Evaluación distribuida transparente para algoritmos evolutivos en JCLEC

Franciso Ibáñez, Alberto Cano, and Sebastián Ventura

Departamento de Informática y Análisis Numérico,
Universidad de Córdoba, España
{fipastor,acano,sventura}@uco.es

Abstract. La evaluación de los individuos en un algoritmo evolutivo constituye generalmente la etapa con un mayor coste computacional. Este hecho se acentúa en los problemas de minería de datos debido al cada vez mayor tamaño de los conjuntos de datos. Existen múltiples propuestas y paquetes software para la paralelización y distribución del cómputo en CPUs multinúcleo, GPUs, y clústeres de nodos de cómputo, pero estos requieren de la reimplementación del código y sus características no siempre se ajustan a la naturaleza de los algoritmos evolutivos. En este trabajo presentamos un *wrapper* para evaluadores en JCLEC que permite la evaluación distribuida de individuos de forma totalmente transparente y sencilla para el usuario sin la necesidad de recodificar. El *wrapper* se encarga automáticamente de la distribución de la población y sincronización de las conexiones. El usuario únicamente ha de especificar en el fichero de configuración los datos de conexión de aquellas máquinas que desee emplear. El estudio experimental analiza la escalabilidad del *wrapper* en un clúster de 12 nodos con un total de 144 CPUs. Los resultados indican el buen desempeño del *wrapper* siendo capaz de aprovechar todos los recursos computacionales de forma transparente.

Keywords: Algoritmos evolutivos, evaluación distribuida, paralelización

1 Introducción

Los algoritmos evolutivos son métodos de optimización basados en los principios de la evolución natural. Los individuos en un algoritmo evolutivo son iterativemente mejorados y la calidad de su solución se evalúa mediante una función de fitness. Los algoritmos poblacionales evolucionan un conjunto de individuos para los que en cada iteración es necesario evaluar su calidad, lo que representa un gran coste computacional. Este problema se acentúa cuando se aplican a la resolución de problemas cuya función de fitness es especialmente costosa, como por ejemplo en minería de datos [1]. Afortunadamente estos problemas son intrínsicamente paralelos y existe una gran facilidad y variedad de opciones para acelerar su resolución mediante técnicas de paralelización y distribución de cómputo [2, 3]. La explosión de la cantidad de información en minería de datos ha motivado una gran expectación alrededor del término Big Data [4, 5], y existen numerosas propuestas software destinadas a facilitar la computación paralela y distribuida en este ámbito. Sin embargo, frecuentemente la curva de aprendizaje de nuevas metodologías motiva el rechazo o desinterés de los usuarios en el uso de estas

nuevas tecnologías paralelas, especialmente cuando requieren de la reimplementación del código del usuario y de su rediseño enfocado a entornos específicos de computación paralela, que pueden resultar estrictos y poco flexibles a las necesidades específicas de cómputo del usuario.

La motivación de este trabajo es proponer un *wrapper* que permita distribuir de forma automática y transparente la evaluación de los individuos de la población de un algoritmo evolutivo en JCLEC [6]. El *wrapper* envuelve un evaluador base (el que implementó el usuario) que se instancia en cada uno de los nodos de cómputo remotos y actúa como un servicio remoto que recibe individuos y devuelve su fitness. El *wrapper* por lo tanto se encargará de distribuir de forma automática los individuos de la población en cada uno de los nodos. La máxima de este trabajo es facilitar la transparencia en la paralelización y distribución del cómputo, sin intervención del usuario ni la recodificación de los evaluadores.

La propuesta se evalúa en un conjunto de problemas del módulo de clasificación de JCLEC [7] empleando conjuntos de datos de diversa dimensionalidad y se mide el tiempo de ejecución de la evaluación empleado por el evaluador base en sus versiones monohilo y multihilo, y con la implementación distribuida realizada por el *wrapper*. Los experimentos se han realizado en un clúster con 12 nodos de cómputo y un total de 144 núcleos. Los resultados obtenidos apoyan la escalabilidad alcanzada por el *wrapper*, logrando aprovechar todos los recursos computacionales disponibles, obteniendo así la máxima paralelización posible.

