Desarrollo de aplicaciones paralelas en Erlang/OTP utilizando múltiples *planificadores*

Juan Ernesto Pisani[1], Pablo Cristian Tissera[1], A. Marcela Printista[1,2]

[1] Departamento de Informática, Laboratorio de Investigación y Desarrollo en Inteligencia Computacional. UNSL.
[2] Conicet - CCT San Luis.
{juanpisani, ptissera, marprinti}@gmail.com

Resumen.
Este trabajo se enfoca en el desarrollo de aplicaciones distribuidas y paralelas mediante el uso de un lenguaje orientado a la concurrencia, denominado Erlang. El objetivo es explorar las capacidades y limitaciones de este lenguaje de programación cuando se desea implementar aplicaciones sobre arquitecturas multicore. Como caso de uso, se describe una secuencia de implementaciones de un modelo de Autómata Celular, el cual presenta ciertas características específicas que lo hacen un modelo atractivo para aplicación de técnicas de paralelización.

Palabras claves: leguaje Erlang, máquina virtual, soporte multicore, planificadores.

1 Introducción

La primera implementación del lenguaje Erlang [1,7], en la década del 70, consistió de un intérprete implementado en Prolog [3,4] para el álgebra desarrollada por la Empresa de Telefonía Ericsson. Dos décadas mas tarde, con la incorporación del entorno de ejecución denominado, *BEAM Virtual Machine,* logra su aceptación en la comunidad de sistemas distribuidos.

Erlang es considerado un **lenguaje de programación orientado a la concurrencia**. Además, Erlang se ajusta en muchos aspectos al **paradigma de lenguaje funcional**. Al igual que los demás lenguajes funcionales enfatiza la aplicación de funciones en contraste con los cambios de estados típicos del paradigma imperativo. Erlang resalta las principales ventajas de todo lenguaje funcional como por ejemplo la ausencia de efectos colaterales, la mayor facilidad para la depuración de programas y la mayor facilidad para la ejecución concurrente, entre otras.

Sumado a estas características, el lenguaje también presenta propiedades de un paradigma de orientación a objetos, como es el caso del polimorfismo el cual admite sobrecargar una función determinada en base a la cantidad de parámetros que la misma reciba en tiempo de ejecución. Actualmente existe una discusión abierta concerniente a todas las características que el lenguaje presenta y que lo apartan de la

definición pura de lenguaje funcional. Por ello a menudo se lo cataloga como lenguaje *híbrido*, dado que un subconjunto del mismo es estrictamente funcional pero incluye construcciones del tipo imperativo e incluso rasgos de la orientación a objetos, aunque no respeta completamente ninguno de estos paradigmas.

Es relevante mencionar que como lenguaje orientado a la concurrencia Erlang proporciona facilidades para la programación distribuida y paralela. El lenguaje integra un conjunto sencillo pero completo de operaciones que permiten la creación y manejo de procesos concurrentes dentro de su máquina virtual. Erlang utiliza procesos ligeros cuyos requisitos de memoria pueden variar de forma dinámica. Los procesos no tienen memoria compartida y se comunican por paso de mensajes asíncronos. En un entorno distribuido, la *Máquina Virtual Erlang* recibe el nombre de Nodo Erlang, pudiendo llegar a constituirse una red de nodos Erlang, cuyos procesos se comunican mediante el paso de mensajes. La administración del hardware subyacente se mantiene en forma transparente a la resolución del problema, a diferencia de lenguajes de programación como C que utilizan librerías auxiliares como PVM o MPICH para lograr el mismo propósito.

En 2010 Erlang agregó soporte SMP (acrónimo del inglés *Symmetric Multi-Processing*) y más recientemente soporte para plataformas multicore, características que serán ampliadas en la sección 2 del presente trabajo.

