Experimentación con algoritmos distribuidos usando herramientas libres y gratuitas

Diversos Autores

No Institute Given

Resumen En un entorno de restricción de costes para grandes instalaciones computacionales, acompañado de la existencia de herramientas en la nube y ordenadores de sobremesa de altas capacidades, la experimentación con algoritmos distribuidos se puede hacer fácilmente combinando ambas cosas. En este trabajo se presenta una metodología de experimentación con algoritmos genéticos distribuidos usando servicios de almacenamiento en la nube tales como Dropbox (o alternativas libres auto-instalables) y aplicaciones para gestión de máquinas virtuales tales como VirtualBox. Usando el almacenamiento en la nube como un sistema de intercambio de soluciones entre los diferentes nodos, se tratará de probar la aplicabilidad de esta metodología así como probar las capacidades de estos nuevos algoritmos evolutivos distribuidos.

1. Introducción

La computación paralela no necesita ser complicada y prever escenarios complejos o grandes variaciones de estructura en los programas secuenciales. La mayor parte de los ordenadores actuales pueden trabajar cómodamente con muchos procesos ejecutándose simultáneamente y poseen sistemas de almacenamiento rápido que pueden usarse para intercambiar información. Implementar un algoritmo que funcione de forma concurrente es, por lo tanto, tan simple como ejecutar varios procesos simultáneamente y que intercambien información a través de un directorio especialmente designado para ello. La eficiencia de la implementación no tiene por qué ser grande (y dependerá sobre todo del tipo de procesador, del número de núcleos que posea y de la velocidad y eficiencia del sistema de ficheros) pero, sin embargo, la simplicidad en la programación es tal que puede compensar la menor ganancia en velocidad obtenida de esta forma. Además, añadir nuevos nodos de computación al sistema paralelo distribuido resultará tan simple como ejecutar tantos procesos simultáneos como la CPU sea capaz de soportar.

Simultáneamente, está cada vez más vigente el uso de infraestructuras *nube* que permiten usar desde un ordenador conectado a la red diferentes recursos tales como CPUs virtuales o discos duros virtuales. El hecho de que sean *virtuales* implica que aparezcan, desde el punto de vista del interfaz de programación, como si se tratara de recursos disponibles desde el sistema operativo. En la práctica, podemos usar un disco duro remoto situado en la nube como si se tratara de un

disco duro local. De esta forma, también simple y transparente al programador podemos paralelizar un algoritmo, simplemente usando una infraestructura de almacenamiento virtual. A la vez, en algunos casos estas infraestructuras son gratuitas, bien por el hecho de que formen parte de la misma organización (discos conectados a la red, NAS o bien infraestructuras creadas con OpenStack u OpenNebula [5]) o bien porque se trate de productos comerciales que poseen una versión gratuita, como se trata de Dropbox, Ubuntu One u otros.

En este trabajo mostramos la primera aproximación al uso de infraestructuras virtuales para la creación de experimentos de computación distribuida de forma gratuita. Estos experimentos los aplicaremos a un tipo de algoritmo denominado algoritmo genético. a un algoritmo genético canónico.

Desde el punto de vista de este trabajo, lo interesante de los algoritmos evolutivos es que se prestan a una fácil paralelización: simplemente dividiendo la población en grupos denominados *islas* [4] y creando algún mecanismo de intercambio de individuos entre ellas se pueden paralelizar. A este mecanismo se le suele denominar *migración* y puede hacerse de diferentes maneras: síncrona o asíncronamente y usando diferentes topologías, es decir, conexiones entre las islas. No es el objetivo de este trabajo revisar estos aspectos y su influencia sobre el resultado de la ejecución del algoritmo; simplemente adoptaremos el método que suponga un cambio menor sobre el código de un algoritmo genético secuencial clásico.

Esto es precisamente lo que vamos a hacer en este trabajo. Usaremos un algoritmo genético que, dividiendo su población en varias *islas*, intercambiará información a través de un directorio compartido. Ese directorio compartido podría sincronizado estar en un servicio tal como Dropbox, pero en este caso no haremos experimentos relacionados con esto, sino que nuestro objetivo es establecer el comportamiento de un algoritmo base con todos los procesos ejecutándose en el mismo ordenador. Trabajaremos además usando un sistema de fuentes abiertas en el que tanto el código como la experimentación como este mismo trabajo están a disposición de la comunidad científica desde el momento de su creación. Todos ellos se pueden descargar de `un.url.anoni.mo`.