La estructura del trabajo es la siguiente. La Sección 2 realiza una breve revisión de las estrategias de paralelización de algoritmos evolutivos. La Sección 3 presenta nuestra propuesta del *wrapper* para evaluación distribuida. La Sección 4 muestra el estudio experimental y analiza los resultados. Finalemente, la Sección 5 recopila las conclusiones del trabajo.

2 Paralelización en algoritmos evolutivos

Los algoritmos evolutivos son métodos de optimización complejos y poseen un alto coste computacional. Sin embargo, existen múltiples oportunidades que permiten su paralelización. Los algoritmos evolutivos son métodos intrínsicamente paralelos [2, 3] puesto que la evaluación de cada uno de los individuos de la población es paralelizable al no existir interdependencia entre ellos desde el punto de vista del cómputo. Esto es lo que se conoce como paralelismo a nivel de población. Además, frecuentemente es posible paralelizar internamente la evaluación dentro de cada uno de los individuos, esto se conoce como paralelismo a nivel de individuo. Concretamente, el paralelismo a nivel de individuo cuando lo que se paraleliza es el tratamiento de datos de forma concurrente también se conoce como paralelismo a nivel de datos. Ambas formas de paralelismo se pueden combinar y resultar en modelos paralelos a nivel de población y a nivel de individuo [8].

Estos modelos se adaptan a los componentes hardware disponibles en cada caso. Así pues, la disponibilidad de procesadores multinúcleo a nivel de usuario ha facilitado el desarrollo de modelos paralelos que permiten aprovechar 4, 6 u 8 núcleos en las CPUs más recientes. El paralelismo a nivel de hilo en CPU es relativamente sen-

cillo de cara a la implementación de usuario, y se beneficia de la rápida intercomunicación de datos entre los núcleos y memoria principal compartida. Sin embargo, la aceleración que se puede obtener es relativamente pequeña debido al poco número de núcleos disponibles en un procesador. La paralelización en tarjetas gráficas (GPUs) supuso un enorme salto en el grado de paralelización disponible, facilitando que millones de hilos pudiesen colaborar bajo un paradigma de paralelización a nivel de población y datos [9]. Sin embargo, la codificación en GPU no es sencilla y habitualmente los usuarios de algoritmos evolutivos no se motivan a recodificar sus algoritmos en otros lenguajes o para arquitecturas hardware específicas. Finalmente, los modelos distribuidos permiten la interconexión en red de nodos de cómputo para aprovechar su potencial conjunto [10, 11]. Sin embargo, de nuevo es complicado convencer al usuario de moverse a metodologías de programación escalables y distribuidas, como por ejemplo MapReduce [12], ya que frecuentemente rechazan la idea de recodificación de sus algoritmos o de modificación del flujo de ejecución secuencial del código hacia uno paralelo. Además, las implementaciones concretas pueden resultar demasiado estrictas para ajustarse a las necesidades específicas de cómputo de los usuarios, como por ejemplo el sistema de ficheros HDFS de Hadoop [13], o el modelo de lenguaje funcional de Spark [14]. Por lo tanto, creemos conveniente el desarrollar un modelo que facilite un modelo de cómputo distribuido transparente sin que el usuario tenga que aprender otro lenguaje/librería, recodificar su código, o tener conocimientos de paralelismo.

