidad del algoritmo y por tanto no limita el paralelismo del mismo. Tampoco se encontraron errores en la realización de los experimentos.

Ya en una investigación anterior en la que se compararon los dos paradigmas, los autores concluyeron que sus resultados sugerían que los lenguajes funcionales fomentaban o estimulaban un mayor grado de reuso (llamado a funciones) que los lenguajes orientados a objetos, comportamiento atribuido en parte al uso de funciones especializadas en el procesamiento de listas, propias de los lenguajes funcionales (Harrison et al., 1996). Al examinar el comportamiento aislado de cada una de las implementaciones frente al número de núcleos utilizados en el procesamiento del algoritmo paralelo, se encuentran dos tendencias similares. En el caso de Haskell, el tiempo de ejecución disminuye a medida que aumenta el número de núcleos. Algo similar sucede con la implementación en Java, donde se observa una disminución del tiempo de ejecución al aumentar el número de núcleos.

Para el análisis de desempeño del procesamiento paralelo es necesario tener en cuenta que el procesamiento en paralelo debe tratar con cargas adicionales propias del paralelismo como son la comunicación y la sincronización entre las tareas. En teoría, si se utilizan N procesadores se podrían obtener tiempos de ejecución N veces más rápidos, pero esto normalmente no es posible (Phan y Hansen, 2014). En lo que tiene que ver con el indicador de speedup, se aprecia un comportamiento coherente con el observado en el tiempo de ejecución, es decir que mientras en Haskell, el speedup aumenta de forma notoria a medida que aumenta el número de núcleos, en Java se presenta un aumento leve del speedup al aumentar el número de núcleos, indicando que el mecanismo utilizado por Haskell genera una distribución más óptima del trabajo entre los diferentes núcleos involucrados en el procesamiento.

Adicionalmente, en el caso de Java es necesario observar que el indicador de speedup no aumenta tan significativamente como podría esperarse en respuesta al incremento en el número de núcleos, debido a que la diferencia en el tiempo de ejecución para N=1 y N=8 es muy pequeña, ya que sólo representa un 3% respecto al tiempo total, a diferencia de Haskell, en el que dicha diferencia, representa un 77% del tiempo total. En cuanto al indicador de eficiencia paralela, se observa un comportamiento similar en las dos implementaciones, ya que en ambos casos el indicador disminuye frente al aumento en el número de núcleos. Para el caso particular del algoritmo escogido, la eficiencia en el uso de los diferentes núcleos utilizados es mayor cuando se utiliza un número menor de ellos, indicando que la carga de trabajo no es suficientemente grande para sacar un mayor provecho de la arquitectura multicore.

CONCLUSIONES

Se encontró una diferencia significativa en el tiempo de ejecución, que sólo pudo ser explicada por el tamaño de la tarea (en términos de tiempo de procesamiento), que en el caso de Java es muy superior ya que está determinada por la cantidad de objetos creados y llamadas a métodos generados por el algoritmo,

dada la naturaleza recursiva de la mayoría de los métodos utilizados. A diferencia de Haskell que hace un uso intensivo de funciones especializadas en el procesamiento de listas, propio de los lenguajes funcionales.

Aunque se pudo confirmar la tesis de varios autores, en el sentido que el paradigma funcional posee grandes potencialidades para la programación paralela, este argumento no puede ser utilizado para explicar la diferencia en tiempos de ejecución encontrada entre las dos implementaciones. En el análisis del experimento realizado, se pudo observar que en la implementación Java el tamaño de las tareas (en términos de tiempo de procesamiento) era muy superior al de la implementación Haskell debido a los elementos secuenciales del algoritmo de solución de sudokus. Por lo anterior, no es posible atribuir la amplia diferencia en los tiempos de ejecución, a los mecanismos de paralelización proporcionados por cada uno de los lenguajes utilizados, ya que en este caso, lo que se paralelizó fue la ejecución del algoritmo solucionador para 1000 estados iniciales diferentes.

En la comparación de las dos implementaciones, se pudo observar que los valores de speedup y eficiencia paralela presentan comportamientos similares. El speedup en ambos casos aumenta a medida que aumenta el número de núcleos, pero este incremento es mucho más notorio en Haskell que en Java. La eficiencia paralela disminuye en ambos casos al aumentar el número de núcleos, pero contrario a lo anterior, el decrecimiento en Haskell es leve frente a un decrecimiento más pronunciado en Java.