Con este trabajo tratamos de demostrar que, usando un mecanismo de intercambio a través de almacenamiento, se pueden conseguir mejoras de velocidad incluso en un sólo ordenador; para ello probaremos desde uno a cuatro procesos. Además, haremos ciertos experimentos preliminares que nos permitan saber qué políticas de migración son las más adecuadas.

El resto del trabajo se organiza como sigue: a continuación exponemos los resultados más sobresalientes en este área. Posteriormente explicamos el algoritmo y la metodología de experimentación usada en la Sección 3. Finalmente expondremos los resultados obtenidos y las conclusiones derivadas de los mismos para terminar con algunas notas de trabajo futuro en la Sección 4.

2. Estado del arte

Los principales resultados en este área han sido publicados recientemente [2,3,9]. En estos trabajos los autores usaron un algoritmo evolutivo implementado en Java para probar diferentes sistemas gratuitos de almacenamiento en nube tales como Dropbox u otros. Como principal resultado se obtuvo el hecho de que se conseguían escalados interesantes, pero sólo con unas pocas máquinas; el sistema se saturaba a partir de un número determinado de nodos. Además, el retraso en la propagación de los resultados de unas máquinas a otras implicaba que el problema de optimización debía de ser de cierto tamaño para que los resultados fueran significativos. El principal problema era que, debido a este retraso en la propagación, la migración introducía también un retraso en la ejecución del algoritmo para que esta fuera capaz de propagarse.

En este trabajo principalmente tratamos de reproducir los resultados obtenidos en el mismo con otro tipo de implementación usando un lenguaje de programación diferente y sin introducir retrasos cuando se hace migración y, además, usando otra implementación diferente que tiene como principal diferencia el hecho de no tener en cuenta si el algoritmo se está ejecutando secuencialmente o conjuntamente con otros nodos. Veremos en la sección siguiente estos detalles de implementación.

3. Detalles de implementación del algoritmo evolutivo y experimentos

Para hacer los experimentos se ha usado la librería `Algorithm::Evolutionary::Simple`, un módulo en Perl que permite crear un algoritmo genético rápidamente y en pocos pasos. Este módulo está optimizado para alcanzar velocidades aceptables [8] a pesar de tratarse de un lenguaje interpretado como el Perl. Este lenguaje, por otro lado, resulta un lenguaje bastante adecuado para trabajar, en general, con algoritmos evolutivos teniendo una variedad de herramientas ya desarrolladas [7] y resultando fácil la implementación de nuevas funciones, estructuras de datos u operadores. El hecho de trabajar con un diferente lenguaje, en todo caso, alterará las prestaciones de base, pero no tendría que tener tanta influencia sobre el escalado.

Para hacer los experimentos se ha buscado un problema que represente cierto reto para un algoritmo genético y cuya evaluación también requiera cierto tiempo, de forma que el algoritmo necesite un número de evaluaciones alto que pueda mejorarse a base de la paralelización. Por eso la función elegida ha sido P-Peaks [1]. En esta función se general aleatoriamente un conjunto de p cadenas binarias de longitud b. P-Peaks devuelve la distancia a la cadena *más cercana*, es decir, el mínimo de las distancias medida a todas las cadenas. La función resulta *pesada* porque hay que medir distancias a un número determinado de cadenas y resulta complicada para un algoritmo evolutivo al tener un número alto de máximos globales (correspondientes a cada una de las cadenas que se han generado). La implementación de esta función es también libre,

Cuadro 1: Valores de los parámetros del algoritmo genético y de la función P-Peaks usada.