Este trabajo tiene como objetivo presentar el desarrollo de una aplicación distribuida utilizando el lenguaje Erlang y analizar la performance de la misma sobre computadoras que disponen de múltiples cores para su ejecución. En la sección 3 se define nuestro caso de estudio que consiste de la Simulación de un autómata celular, la sección 4 desarrolla las implementaciones realizadas y la sección 5 realiza un análisis comparativo de las mismas. Finalmente la sección 6 presenta las conclusiones de esta experiencia y el trabajo de investigación futuro.

2 Modelo de Programación Concurrente

En Erlang toda actividad concurrente es encapsulada dentro de una entidad computacional denominada proceso. Los procesos en Erlang son entidades independientes, no cuentan con memoria compartida ya que cada proceso tiene su propia memoria y los cambios realizados en la misma son de carácter local. Los distintos procesos interactúan entre ellos vía pasaje de mensajes asincrónicos.

Cabe aclarar que en este contexto haremos uso del término *proceso* para referirnos a entidades independientes dentro del entorno de ejecución de Erlang. Dichos procesos son administrados directamente por el entorno de ejecución y no por el sistema operativo. Es importante también diferenciar el concepto de procesos Erlang, los cuales representan entidades independientes entre sí, con un thread de ejecución los cuales comparten estructuras de datos y recursos comunes. Generalmente los programas en Erlang se componen de decenas, miles o incluso cientos de miles de pequeños procesos, todos ellos funcionando de forma independiente e interactuando entre sí mediante el envío de mensajes para coordinar las distintas actividades a realizar y poder llegar a la solución de un problema determinado. Los programas organizados de esta manera escalan fácilmente ya que

ante una demanda excesiva de trabajo es posible crear más procesos de modo tal que la demanda sea satisfecha sin pérdida de rendimiento.

2.1 Programación Multicore

Como mencionamos anteriormente, en Erlang toda concurrencia es explicita y forma parte del lenguaje. El lenguaje brinda un conjunto reducido pero robusto de primitivas para poder crear procesos y facilitar las comunicaciones entre ellos. Este conjunto sólo incluye tres operaciones que permitirán en definitiva crear procesos, enviar mensajes a procesos previamente creados y recibir mensajes respectivamente.

Erlang provee soporte para una implementación Multicore. Esto se logra con una operación combinada entre la Máquina Virtual Erlang la cual proporciona el marco en el cual las aplicaciones serán ejecutadas y el soporte SMP provisto por el lenguaje. Las aplicaciones Erlang se compilan y luego son ejecutadas por la Máquina Virtual (es decir su entorno de ejecución) independientemente del hardware subyacente que se utilice.

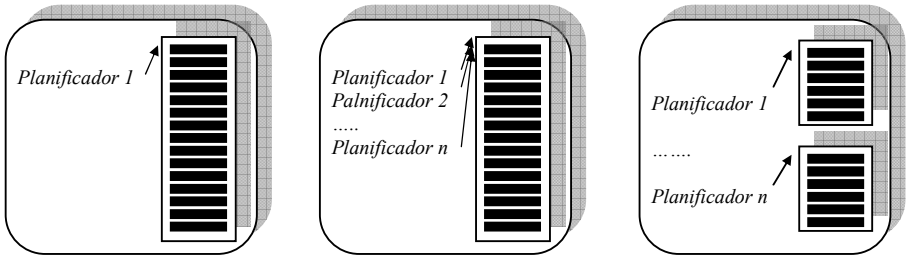

Fig. 1. Máquina Virtual Erlang: (izq) sin Soporte SMP (centro) varios panificadores para coordinar el trabajo desde una única cola y (der) varios planificadores organizan el trabajo desde n colas de trabajos.

Inicialmente, el soporte SMP se encuentra deshabilitado, por lo que el entorno de ejecución de Erlang inicia solo un planificador el cual se ejecuta sobre el thread principal de ejecución de la Máquina Virtual. La función de este planificador es coordinar directamente los distintos procesos y trabajos de entrada/salida, a medida que los mismos se encuentren disponibles, desde una única cola de ejecución como puede verse en la Fig. 1 (izq).