3 Wrapper para evaluación distribuida

La idea del diseño del *wrapper* se basa en la simplificación, transparencia, y reusabilidad el código. El *wrapper* contiene un evaluador base que instanciará en cada uno de los nodos de cómputo. El evaluador base es el que el usuario implementó para su algoritmo. Este diseño de reusabilidad de los evaluadores ya implementados está basado en patrón de diseño *decorator*. Esto nos permite no tener que crear sucesivas clases que hereden de los evaluadores incorporando la nueva funcionalidad de evaluación remota. Los nodos de cómputo se indican en el fichero de configuración del experimento, y simplemente es necesario conocer su dirección IP y un usuario/contraseña con el cual poder conectarse, no siendo necesaria ninguna intervención por parte del usuario. Por lo tanto, se delega en el *wrapper* la responsabilidad de conectarse a los nodos, transferir los ejecutables y ficheros de datos necesarios, y de levantar en cada nodo el servicio de evaluación a la espera de atender peticiones remotas. De esta forma se evita la intervención del usuario en la transferencia de ficheros, ejecución de comandos o interactuación con los nodos remotos. Así pues, la máxima de no intervención permite en última instancia el empleo transparente de todos los recusos disponibles, como por ejemplo utilizar todos los PCs de un aula de informática indicando simplemente la lista de IPs, sin tener que hacer ninguna acción en los PCs. La Figura 1 muestra un ejemplo de la parte del fichero de configuración resultante para el empleo de tres equipos.

La implementación del *wrapper* se realiza mediante Java RMI y llamadas a procedimientos remotos. La clase *net.sf.jclec.DistributedEvaluator* se encarga de leer los parámetros del evaluador base y los datos de conexión a los nodos de cómputo. La clase implementa la interfaz *IDistributedEvaluator* mostrada en la Figura 2, que especifica

```
<evaluator type="net.sf.jclec.DistributedEvaluator">
  <base-evaluator type="net.sf.jclec.experiments.MyEvaluator">
    ... base-evaluator parameters ...
  </base-evaluator>
  <remote-hosts>
    <host ip="192.168.1.205" username="user1" password="passwd2"/>
    <host ip="192.168.1.207" username="user2" password="passwd2"/>
    <host ip="192.168.1.213" username="user3" password="passwd3"/>
  </remote-hosts>
</evaluator>
```

Fig. 1. Configuración del experimento usando un evaluador distribuido.

los métodos contenidos en el evaluador distribuido. El método de configuración del evaluador distribuido crea tantos hilos como nodos se hayan especificado para proceder a levantar el servicio de evaluación remota en cada máquina. El API de la librería JSch facilita las herramientas para la conexión remota por SSH a cada nodo, la transferencia del *jar* ejecutable y de los ficheros de datos (en su caso). Finalmente, se ejecuta el servicio remoto a la espera de peticiones de evaluación de individuos, por lo tanto este servicio es persistente y permite minimizar el tiempo de respuesta cada vez que se requiera realizar una llamada a la evaluación remota de individuos. El método shutdown permite terminar los servicios remotos cuando el algoritmo haya finalizado, liberando los recursos en los nodos de cómputo.

La población se divide de forma uniforme entre el número de unidades de cómputo disponible, de tal forma que si se dispone de máquinas heterogéneas con un diferente número de núcleos, se balanceará automáticamente el número de individuos a evaluar en cada nodo. Así pues, nodos con un mayor número de CPUs recibirán un mayor número de individuos a evaluar. Este balanceo se realiza automáticamente mediante la comunicación por parte de los nodos de cómputo del número de CPUs disponibles. Opcionalmente se puede especificar de forma manual en el fichero de configuración el número de núcleos a emplear en caso de no querer emplear todas las CPUs. La Figura 3 muestra el flujo de trabajo para la distribución de la evaluación en nodos de cómputo remotos.

Esta jerarquía en la granularidad de la evaluación de los individuos permite que en el caso de disponer de unidades hardware específicas, por ejemplo GPUs, en algunos de los nodos de cómputo, se puedan aprovechar dichas arquitecturas heterogéneas para paralelizar la evaluación siguiendo la metodología descrita en los siguientes trabajos de evaluación de evolutivos en GPU [15–18].

```
public interface IDistributedEvaluator extends Remote
{
  public void configure(String settings) throws RemoteException;
  public IFitness[] evaluate(List<IIndividual> inds) throws RemoteException;
  public IFitness evaluate(IIndividual ind) throws RemoteException;
  public void shutdown() throws RemoteException;
}
```

Fig. 2. Interfaz del evaluador distribuido.