Parámetro	Valor
Selección	Rueda de ruleta
Mutación	1-bit
Entrecruzamiento	2 puntos
P (número de picos)	256
b (bits del cromosoma)	512
Población base	1024

está escrita en Perl y forma parte del módulo `Algorithm::Evolutionary` denominándose `Algorithm::Evolutionary::Fitness::P_Peaks`; este módulo es estándar y está incluido en la librería `Algorithm::Evolutionary` mencionada anteriormente: `http://search.cpan.org/~jmerelo/Algorithm-Evolutionary-0.78/lib/Algorithm/Evolutionary/Fitness/P_Peaks.pm`

Los parámetros base usados en el algoritmo evolutivo se muestran en la tabla 1. En general, son los valores por omisión de la librería. Sin embargo, como se ha comentado, P y b han sido elegidos para hacer el trabajo suficientemente difícil como para que cada ejecución del algoritmo dure un tiempo considerable. Por otro lado, tratándose de un problema difícil, se ha escogido un tamaño grande de población para que el algoritmo sea capaz de encontrar la solución; con tamaños más pequeños de población (o subdivisiones del tamaño) se vio que en muchos casos el algoritmo se quedaba estancado y era incapaz de encontrar la solución.

Para probar las prestaciones en paralelo del algoritmo se dividió la población entre dos y entre cuatro y se probó con un número igual de procesos. Los procesos se lanzaban desde un programa de línea de órdenes de Linux de forma que el inicio de los procesos es simultáneo (salvo tiempo de inicio de un proceso, que es muy bajo en todo caso). Había un proceso *principal* y otros *secundarios*; la principal diferencia es que este proceso *principal* decide cuando comienza y termina cada algoritmo y se usa también para medir la duración de los mismos. Esto puede significar que, si la solución se encuentra por primera vez en alguno de los procesos secundarios, el programa puede continuar durante un tiempo adicional; sin embargo, no suele demorarse mucho dado que se intercambian cromosomas entre unos procesos y otros. El hacerlo así, además, evita que hagan falta mecanismos adicionales de comunicación del final que tengan que propagarse y simplifica la programación que, en realidad, es exactamente la misma para el programa secuencial y el paralelo. El tiempo de ejecución se mide simplemente a través de la diferencia en segundos entre el tiempo de creación de dos ficheros que se crear, precisamente, al comenzar y terminar este programa.

Para la versión paralela hace falta una *política de migración*. En nuestro caso se ha elegido guardar un cromosoma aleatorio elegido entre el 50 % con más fitness y tomar, a la vez, uno aleatorio del directorio en el que lo han depositado el resto. El elegir uno aleatorio entre los mejores coincide con la política que

suele obtener mejores resultados en modelos isla, según hemos podido establecer en el pasado [6]. De hecho, algunas pruebas hechas depositando el mejor en cada generación ha dado peores resultados, provocando que en la mayor parte de los casos no termine el algoritmo. La estrategia aleatoria, aparte de rápida (no necesita ordenar ni hacer ninguna otra operación) tiene la ventaja de la estrategia de depósito de cromosomas: no necesita tampoco llevar a cabo ningún cambio en el código para el caso paralelo.

Tanto el código como el resultado de los experimentos (que analizaremos en la siguiente sección) están disponibles de forma abierta en la siguiente dirección: `https://code.launchpad.net/\~jjmerelo/simplea/trunk`. El objetivo es que la comunidad científica se beneficie de esta ciencia abierta no sólo en los resultados, sino también en los datos que podrán ser, en caso de desearlo, analizados de forma independiente.

4. Resultados, conclusiones y trabajo futuro

Los experimentos se ejecutaron 10 veces. En todos los casos se encontró la solución, salvo en el caso en el que se hizo con cuatro nodos, en el que acabó sólo en 6 ocasiones. En todo caso, los resultados se muestran sobre los que efectivamente terminaron.

El primer resultado es que efectivamente, a pesar de ejecutarse en un sólo ordenador y cargar la tabla de procesos del mismo, se consigue una mejora en la velocidad con el número de *nodos*. Esto se muestra en las Figuras 1 y 2; en el primer caso los tiempos han sido tomados en un ordenador AMD con séxtuple núcleo ejecutando Ubuntu 12.04; en el segundo es un portátil Toshiba Portegé ejecutando el mismo sistema operativo y con un Intel i5 de procesador; en este caso el disco duro es un SSD lo que lo hace, teóricamente, más rápido que en el primer caso.