En el caso de hacer uso del soporte SMP, se podrán iniciar varios planificadores, los cuales se ejecutarán sobre threads a nivel de aplicación totalmente independientes, situación que se puede observar en la Fig. 1 (centro). Se debe tener en cuenta que los distintos planificadores, en este caso, estarán bajo una condición de competencia permanente requiriéndose mecanismos de sincronización para el acceso a la cola de ejecución. Esta implementación en particular fue utilizada para las primeras versiones de la plataforma que brindaron soporte SMP para arquitecturas de hardware multicore.

Para solucionar el problema de sincronización se introdujeron múltiples colas de ejecución, característica útil en caso de disponer de varios recursos de computación (cores). Como se puede observar en la Fig. 1 (der) los procesos son asignados a distintos planificadores y cada uno de ellos podrá planificar su ejecución de forma independiente.

Finalmente se introdujo lógica de migración de procesos entre planificadores a fin de evitar que la proliferación de procesos sobrecargue alguna cola de ejecución en tiempo de ejecución. La lógica de migración existente debe ser eficiente y a la vez razonablemente justa para garantizar un balance de carga de los distintos procesos entre los planificadores instanciados. Esta configuración se muestra en la Fig. 2.

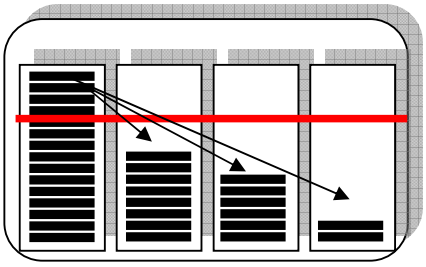

Fig. 2. Máquina Virtual Erlang con soporte SMP, varios planificadores, varias colas de ejecución y con capacidades de migración.

La lógica de migración implementada resulta transparente para el programador.

3 Caso de Estudio: Simulación del Juego de la Vida

El autómata celular (AC) [4] que resuelve *El Juego de la Vida* fue diseñado por el matemático británico John Horton Conway en 1970. Su nombre se debe a la estrecha analogía que tiene el juego con el auge, la caída y alternancias de una sociedad de organismos vivos. Desde su publicación [5], ha atraído mucho interés debido a la gran variabilidad de la evolución de los patrones.

Para la simulación del Juego de la Vida, se utiliza un modelo de autómata celular, el cual consiste de una cuadrícula dividida en celdas contiguas que representará el tablero de juego. Para la evolución de la simulación, cada celda chequea el valor de su celda y el de sus ocho celdas vecinas (vecindad de Von Neuman) y entonces decide acerca de la vida o muerte de un organismo en la misma. Cada celda del tablero contiene a lo sumo un organismo y los distintos organismos residentes en las distintas celdas son los participantes del juego. Su evolución está determinada por el estado inicial del juego y no requiere de ninguna entrada de datos posterior.

La configuración inicial del juego consiste en distribuir una determinada cantidad de organismos sobre las distintas celdas disponible y a partir de allí observar como las "leyes genéticas" (definidas por Conway) se aplican para los nacimientos,

muertes y supervivencias de los distintos organismos o población inicial que participa en el juego:

1. Supervivencia: cada organismo rodeado por dos o tres organismos en la presente generación, sobrevive para la próxima generación.
2. Nacimiento: toda celda sin un organismo y rodeada por tres organismos en la presente generación, incluirá un organismo en la próxima generación.
3. Extinción (Disgregación): cada organismo rodeado por sólo un organismo o ninguno en la presente generación, morirá en la próxima generación.
4. Extinción (Hacinamiento): cada organismo rodeado por cuatro o más organismos en la presente generación, morirá para la próxima generación.