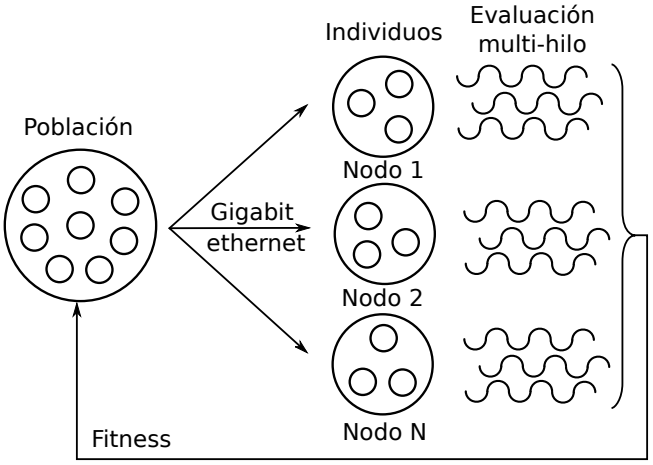

Fig. 3. Flujo de evaluación distribuida de los individuos de la población.

4 Estudio experimental

El estudio experimental analiza la escalabilidad de la evaluación de individuos en problemas de clasificación empleando un algoritmo de programación genética. Para ello contamos con un clúster compuesto por un nodo principal más 12 nodos de cómputo con 2 procesadores Intel Xeon E5645 @ 2.4 GHz de 6 núcleos cada uno y sistema Rocks cluster 6.1.1 x64. La suma total es de 144 núcleos de cómputo con montante de 448 GB de memoria, conectados por gigabit ethernet.

Se comparará el tiempo de evaluación de una población de individuos empleando un evaluador base monohilo (secuencial) y multihilo (paralelo), así como el *wrapper* para la evaluación distribuida. La Tabla 1 muestra los resultados de los tiempos de evaluación (ms) para una población de 250 individuos, así como las aceleraciones relativas obtenidas entre los evaluadores. La tabla se encuentra ordenada según el número de instancias del conjunto de datos. Los tiempos mostrados en la tabla representan el tiempo total que se requiere para evaluar la población, que incluye tanto el tiempo de evaluación local/remoto, como el coste de transmisión de datos por red y de creación de hilos en sus respectivos escenarios, para reflejar una comparativa justa.

Los resultados son muy significativos ya que se pueden extraer varias conclusiones muy interesantes. La primera es la clara tendencia lineal del aumento del tiempo de

Dataset	Instancias	Secuencial	Paralelo	Distribuido	Paralelo vs Secuencial (×)	Distribuido vs Paralelo (×)	Distribuido vs Secuencial (×)
iris	150	18	4	28	4.14	0.16	0.65
sonar	208	25	6	29	4.05	0.22	0.87
glass	214	26	8	30	3.42	0.25	0.87
heart	270	34	9	31	3.57	0.30	1.08
ionosphere	351	44	8	33	5.59	0.24	1.35
pima	768	98	11	33	9.09	0.32	2.95
vowel	990	128	20	34	6.34	0.60	3.81
texture	5,500	697	92	58	7.59	1.58	11.98
mushroom	5,644	697	83	59	8.38	1.41	11.78
satimage	6,435	840	84	61	10.00	1.37	13.70
thyroid	7,200	918	85	73	10.77	1.17	12.57
twonorm	7,400	920	73	63	12.64	1.16	14.61
penbased	10,992	1,389	109	92	12.80	1.18	15.15
kr-vs-k	28,056	3,302	311	182	10.62	1.71	18.16
shuttle	58,000	7,237	1,597	311	4.53	5.14	23.29
census	299,284	42,843	9,937	971	4.31	10.24	44.15
kddcup	494,020	70,561	16,498	1,649	4.28	10.01	42.79
airlines	539,383	75,609	22,086	1,869	3.42	11.82	40.46
covtype	581,012	81,072	21,126	1,962	3.84	10.77	41.31
poker	1,025,009	135,912	33,717	3,344	4.03	10.08	40.65
kddcupfull	4,898,431	684,872	167,494	14,093	4.09	11.89	48.60

Table 1. Resultados. Tiempos de evaluación (ms) y aceleración relativa (×).