Como se puede ver en ambas gráficas, el añadir nodos consigue rebajar el tiempo necesario para hallar la solución. De hecho, parte de esta mejora se debe al hecho de que se usen menos individuos en la población; pero una parte también se debe a que se están evaluando simultáneamente más individuos. De hecho, la mejora para cuatro nodos es de un 70 % en la velocidad en el primer caso, mientras que la mejora para dos nodos es de un 40 % en el segundo caso, algo mejor en el primer caso. Esto, en parte, puede deberse al hecho de que realmente no nos preocupamos de cuando halla la solución cualquier nodo, sino sólo cuando la halla uno de ellos; pero en parte también a que se nota la carga del sistema al ejecutar varios procesos simultáneamente, resultando excesiva tanto para la CPU como para la entrada/salida. De hecho, en el primer caso las tres diferencias son significativas usando el test no paramétrico de Wilcoxon, mientras que en el segundo caso la diferencia no es significativa. Esto puede significar que la mayor velocidad del disco duro haga que la diferencia entre uno y otro sea menor al dedicar menos tiempo al escribir en disco; esto podría indicar que uno de los factores que influyen en el tiempo es el tiempo empleado en leer del sistema de ficheros. Habrá que evaluar mediante un profiler este tipo de hipótesis para

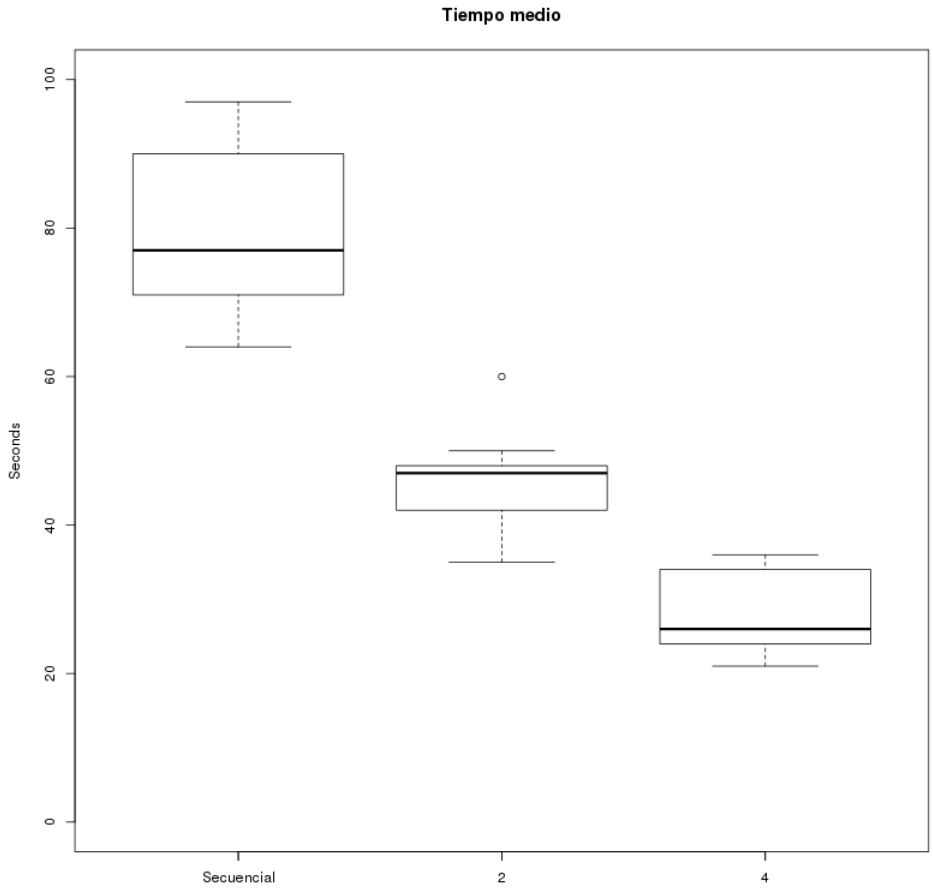

Figura 1: Boxplot del tiempo medio (en segundos) necesario para terminar el algoritmo usando un programa secuencial y dos y cuatro procesos simultáneos. Los tiempos han sido tomados en un ordenador de sobremesa.

probar si es cierta o no, aunque una evaluación preliminar indica que el tiempo invertido en leer y escribir del disco duro es tres órdenes de magnitud inferior al tiempo total del programa.