En un AC, las reglas de evolución se aplican simultáneamente a todas las celdas del autómata, lo que constituye una generación. En este caso de estudio, la aplicación de las "leyes genéticas" a todas las celda del tablero, constituye un "movimiento" en la "historia de vida" del juego. Por ende la primera generación se obtiene luego de aplicar las reglas de evolución a la configuración inicial del juego. A partir de allí podremos repetir el procedimiento para obtener sucesivas generaciones. Durante el transcurso del juego se puede observar cómo la población inicial evoluciona constantemente debido a cambios que resultan inesperados, aunque esto puede no ocurrir hasta después de un gran número de generaciones. La Fig. 3. muestra ejemplos de patrones particulares en los cuales se pueden observar distintos comportamientos generalizados para el juego de la vida.

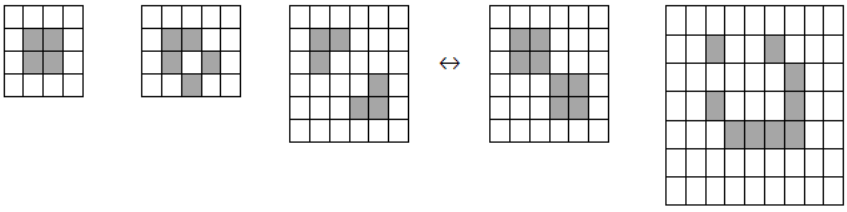

Fig. 3. Patrones del Juego de la Vida: estáticos (1ero. y 2do.), recurrente (3ro. y 4to.) y un patrón infinito (5to.)

4 Implementaciones

Para el desarrollo de la aplicación concurrente que describiremos en este trabajo, se utilizó el framework llamado OTP (acrónimo del inglés *Open Telecom Platform*) [6,8]. Esta herramienta de trabajo organiza los programas en una jerarquía de procesos conocida como *árbol de supervisión*. Dicha jerarquía estructura los distintos procesos que conforman un programa en procesos **Subordinados o Workers,** aquellos procesos que se encargan de procesar la lógica interna de un programa determinado y los **Supervisores** que controlan el comportamiento de los workers, los inician o finalizan. Los procesos supervisores y workers cuentan con una estructura

similar y la única diferencia existente entre ellos es la posibilidad, por parte de los primeros, de poder reiniciar a aquellos procesos que supervisan.

Para la experimentación se consideró un autómata celular inicializado aleatoriamente y un periodo de evolución igual a 100 generaciones. El espacio celular del autómata está representado por una matriz bidimensional NxN donde N=256. Los test correspondientes fueron ejecutados sobre un multiprocesador de 32 cores, con memoria de 64GB (16x4GB), 1333MHz y un disco de 500GB. 4x AMD Opteron 6128, 2.0GHz, 8C, 4M L2/12M L3, 1333 Mhz. El sistema operativo utilizado es Linux Centos 6 con soporte de 64 bits. Los resultados se obtuvieron variando el número de planificadores o threads de la Máquina Virtual Erlang y asignando cada uno de ellos a un core distinto del servidor multicore disponible.

Primer Implementación (v.1)

En esta implementación *cada celda fue representada por un proceso independiente*, de modo tal que el conjunto de procesos representa el espacio celular del autómata. El tablero de juego puede visualizarse como un conjunto de procesos interconectados entre sí dentro del entorno de ejecución de Erlang. Esta implementación requirió de un proceso adicional para sincronizar la evolución general del autómata, quién también hizo las veces de un servidor centralizado de búsqueda (CSS del acrónimo *Central Search Server*) encargado de identificar a la vecindad de un proceso.

