Adaptando algoritmos evolutivos paralelos al lenguaje funcional Erlang

No author given

No institute given ANONYMOUS@SECRET.COM

Resumen En el presente artículo se describe la modelación e implementación de un modelo de Computación Evolutiva sobre los paradigmas de programación funcional y concurrente al mapearse los conceptos de los modelos sobre los de los paradigmas. Bajo este enfoque son expuestas las ventajas de estos en dicho dominio de aplicación, analizándose algunas de las decisiones de diseño tomadas y viendo sus resultados en la implementación de un caso de estudio.

Key words: Algoritmos Evolutivos, Lenguajes Funcionales, Lenguajes Concurrentes, Erlang, Implementación de Algoritmos, Modelación

1. Introducción

El campo de la Computación Evolutiva (CE) es rico en características, modelos y problemas pero no lo es igual en cuanto a paradigmas de programación a usar en la implementación de los algoritmos. Tecnologías como Java, C/C++ y Perl se mantienen como las más usadas y aunque se acepta la importancia de la implementación [1], no es mucho el interés de la comunidad científica en abordar nuevos lenguajes de programación por novedosos y prometedores que pudieran ser.

Dos paradigmas emergen actualmente en la industria del software como herramientas de abstracción: el funcional y el concurrente, ante viejos y nuevos problemas. Aunque los lenguajes funcionales están a la mano desde hace buen tiempo, LISP es uno de los lenguajes de programación más longevos que existe, nunca han sido de mucho uso fundamentalmente por no disponerse de implementaciones eficientes. Esto ha ido cambiando, sus desventajas superándose y las ventajas llevando a que varios de sus conceptos sean incluidos en lenguajes orientados a objetos modernos (ejemplo C#).

El desafío de los multi-cores [2], entendiéndola como la necesidad de hacer paralelo el más simple de nuestros programas si queremos que haga uso de los modernos y potentes microprocesadores, lleva a crear (o retomar), patrones para el desarrollo de estos algoritmos muchos de los cuales acaban como nuevas características en lenguajes de programación. Ejemplos de estos son: Clojure, Go, Scala, Haskell y Erlang; en todos aparecen construcciones que soportan dentro del propio lenguaje modos de manejar la concurrencia: facilitando razonar sobre ella y simplificando su sintaxis.

El resto del trabajo se estructura como sigue: estado del arte, dentro del que se caracterizan los paradigmas funcional y concurrente y se describe el lenguaje Erlang que los incluye. A continuación se muestra la modelación de un algoritmo genético usando conceptos de los paradigmas antes expuestos (Sección 3); y finalmente, en la Sección 4, se presentan los resultados y conclusiones.

2. Estado del arte

Independientemente a la variedad de problemas a los que es aplicable la CE, en particular los Algoritmos Genéticos (AG), los conceptos en los que se basan son los mismos y así lo reflejan de una manera u otra las diferentes implementaciones realizadas. Lógicamente esto ha culminado en la creación de bibliotecas que encapsulan los elementos comunes según la variante arquitectónica que se necesite usar. A semejanza del resto del espectro de proyectos software, el paradigma dominante para la creación de tales bibliotecas es el Orientado a Objetos (OO).

2.1. Herramientas existentes

Son numerosas las herramientas desarrolladas para el trabajo con algoritmos evolutivos; en dependencia de su objetivo soportan diferentes plataformas, modelos de AE, integración con otros entornos y tamaños de los problemas. Entre las más representativas se encuentran: ECKit[1], JDEAL[2], ECJ[3], ParadisEO[4], EASEA[5] y MALLBA[6].

ECJ ECJ consiste en un conjunto de herramientas para el trabajo con poblaciones, posee facilidades para la optimización paralela y la optimización multiobjetivo [3]. Entre otras formas de representación admite la programación genética. Está implementado en Java y soporta múltiples modelos de AG paralelos: modelo de islas y celular asíncrono, maestro-esclavo y distribución coevolucionaria.