Por otro lado, en el primer caso se puede ver en la figura 3 la evolución procede de forma bien diferente dependiendo del número de procesos. En este caso lo que se traza para cada número de procesos es el tiempo de creación del fichero con el individuo que se está migrando tomado directamente del sistema de ficheros; el tiempo tiene resolución de segundos; esto es una ventaja adicional de usar este sistema, que te permite ver la evolución, con el tiempo, del fitness. Si recordamos que en realidad el que se graba es un individuo aleatorio entre

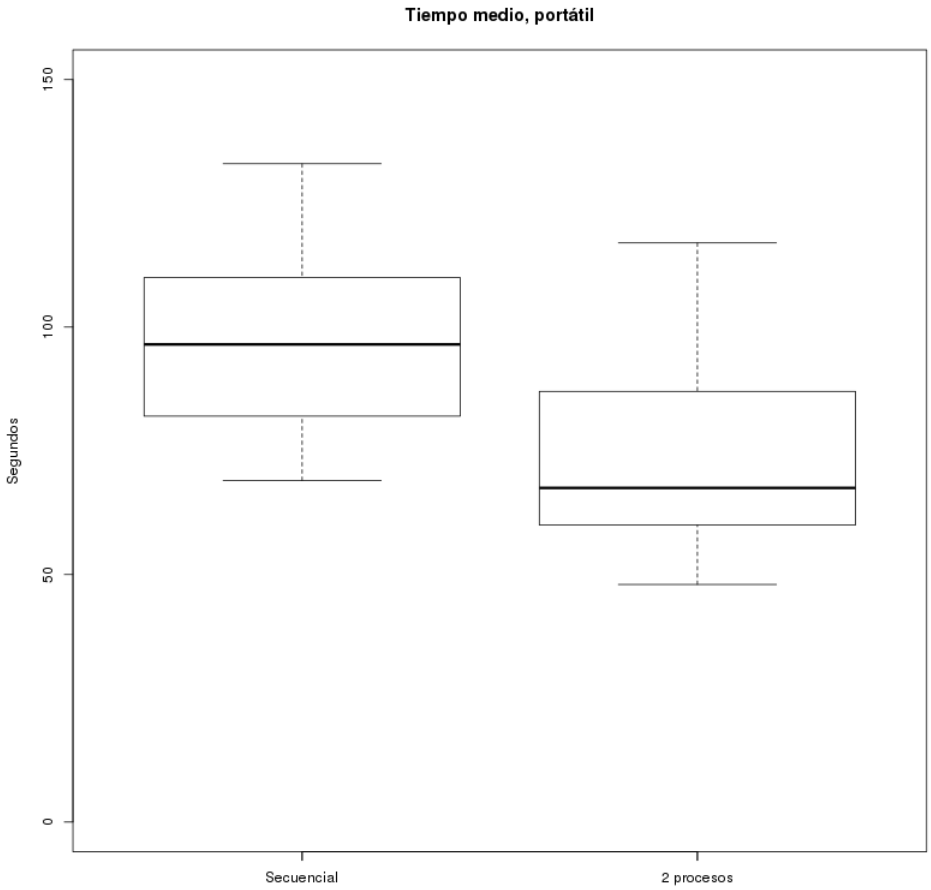

Figura 2: Boxplot del tiempo medio (en segundos) necesario para terminar el algoritmo usando un programa secuencial y dos procesos simultáneos en un ordenador portátil.

los 50 % mejores, vemos que, en todo caso y para un tiempo determinado, por ejemplo 10 segundos, cuando hay 4 nodos se ha avanzado mucho más que cuando se usan dos o un solo nodo; esto prueba que el algoritmo evolutivo se puede paralelizar usando este simple mecanismo que es, también, extensible a sistemas que permitan almacenamiento transparente en la nube como Dropbox, tal como queríamos probar.