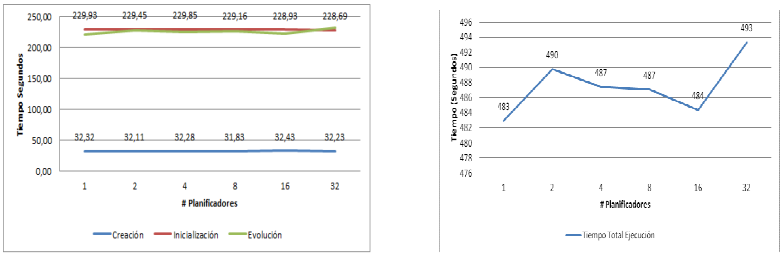

Fig. 4. v.1: (izq) tiempo total de creación de procesos, de inicialización de procesos y total de evolución; (der) tiempo total de ejecución de la simulación.

Como puede observarse en la Fig. 4, los resultados de la v.1 distan de ser alentadores en el contexto de computación distribuida o paralela, dado que los tiempos de ejecución de la aplicación no varían al aumentar el número de planificadores que utilice el entorno de ejecución de Erlang.

Segunda Implementación (v.2)

El componente CSS utilizado en la v.1 utiliza como principal estructura de datos para almacenar toda la información relacionada a los procesos, una lista secuencial ordenada. La alta demanda de trabajo requerido por todos los procesos creados hace que esta lista secuencial ordenada sea una mala opción como estructura de datos inicial dado que limita severamente el rendimiento del sistema completo.

Con el objetivo de acelerar el proceso de inicialización se reemplazó la lista secuencial por una estructura denominada *ETS* (acrónimo del inglés, *Erlang Term Storage*) la cual forma parte del lenguaje y cuenta con soporte para operaciones de lectura/escritura concurrente. Conceptualmente una tabla ETS es una estructura de datos capaz de almacenar cualquier término *clave – valor* en Erlang, en una analogía directa con arreglos asociativos encontrados en muchos otros lenguajes de programación. Erlang provee un conjunto amplio de operaciones que pueden realizarse con tablas ETS, como por ejemplo consultas que involucren *pattern-matching*. El tipo determinado de una tabla ETS queda establecido al momento de su creación [1].

Para almacenar la información relacionada a los procesos del autómata, la v.2 utiliza una tabla ETS del tipo *ordered_set*. Entre la información que se almacena en esta estructura se encuentra la posición determinada (X, Y) de cada proceso, la cual se utiliza como clave principal en la tabla ETS, y el Identificador de Proceso o PId el cual será compartido por los distintos procesos que lo requieran.

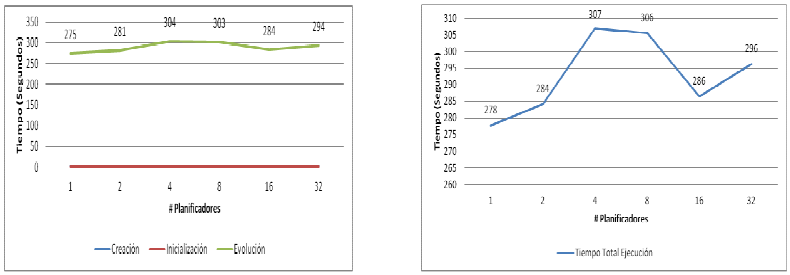

Fig. 5. v.2: (izq.) tiempo total de creación de procesos, de inicialización de procesos y de evolución; (der.) tiempo total de ejecución de la simulación.

La Fig. 5 (izq.) refleja la reducción en el tiempo total requerido para completar la fase de inicialización producto del uso de una estructura más eficiente. Sin embargo, se puede observar que el tiempo de simulación menor es logrado con un único planificador. Este pobre desempeño, se debe a probables cuellos de botella que permanecen y que serán abordados en próximas versiones.

Tercera Implementación (v.3)

La coordinación existente entre el proceso administrador y los demás procesos workers produce una merma en el rendimiento de la aplicación ya que todo el procesamiento se concentra sobre un único punto de control. Durante el tiempo que le insume al proceso administrador distribuir todo el trabajo entre los procesos existentes, el resto de los planificadores permanecen ociosos. En este escenario, solo un único planificador dispone de exclusividad para ejecutar su código fuente mientras los demás planificadores permanecen ociosos hasta que la ejecución del proceso administrador termina.