ParadisEO Marco de trabajo orientado a objetos escrito en C++ para el diseño de metaheurísticas paralelas y distribuidas, basado en Evolving Objects [4], incluye soporte para algoritmos evolutivos, búsquedas locales con soporte para los patrones paralelos y distribuidos más comunes [5]. Posee soporte solamente para los modelos clásicos de AG paralelos.

[1] http://cs.gmu.edu/ eclab/tools.html
[2] http://laseeb.isr.ist.utl.pt/sw/jdeal/home.html
[3] http://cs.gmu.edu/ eclab/projects/ecj/
[4] http://paradiseo.gforge.inria.fr/
[5] https://lsiit.u-strasbg.fr/easea/index.php/EASEA_platform
[6] http://neo.lcc.uma.es/software/mallba/index.php

EASEA *EAsy Specification of Evolutionary Algorithms* es una plataforma de evolución artificial masivamente paralela desarrollada por SONIC (Stochastic Optimisation and Nature Inspired Computing). Posee un lenguaje propio en el que se se describe el algoritmo: estructura de los individuos, operador de inicialización, función de aptitud y los operadores de crossover y mutación; a partir de esto la plataforma traduce a un conjunto de ficheros C++ que pueden además ser compilados para que usen las GPUs nVidia. Solo soporta el modelo de islas de pGA.

MALLBA MALLBA consiste en una biblioteca integrada de plantillas para optimización combinatoria: técnicas exactas, heurísticas e híbridas. Soporta ejecución tanto en ambientes secuenciales como paralelos, teniendo en cuenta el uso tanto de redes LANs como WANs [6]. Está implementada en C++ e incluye las versiones distribuidas y celulares de AG así como las técnicas CHC y μCHC.

2.2. Programación funcional

El paradigma de la programación funcional por su parte, aún cuando ofrece varias ventajas, no ha sido muy usado. Un tiempo atrás se exploraron en el campo de la Programación Genética [7,8,9], más recientemente en la neuroevolución [10]; sin embargo dentro de los AG ha sido poca su presencia [11].

La programación funcional se caracteriza por el uso de las funciones como datos (pasándolas por parámetros y devolviéndolas como resultados), en particular de las funciones puras: aquellas cuyo resultado solo depende de los parámetros de entrada, excluyendo los cambios de estado, por lo que constituye un excelente enfoque del desarrollo de algoritmos concurrentes que tienen precisamente en la comunicación via cambios de estado a la primera fuente de errores y complejidad.

El uso de listas, con implementaciones muy optimizadas, es omnipresente en la PF también, siendo en los AG también una de las estructuras de datos más utilizadas. Lo que da un conveniente pareo de conceptos entre dominio de la solución y dominio de implementación.

2.3. Erlang

Erlang es un lenguaje de programación funcional, concurrente y distribuido, adecuado para la construcción de sistemas que requieran grandes niveles de distribución, tolerancia a fallos y disponibilidad. Ha sido en más de una ocasión escogido por encima de C/C++ para el desarrollo de sistemas de uso intensivo de recursos [12] dada la eficiencia de su ejecución.

Utiliza el modelo *actor* para su implementación del paradigma de programación concurrente y posee facilidades para la integración con otros lenguajes tales como C/C++ y Java. Sus procesos son manejados por su máquina virtual (MV), que tiene un planificador por cada núcleo de la CPU, lo cual lo hacen ideal para la actual (y venideras) generación de procesadores multi-núcleos.

El que sea un lenguaje concurrente significa que posee entre sus tipos de datos el de proceso, siendo valores de primera clase, no facilidades proveídas por bibliotecas. Siendo en principio tan ligeros que una misma instancia de la MV puede tener a la vez millones en ejecución.

3. Diseñando AGs en un lenguaje funcional y concurrente

Para lograr una buena implementación de un algoritmo, cualquiera sea su naturaleza, es necesario tener en cuenta las características del lenguaje en el que se realizará así como cada concepto constituyente del dominio del problema. Se usará como caso de estudio un AG paralelo híbrido, sobre una topología de isla, en la que cada nodo será a su vez un AG concurrente basado en piscina. El lenguaje de programación será Erlang y el problema *OneMax*.