evaluación secuencial según aumenta el tamaño del conjunto de datos, cuestión esperable debido a la complejidad lineal de la función de evaluación con respecto al número de instancias. La segunda es la gran ventaja que supone el uso de un procesador multinúcleo para realizar una evaluación paralela, llegando a obtener aceleraciones cercanas al número de núcleos disponibles (12) en cada procesador. Sin embargo, es muy destacable mencionar que a partir de cierto tamaño del conjunto de datos, esta aceleración se decrementa hasta valores cercanos a $4\times$. Esto es debido al mayor tamaño en memoria principal del conjunto de datos y a la paginación que sufre la memoria, atendiendo al acceso concurrente de múltiples hilos (12) a grandes segmentos de memoria ocupados por el conjunto de datos, lo que produce desalineamiento en la lectura de memoria. Este comportamiento es intrínseco al modelo de programación de hilos y memoria de la máquina virtual de Java. La tercera es el comportamiento del evaluador distribuido, en el que se observa un tiempo mínimo base entorno a los 30 ms, independientemente del pequeño tamaño del dataset. Este tiempo realmente representa un overhead introducido por las latencias que ocurren en las transferencias y comunicaciones en red, tiempo relativamente pequeño pero que es inevitable. Este overhead podría reducise con tecnologías específicas de alta velocidad y baja latencia como InfiniBand. El overhead provoca que para conjuntos de datos pequeños (menores a 1,000 instancias) no sea recomendable pasar a un evaluador distribuido frente a uno paralelo (aceleraciones < 1). Sin embargo, se observa que conforme aumenta el tamaño del conjunto de datos la aceleración obtenida aumenta, destacando que para los grandes conjuntos de datos, los 12 nodos de cómputo en la evaluación distribuida se acercan a un $12\times$ frente a la evaluación paralela, verdadero objetivo de este trabajo. Finalmente, si comparamos los resultados con respecto a la evaluación secuencial se observa un rendimiento de hasta

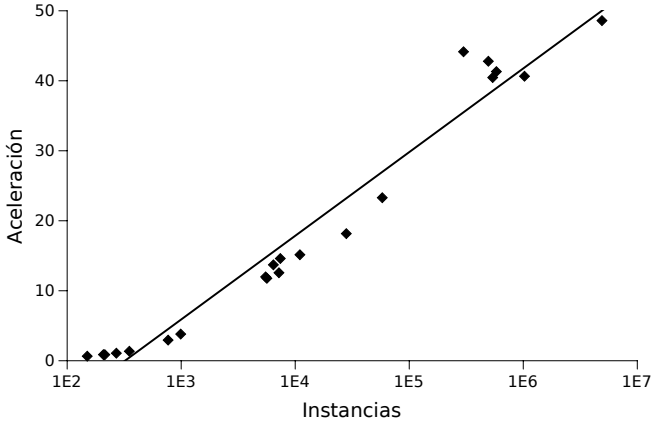

Fig. 4. Aceleración del evaluador distribuido con respecto al secuencial.

48×. La tendencia del evaluador distribuido se aprecia en la Figura 4 donde se ilustra la aceleración con respecto al evaluador secuencial. Este gran rendimiento es muy destacable, especialmente si atendemos a que ha sido con un coste de recodificación cero para el usuario.

5 Conclusiones

En este trabajo hemos presentado una solución de escalabilidad y de distribución de cómputo empleando un *wrapper* para evaluadores de algoritmos evolutivos en JCLEC. El *wrapper* permite la distribución y paralelización de la evaluación de individuos de forma totalmente transparente para el usuario, sin que tenga que recodificar su software ni poseer conocimientos de paralelismo. La implementación mediante llamadas a procedimientos remotos con RMI permite comunicaciones eficientes memoria-memoria con los nodos de cómputo, permitiendo obtener el resultado de la evaluación con tiempo de respuesta y overhead mínimo. El automatismo en la copia de los archivos ejecutables y de datos a los nodos de cómputo, así como el inicio/apagado de los servicios remotos de forma automática facilita enormemente el aprovechamiento de los recursos computaciones sin la intervención del usuario. Los resultados de los experimentos evaluados en problemas de clasificación demuestran el gran desempeño del *wrapper*, facilitando la escalabilidad de los algoritmos a grandes conjuntos de datos o a grandes poblaciones, donde el tiempo de la función de evaluación es crítico para el algoritmo evolutivo.