La v.3 replantea la estructura jerárquica utilizada y propone utilizar tantos procesos administradores como planificadores se estén utilizando en el entorno de ejecución de Erlang. De este modo se espera adaptar la ejecución del programa de acuerdo a la disposición de hardware con la que se cuenta.

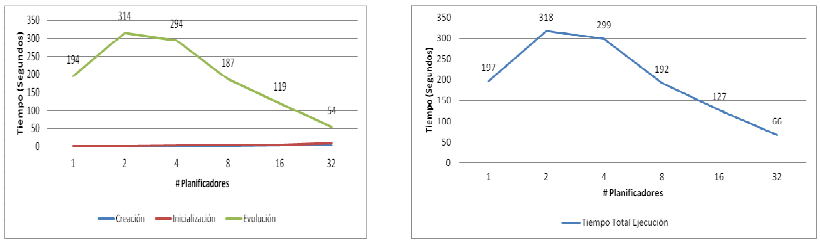

Fig. 6. v.3: (izq.) tiempo total de creación de procesos, de inicialización de procesos y total de evolución; (der.) tiempo total de ejecución de la simulación.

Como se muestra en la Fig. 6, la nueva estructura de la aplicación con múltiples procesos administradores hace que el rendimiento de la aplicación mejore a medida que se disponen de más planificadores durante el tiempo de ejecución.

Cuarta Implementación (v.4)

Esta implementación tiene el objetivo de maximizar la performance de la aplicación evitando penalidades relacionadas a una excesiva administración de procesos. La nueva estructura de programa consta de procesos workers capaces de contener más de una celda del autómata celular en su estado interno, quedando el espacio celular distribuido equitativamente entre los procesos creados inicialmente. Toda esta información será almacenada en una tabla ETS a fin de que las distintas operaciones que deban realizarse sobre ella sean eficientes.

El código fuente del programa se simplificó dado que en esta implementación no se utiliza un proceso como CSS. Tampoco es requerida una etapa de inicialización de procesos ya que las distintas tablas ETS creadas utilizan como clave de búsqueda la posición (X, Y) de una celda dada del autómata.

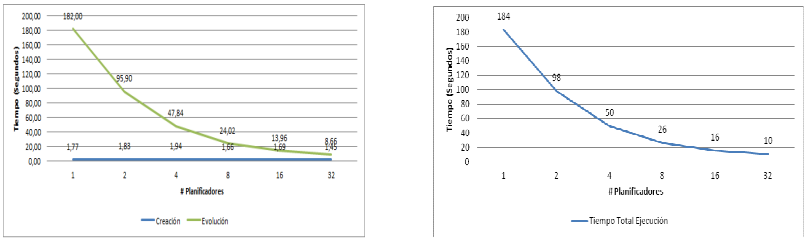

Fig. 7. v.4: (izq.) tiempo total de creación de procesos, de inicialización de procesos y total de evolución; (der.) tiempo total de ejecución de la simulación.

Los resultados mostrados en la Fig. 7. reflejan que una reducción en el número de procesos creados inicialmente para resolver la aplicación conlleva una menor demanda de trabajo por parte del entorno de ejecución a la hora de administrar los distintos procesos creados durante la ejecución del programa, mejorando significativamente los tiempos totales de ejecución a medida que se agregaron planificadores.

5. Análisis Comparativo

Una observación inmediata surge al comparar las últimas dos versiones implementadas con Erlang/OTP, donde claramente el número de procesos creados en tiempo de ejecución representa un factor determinante a la hora de medir la performance de nuestras aplicaciones.