Adicionalmente hemos hecho alguna prueba con directorios compartidos a través de Dropbox y alojados en diferentes máquinas virtuales dentro del mismo ordenador. Esto presenta una serie de retos, el principal que la velocidad de tales máquinas virtuales va a ser muy diferente y continuando con el problema

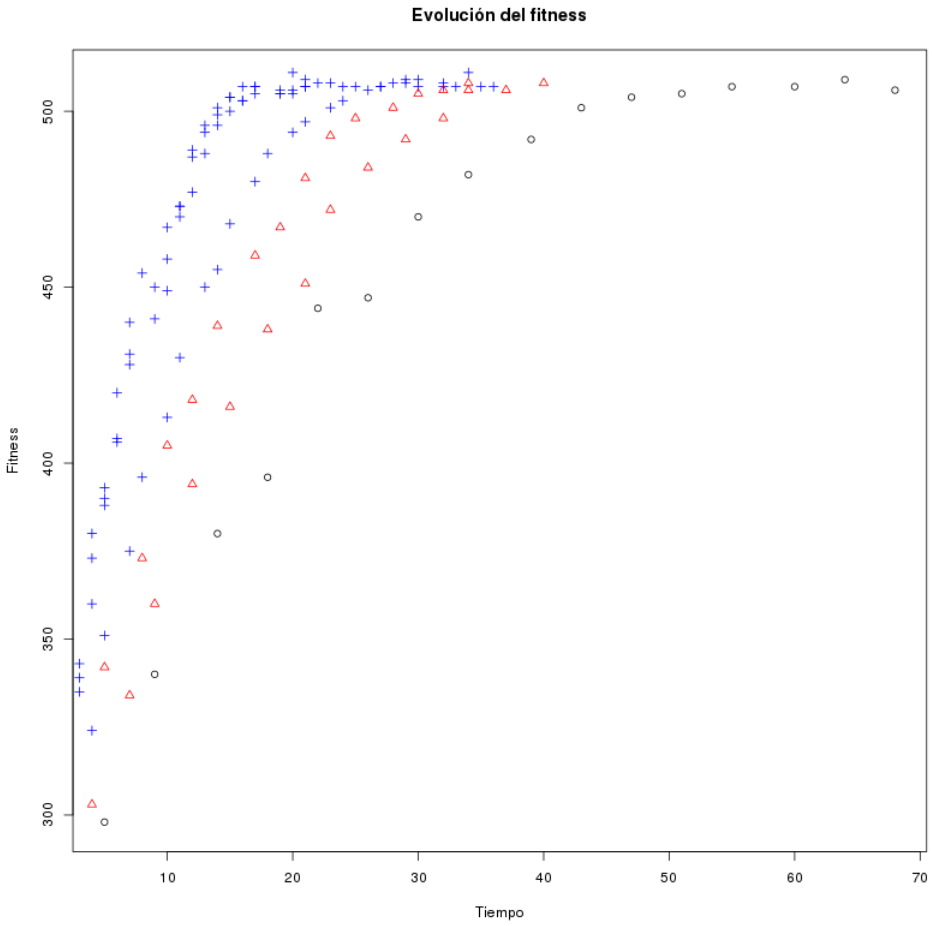

Figura 3: Evolución del fitness de una instancia de cada uno de los tres experimentos; los círculos indican el experimento con un solo proceso, los triángulos con dos y las cruces con 4 procesos. El tiempo en el eje de abscisas es el tiempo real; el fitness máximo es 512.

del retraso en la aparición de los ficheros individuales en el resto de los nodos. Sin embargo, algunas pruebas iniciales (que se pueden ver en el repositorio de código y datos indicados) indican que, aunque se consiguen ciertas mejoras al añadir un nuevo nodo, no está claro que sean significativas, por lo que hay que avanzar haciendo experimentos en este sentido, probando con diferentes configuraciones, máquinas virtuales y parametrización de las mismas. Esto es algo que se propone como trabajo futuro.

Este trabajo ha sido el comienzo de una línea de investigación en la que se pretenden montar máquinas paralelas con la infraestructura de la que se

dispone en cualquier despacho o laboratorio. Utilizándo un multiprocesador (varias máquinas con espacio de direcciones distribuido unidas por una red de interconexión) y un espacio de almacenamiento en disco al que todos los procesadores pueden acceder, se ha reproducido el sistema de comunicación de un multiprocesador con espacio de direcciones compartido, probando que efectivamente la arquitectura es posible y es posible ejecutar un algoritmo evolutivo utilizando esta infraestructura, contando sólo con los recursos disponibles y un esquema de programación SPMD.