Los componentes del AG paralelo identificados como principales a la hora de diseñar la implementación aparecen listados en la Tabla 1. Las construcciones de Erlang identificadas para la modelación son las expuestas en la Tabla 2.

Componente AG	**Papel**	**Descripción**
cromosoma	Representación de la solución al problema.	cadena binaria
cromosoma evaluado	Par {cromosoma, fitness}.	cantidad de valores 1
población	Conjunto de cromosomas.	lista
cruzamiento	Relación entre dos cromosomas que da por resultado otros dos nuevos.	función de cruzamiento
mutación	Modificación de un cromosoma.	función de cambio de un valor
selección	Criterio para obtener una sublista a partir de la población.	función de selección
piscina	Población compartida entre las unidades de cálculo en un nodo.	población
isla	Nodo de la topología.	población
migración	Evento aleatorio de intercambio de cromosomas.	mensaje

Tabla 1: Componentes del AG paralelo.

3.1. Biblioteca erlEA

En la realización del caso de estudio fue implementado, teniendo en cuenta los conceptos seleccionados anteriormente, un proyecto Erlang con varios módulos. El código se encuentra bajo licencias libres en la dirección: `http://secret.com`. Sus principales módulos y funciones son descritas a continuación.

Componente Erlang	**Papel**
tupla	Tipo de datos para representar entes compuestos cuyas componentes sean de diferentes tipos y no varíen en el tiempo.
lista	Tipo de datos para representar entes compuestos cuyas componentes sean de igual tipo y varíen en el tiempo.
función	Relaciones entre datos, operaciones.
actor	Unidad de ejecución, proceso.
ets	Listado de cromosomas compartidos mediante la piscina.
módulo random	Generación de números aleatorios.

Tabla 2: Construcciones de Erlang.

Módulo reproducer Este es el módulo del proceso que selecciona la subpoblación a reproducir, los padres, realiza el cruzamiento y desencadena las migraciones. Como actor responde a los mensajes *evolve*: para realizar una iteración y *emigrateBest* para efectuar una emigración. Las funciones con las que logra esto aparecen enumeradas en la Tabla 3.

Función	**Descripción**
`extractSubpopulation(Table, N)`	A partir de una *ets* y una cantidad, selecciona de la *ets* un grupo de cromosomas.
`bestParent(Pop2r)`	Selecciona de una lista de cromosomas el mejor individuo.
`selectPop2Reproduce(Pop, N)`	Selecciona aleatoriamente un conjunto de pares de una lista de cromosomas.
`crossover(Ind1, Ind2)`	Realización de un cruce y mutación sobre el mismo, a partir de dos cromosomas.

Tabla 3: Funciones del módulo reproducer.

Módulo evaluator Este es el módulo del proceso que calcula el fitness, hace periódicas consultas sobre el pool para obtener individuos a los que calcularle el fitness. Está compuesto por la función maxOnes/1 que calcula el fitness y por el mensaje eval, dicho mensaje es el que activa al evaluador para que calcule.

Módulo poolManager Este es el módulo del proceso encargado de inicializar el trabajo de las piscinas así como enrutar los mensajes entre los evaluadores. Es el encargado de controlar la finalización del algoritmo una vez se ha encontrado la solución. Los mensajes a los que responde este actor aparecen enumerados en la Tabla 4.

Mensaje	Descripción
`evolveDone`	Finalización de una iteración de reproducción.
`evalDone`	Finalización de una iteración de evaluación.
`solutionReached`	Obtención de la solución.
`migration`	Realización de una inmigración.

Tabla 4: Mensajes a los que responde el actor del módulo poolManager.

Módulos auxiliares Los módulos ya descritos contienen toda la lógica del AG, faltan sin embargo, para que sea operativo el software, algunos códigos no funcionales.

Dichos módulos son:

experiment – Encargado de iniciar una corrida del experimento.
configBuilder – Especificación de los parámetros de un experimento.
profiler – Análisis del comportamiento: tiempos de ejecución, cantidad de iteraciones, etc.