La v.3 utilizó tantos procesos como celdas existían en el espacio celular más un conjunto de procesos auxiliares. Este escenario demandó una mayor planificación de procesos para su ejecución por parte de los distintos planificadores instanciados. Una excesiva planificación implica cambios de contexto de procesos más frecuentes para permitir la ejecución de la mayor cantidad de procesos mientras se esté utilizando algún core o CPU físico.

Todas estas latencias involucradas durante la ejecución del programa resultaron en una performance pobre y sin posibilidad de escalar. La Tabla 1 compara el rendimiento de los tiempos de ejecución obtenidos utilizando un número creciente de planificadores. En el mejor de los casos, 32 planificadores, sólo se presentó un factor de mejora de 3 veces.

Tabla 1. Factor de rendimiento para la implementación v.3.

#Planificadores	Tiempo Total de Ejecución	Factor de Rendimiento
1	197 seg.	1X
2	**318 seg.**	**0,61X**
4	**299 seg.**	**0,65X**
8	192 seg.	1,02X
16	127 seg.	1,55X
32	66 seg.	2,98X

La v. 4, utilizó un enfoque de evolución distinto. La utilización de un número reducido de procesos condujo a un menor tiempo de latencia en planificación de procesos. Menor tiempo de latencia implica mayor tiempo para ejecución de procesos mientras se esté utilizando algún core específico lo cual impacta directamente sobre los distintos tiempos de ejecución del programa.

La Tabla 2 mide el rendimiento de la última implementación. Como se puede observar, disponer de mayor cantidad de planificadores conduce a una disminución en el tiempo total de ejecución, pero la eficiencia puede verse afectada considerablemente si la aplicación no realiza un balance entre número de planificadores y el número de cores disponibles.

Tabla 2. Factor de rendimiento para la implementación v.4

#Planificadores	Tiempo Total de Ejecución	Factor de Rendimiento	Rendimiento/ #Planificadores
1	184 seg.	1X	100%
2	98 seg.	1,88X	93,88%
4	50 seg.	3,68X	92,00%
8	26 seg.	7,08X	88,46%
16	16 seg.	11,50X	71,88%
32	10 seg.	18,40X	57,50%

6 Conclusiones

En los últimos años, paulatinamente han comenzado a recibir atención propuestas de lenguajes orientados explícitamente a la programación concurrente y paralela, tal es el caso de Erlang y su entorno de desarrollo OTP. En este trabajo se discutieron una serie de implementaciones distribuidas enfocándonos principalmente en el soporte SMP que el lenguaje provee para poder ejecutar en arquitecturas multicore.

La experimentación realizada en este trabajo, intentó mostrar como los distintos factores involucrados en un programa Erlang impactan en el rendimiento obtenido, donde evidentemente, el número de procesos o planificadores juega un rol fundamental.

El trabajo realizado en este paper, será transferido al desarrollo de aplicaciones distribuidas más complejas. El uso intensivo de procesos para mantener ocupados durante el mayor tiempo posible a los distintos cores de un procesador multicore, junto a la eliminación de cuellos de botellas secuenciales y efectos colaterales deberán ser tenidos en cuenta al inferir el comportamiento de estas aplicaciones.

Referencias

1. Armstrong, J.: *Programming Erlang: Software for a Concurrent World* Pragmatic Bookshelf (2007).
2. Kowalski, R.A.: *Logic for Problem Solving*, North Holland (1979)
3. Sterling, L., Shapiro E.: *The Art of Prolog*, MIT Press, (1986).
4. Wolfram, S.: *Cellular Automata and Complexity* (collected papers). Addison Wesley. (1994).
5. Gardner, M.: *Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life"*. Scientific American 223, 120–123 (1970).
6. Logan, M., Merritt E., Carlsson, R.: *Erlang and OTP in Action* – Manning Publications; Edición: Pap/Psc. (2010).
7. Erlang Programming Language: http://www.erlang.org
8. OTP Design Principles User's Guide: http://www.erlang.org/doc/design_principles/users_guide.html