Como futuro trabajo nos planteamos añadir además una distribución automática del cómputo a nivel de datos para problemas específicos en minería de datos, frente al modelo propuesto en este trabajo que es válido para cualquier problema que afronte un algoritmo evolutivo. La idea para afrontar realmente problemas de big data implica la división horizontal del conjunto de datos en las unidades de cómputo (mapeo), de fal forma que cada nodo pudiese almacenar en memoria principal un subconjunto de

instancias. Sin embargo, para ello se requiere de una recodificación del evaluador, para que cada procedimiento remoto no devuelva un fitness, sino una submatriz de confusión parcial correspondiente al subconjunto de datos evaluado en cada nodo. Finalmente, la agregación (reducción) se corresponde con la adición de las submatrices en la matriz de confusión final y el cálculo de las métricas de fitness.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Economía y Competitividad bajo el proyecto TIN2014-55252-P, los fondos FEDER, y la beca FPU AP2010-0042.

References

1. Freitas, A.: A review of evolutionary algorithms for data mining. Soft Computing for Knowledge Discovery and Data Mining (2008) 79–111
2. Nedjah, N., Alba, E., de Macedo Mourelle, L.: Parallel evolutionary computations. Studies in Computational Intelligence **22** (2006)
3. Luque, G., Alba, E.: Parallel genetic algorithms: Theory and realworld applications. Studies in Computational Intelligence **367** (2011)
4. Marx, V.: The big challenges of big data. Nature **498** (2013) 255–260
5. Wu, X., Zhu, X., Wu, G.Q., Ding, W.: Data mining with big data. IEEE Transactions on Knowledge and Data Engineering **26** (2014) 97–107
6. Ventura, S., Romero, C., Zafra, A., Delgado, J., Hervás, C.: JCLEC: a Java Framework for Evolutionary Computation. Soft Computing **12** (2007) 381–392
7. Cano, A., Luna, J.M., Zafra, A., Ventura, S.: A classification module for genetic programming algorithms in JCLEC. Journal of Machine Learning Research **16** (2015) 491–494
8. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation **6** (2002) 443–462
9. Gomez-Pulido, J., Vega-Rodriguez, M., Sanchez-Perez, J., Priem-Mendes, S., Carreira, V.: Accelerating floating-point fitness functions in evolutionary algorithms: A FPGA-CPU-GPU performance comparison. Genetic Programming and Evolvable Machines **12** (2011) 403–427
10. Salto, C., Luna, F., Alba, E.: Distributed evolutionary algorithms in heterogeneous environments. In: 8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing. (2013) 606–611
11. Apolloni, J., García-Nieto, J., Alba, E., Leguizamón, G.: Empirical evaluation of distributed differential evolution on standard benchmarks. Applied Mathematics and Computation **236** (2014) 351–366
12. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM **51** (2008) 107–113
13. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 26th IEEE Symposium on Mass Storage Systems and Technologies. (2010)
14. Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: 2nd USENIX Conference on Hot Topics in Cloud Computing. (2010) 10–10
15. Cano, A., Zafra, A., Ventura, S.: Speeding up the evaluation phase of GP classification algorithms on GPUs. Soft Computing **16** (2012) 187–202

16. Cano, A., Luna, J.M., Ventura, S.: High performance evaluation of evolutionary-mined association rules on GPUs. Journal of Supercomputing **66** (2013) 1438–1461
17. Cano, A., Zafra, A., Ventura, S.: Parallel evaluation of pittsburgh rule-based classifiers on GPUs. Neurocomputing **126** (2014) 45–57
18. Cano, A., Zafra, A., Ventura, S.: Speeding up multiple instance learning classification rules on GPUs. Knowledge and Information Systems **44** (2015) 127–145