Los tiempos empleados en la comunicación van retardados por la necesidad de escribir en disco para que los procesos se comuniquen, y esa será nuestro siguiente objetivo, intentar simplificar al máximo la programación intentando evitar al máximo este tipo de operaciones, acercando más la arquitectura propuesta a una más eficiente.

Aunque los resultados obtenidos no son sorprendentes, nuestro objetivo no es superar la velocidad de las máquinas diseñadas para computación paralela, sino simplemente probar que existe la posibilidad de ejecutar en paralelo con los recursos mínimos una aplicación cualquiera.

Por otro lado, una vez más habrá que hacer un profiling extensivo de la aplicación para detectar los cuellos de botella y comprobar qué provoca retardos y en qué caso. También se comprobará con diferentes tipos de configuraciones que incluyan directorios sincronizados usando almacenamiento en la nube o directorios montados remotamente usando otro tipo de mecanismos. Eventualmente, lo que se pretende es crear un sistema que pueda distribuirse fácilmente entre diferentes ordenadores y crear experimentos distribuidos sin necesidad de ninguna compra ni de hardware ni de servicios.

5. Agradecimientos

Referencias

1. Enrique Alba, Mario Giacobini, Marco Tomassini, and Sergio Romero. Comparing synchronous and asynchronous cellular genetic algorithms. In *Parallel Problem Solving from Nature – PPSN VII*, pages 601–610. Springer, 2002.
2. Maribel García Arenas, Juan J. Merelo Guervós, Antonio Miguel Mora, Pedro A. Castillo, Gustavo Romero, and Juan Luís Jiménez Laredo. Assessing speed-ups in commodity cloud storage services for distributed evolutionary algorithms. In *IEEE Congress on Evolutionary Computation*, pages 304–311. IEEE, 2011.
3. Maribel García Arenas, Juan Julián Merelo Guervós, Pedro A Castillo, Juan Luís Jiménez Laredo, Gustavo Romero, and Antonio Miguel Mora. Using free cloud storage services for distributed evolutionary algorithms. In Natalio Krasnogor and Pier Luca Lanzi, editors, *GECCO*, pages 1603–1610. ACM, 2011.
4. Juan J. Merelo Guervós, Antonio Miguel Mora, Carlos M. Fernandes, Anna Isabel Esparcia-Alcázar, and Juan Luís Jiménez Laredo. Pool vs. island based evolutionary algorithms: An initial exploration. In Fatos Xhafa, Leonard Barolli, and Kin Fun Li, editors, *3PGCIC*, pages 19–24. IEEE, 2012.
5. Francisco Magaz Villaverde. Opennebula y hadoop: Cloud computing con herramientas open source. 2012.

6. Juan J. Merelo, Antonio M. Mora, Pedro A. Castillo, Juan L. J. Laredo, Lourdes Araujo, Ken C. Sharman, Anna I. Esparcia-Alcázar, Eva Alfaro-Cid, and Carlos Cotta. Testing the intermediate disturbance hypothesis: Effect of asynchronous population incorporation on multi-deme evolutionary algorithms. In Gunter Rudolph, Thomas Jansen, Simon Lucas, Carlo Poloni, and Nicola Beume, editors, *Parallel Problem Solving from Nature - PPSN X*, volume 5199 of *LNCS*, pages 266–275, Dortmund, 13-17 September 2008. Springer.
7. Juan-Julián Merelo. A Perl primer for evolutionary algorithm practitioners. *SIGEVOlution*, 4(4):12–19, 2010.
8. Juan-Julián Merelo-Guervós, Gustavo Romero, Maribel García-Arenas, Pedro A. Castillo, Antonio-Miguel Mora, and Juan-Luís Jiménez-Laredo. Implementation matters: Programming best practices for evolutionary algorithms. In Joan Cabestany, Ignacio Rojas, and Gonzalo Joya Caparrós, editors, *IWANN (2)*, volume 6692 of *Lecture Notes in Computer Science*, pages 333–340. Springer, 2011.
9. K Meri, MG Arenas, AM Mora, JJ Merelo, PA Castillo, P García-Sánchez, and JLJ Laredo. Cloud-based evolutionary algorithms: An algorithmic study. *Natural Computing*, 2013.