4. Resultados y conclusiones

El diseño fue puesto a prueba con cromosomas de longitud 128, una población de 256 individuos por isla, 4 islas, 10 evaluadores y 5 reproductores por isla obteniéndose la solución del problema a los 3.922001 segundos y con 228 migraciones. Aunque los resultados no son los mejores posibles, si se obtuvo el óptimo lo que demuestra que el algoritmo fue correcto en su implementación.

Con este trabajo se muestra la simplicidad de implementación de un modelo híbrido de AG, en su versión concurrente, normalmente abrumadoramente compleja sencillamente por ser concurrente. En la arquitectura las unidades de ejecución fueron mapeadas a actores, las estructurales a módulos y las procedurales a funciones; quedando claramente identificadas y listas a futuras extensiones.

Como trabajo futuro queda la implementación de un experimento más complejo dónde se use además una arquitectura distribuida y heterogénea. Dado que el lenguaje tiene de manera nativa el soporte para la distribución de procesos y su MV está implementada para multitud de plataformas hacer esto sólo conllevaría cambios en las funciones que tratan con los cromosomas lográndose un elevando nivel de reutilización del diseño.

5. Agradecimientos

Este trabajo ha sido realizado gracias al proyecto TIN2011-28627-C04-02 (ANYSELF) auspiciado por el Mineco, y al P08-TIC-03903 del Gobierno Regional de Andalucía. Es también respaldado por el Programa de Doctorado de la AUIP y por el proyecto 83 del Campus CEI BioTIC.

Referencias

1. Merelo-Guervós, J.J., Romero, G., García-Arenas, M., Castillo, P.A., Mora, A.M., Jiménez-Laredo, J.L.: Implementation matters: Programming best practices for evolutionary algorithms. In Cabestany, J., Rojas, I., Caparrós, G.J., eds.: IWANN (2). Volume 6692 of Lecture Notes in Computer Science., Springer (2011) 333–340
2. Herb Sutter and James R. Larus: Software and the concurrency revolution. ACM Queue **3**(7) (2005) 54–62
3. Luke, S.: Essentials of Metaheuristics. Zeroth Edition (2010)
4. Keijzer, M., Merelo, J.J., Romero, G., Schoenauer, M.: Evolving objects: A general purpose evolutionary computation library. Artificial Evolution **2310** (2002) 829–888
5. Liefooghe, A., Jourdan, L., Talbi, E.: A software framework based on a conceptual unified model for evolutionary multiobjective optimization: ParadisEO-MOEO. European Journal of Operational Research (2010)
6. Alba, E., Almeida, F., Blesa, M., Cotta, C., Díaz, M., Dorta, I., Gabarró, J., León, C., Luque, G., Petit, J., Rodríguez, C., Rojas, A., Xhafa, F.: Efficient parallel LAN/WAN algorithms for optimization. the MALLBA project. Parallel Computing **32**(5-6) (2006) 415–440
7. Briggs, F., O'Neill, M.: Functional genetic programming and exhaustive program search with combinator expressions. Int. J. Know.-Based Intell. Eng. Syst. **12**(1) (2008) 47–68
8. Huelsbergen, L.: Toward simulated evolution of machine-language iteration. In: Proceedings of the First Annual Conference on Genetic Programming. GECCO '96, Cambridge, MA, USA, MIT Press (1996) 315–320
9. Walsh, P.: A functional style and fitness evaluation scheme for inducting high level programs. In Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E., eds.: Proceedings of the Genetic and Evolutionary Computation Conference. Volume 2., Orlando, Florida, USA, Morgan Kaufmann (1999) 1211–1216
10. Sher, G.I.: Handbook of Neuroevolution Through Erlang. Springer (2013)
11. Hawkins, J., Abdallah, A.: A generic functional genetic algorithm. In: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications. AICCSA '01, Washington, DC, USA, IEEE Computer Society (2001) 11–
12. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly Media, Inc. (2009)