En las pruebas realizadas se pudo evidenciar que la eficiencia con la que es utilizado el poder de procesamiento de la arquitectura, disminuye a medida que se involucra un mayor número de núcleos, lo que es mucho más notorio en Java que en Haskell. Teniendo en cuenta lo anterior, se puede concluir que, aun utilizando paralelización básica, Haskell hace una utilización más óptima de la arquitectura multicore, y que las grandes diferencias observadas en el tiempo de ejecución, obedecen más a temas de implementación del algoritmo como tal, que a los mecanismos utilizados para implementar la paralelización.

REFERENCIAS

Armstrong, J., Programming Erlang - Software for a Concurrent World, 2ª Ed., The Pragmatic Programmers, Dallas, Estados Unidos de América (2013)

Berthold, J., Marlow, S., Hammond, K. y Al Zain, A. D., Comparing and Optimising Parallel Haskell Implementations for Multicore Machines, doi: 10.1109/ICPPW.2009.10, International Conference on Parallel Processing Workshops, 386-393 (2009)

Castro, S., Medina, B. y Camargo, L., Supervisión y control industrial a través de telefónos inteligentes usando un computador de placa única Raspberry Pi, doi: 10.4067/S0718-07642016000200015, Información Tecnológica, 27(2), 121-130 (2016)

Díaz, J., Muñoz-Caro, C. y Niño, A., A survey of parallel programming models and tools in the multi and many-core era, IEEE Transactions on parallel and distributed systems, 23(8), 1369-1386 (2012)

Gónzalez, F., Programacion funcional: conceptos, Ingeniería E Investigación, 40(1), 65–71 (1998)

Grama, A., Gupta, A., Karipis, G. y Kumar, V., Introduction to Parallel Computing, 2ª Ed., pp. 656, Addison-Wesley (2003)

Harrison, R., Samaraweera, L.G., Dobie, M.R., y Lewis, P.H., Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs, doi: 10.1049/sej.1996.0030, Software Engineering Journal, 11, 247–254 (1996)

Hansen, M. y Rischel, H., Functional programming using F#, Cambridge University Press, Cambridge, Inglaterra (2013)

Hinsen, K., The promises of functional programming, Computing in Science & Engineering, 86-90 (2009)

Hoyos, J. y Puertas, A., La programación funcional y las arquitecturas multicore : Estado del Arte, Ingenio Magno, 6(2), 124–136 (2015)

Loidl, H., Rubio, F., Scaife, N., y Hammond, K., Comparing Parallel Functional Languages: Programming and Performance, Higher-Order and Symbolic Computation, 16, 203–251 (2003)

Marlow, S., Parallel and concurrent programming in Haskell, 1ª Ed., pp. 304, O'Reilly Media, Inc., Sebastopol (2013)

Moraglio, A., Togelius, J., y Lucas, S., Product Geometric Crossover for the Sudoku Puzzle, doi: 10.1109/CEC.2006.1688347, IEEE International Conference on Evolutionary Computation, 470–476 (2006)

Norving, P., Solving Every Sudoku Puzzle, (en la web: http://norvig.com/sudoku.html, acceso: 14 de septiembre de 2016) (2016)

Oracle, The Java Tutorials - Fork/Join, (en la web: https://goo.gl/rDyf3a, acceso: 26 de octubre de 2016) (2015)

Phan, A.D. y Hansen, M. R., An approach to multicore parallelism using functional programming: A case study based on Presburger Arithmetic, doi: 10.1016/j.jlamp.2014.07.002, Journal of Logical and Algebraic Methods in Programming, 1, 1–17 (2014)

Rivoire, S., A breadth-first course in multicore and manycore programming, doi: 10.1145/1734263.1734339, Proceedings of the 41st ACM Technical Symposium on Computer Science Education - SIGCSE'10, 214 (2010)

TIOBE © 2017 software BV. TIOBE Index for January 2016 (en la web: https://goo.gl/qVfPjn, acceso: 5 de febrero de 2016) (2016)

Vara, A., Formalizando el proceso de depuración en programación funcional paralela y perezosa, Tesis Doctoral, Universidad Complutense de Madrid, España (2008)