Cloud Driven Design of a Distributed Genetic Programming Platform

Owen Derby, Kalyan Veeramachaneni, and Una-May O'Reilly

Massachusetts Institute of Technology, USA.
{ocderby,kalyan,unamay}@csail.mit.edu

Abstract. We describe how we design FlexGP, a distributed genetic programming (GP) system to efficiently run on the cloud. The system has a decentralized, fault-tolerant, cascading startup where nodes start to compute while more nodes are launched. It has a peer-to-peer neighbor discovery protocol which constructs a robust communication network across the nodes. Concurrent with neighbor discovery, each node launches a GP run differing in parameterization and training data from its neighbors. This factoring of parameters across learners produces many diverse models for use in ensemble learning.

Keywords: cloud computing, machine learning, genetic programming, distributed evolutionary computation.

1 Introduction

Recent availability of on-demand massive compute resources, i.e. *clouds*, has encouraged research into massive parallelization of machine learning (ML) systems (see *Mahout* and *GraphLab* [7,8]). Because clouds are different from other distributed resources like clusters or grids, they impose new requirements on how we parallelize ML, but also offer new opportunities[1]. In a cloud, the time to acquire many nodes is often quite long. To efficiently use a cloud, ML needs to start as soon as the first instance is acquired and not wait for the final instance's acquisition. When learning for an extended period of time, it needs to be designed as an online system where best interim results can be obtained at any time. It needs to expand or contract by taking advantage of cloud's elasticity to respond to varying resource costs and handle node failures.

These requirements drive innovation in both cloud-scale evolutionary computation (EC) and any machine learning enabled by EC. In this contribution we present FlexGP which is an elastic, cloud-scale, distributed platform using Genetic Programming for ML (GPML). FlexGP is a new version of previous work in [9]; we will refer to the old version as FlexGP-ECJ to distinguish between the two. We focus on GPML here, but any EA could be used with FlexGP. GPML is not well-suited for use with any of the existing cloud parallelization frameworks. Some of these frameworks, like MapReduce, were designed for completing large, single-run computations and do not readily

[1] These include the ability to learn many heterogeneous models very cheaply and the ability to learn for an extended period of time.

A.I. Esparcia-Alcázar et al. (Eds.): EvoApplications 2013, LNCS 7835, pp. 509–518, 2013.

accommodate the iterative nature of GPML. Others were retrofitted from grid or cluster architectures and thus do not fully realize the potential of the cloud. An example is FlexGP-ECJ which retrofitted grid-based EC software for use on the cloud. Instead, FlexGP has been intended, from its outset, to run on the cloud, and has been designed to take full advantage of the cloud.

We focus on the cloud-oriented design aspects of FlexGP. It is designed with an understanding that the cloud supplies sufficient computational resources upon request, yet expects these resources might fail or be delivered with unknown latency. It is conceived as a long-running computational learner, evolutionarily adapting and continuously improving its model, whilst allowing for drastic changes in supplied cloud resources and topology. It is implemented as a collaboration of many heterogeneous FlexGP instances, independently learning a model and observing the topology of the network. CloudGPML is implemented on a private cloud running OpenStack, a free and open source software suite providing Infrastructure as a Service (IaaS) for clouds. To integrate timestamps across nodes, we rely on Network Time Protocol (NTP), standard on Ubuntu 12.04, to provide accurate time synchronization of the nodes. This is sufficient for our purposes, as FlexGP operates on the scale of many seconds to minutes, and is not affected by microsecond variations.

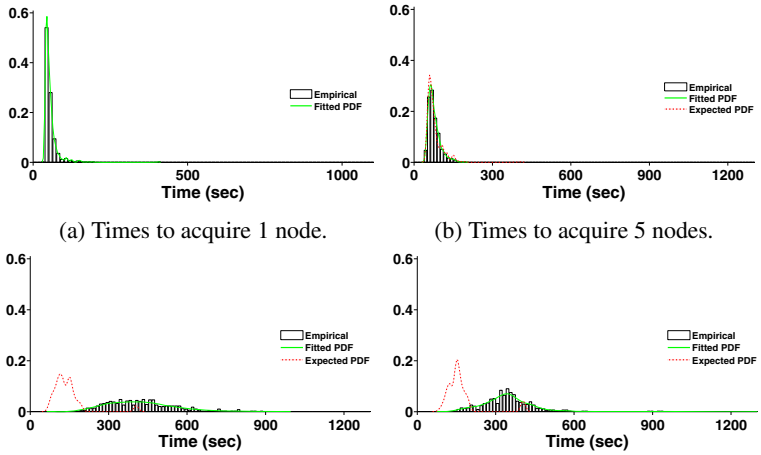
FlexGP-ECJ used a centralized master to coordinate starting GPML on nodes and establishing a network topology. This caused severe bottlenecks in starting the system and introduced vulnerabilities to node failures. In FlexGP, there is no single controller coordinating the system. It launches via a unique parallel asynchronous startup protocol. Integrated within this protocol is a stochastic factorization of the GPML parameters, creating a heterogeneous network of diverse learners. FlexGP runs a completely decentralized *gossip* neighbor discovery protocol at its IP layer. The protocol establishes the network simultaneously with the cascaded launch and integrates new instances into the network. It is resilient to instance failure and allows communication to continue even when instances disappear. Finally, it also enables interim results collection.

We proceed in the following manner: Sect. 2 describes the asynchronous launch procedure of FlexGP. Sect. 3 describes how each GPML learner is started with different data and parameters, ensuring a diverse set of models to support ensemble learning. Sect. 4 describes its IP discovery protocol. Recognizing that it is easier to compare FlexGP to other approaches after it has been described, a discussion of related work is deferred to Sect. 5. Sect. 6 concludes.

2 Parallel Asynchronous Startup

Applications typically request instances from a cloud in batches. The cloud possibly queues these batch requests and may decompose them; interleaving them with requests from other users. This might depend on batch size or the cloud's use of an internal fine-grained queue and a scheduler. Regardless of what a particular cloud does, the instance scheduler implementation should be treated as opaque by application designers.

In designing FlexGP's launch protocol, we started by studying the severity of latency in acquiring cloud instances. We assume that the time elapsed between requesting an instance and when that instance has booted and begins running our code, the *latency*,



(a) Times to acquire 1 node.

(b) Times to acquire 5 nodes.

(c) Times to acquire 50 nodes.

(d) Times to acquire 100 nodes.

Fig. 1. Probability distribution functions (PDF) of times to acquire nodes

is modeled by some distribution $P(u)$. We first estimated $P(u)$ by acquiring a single instance 1,000 times and measuring the latency, u, of each request. The data and its distribution are reported in Fig. 1a. If we optimistically assume a batch request of n instances is served in parallel as n independent requests by the scheduler, then the total latency, v_n, of the request ought to be the maximum of n independent samples drawn from $P(u)$. We estimated $P(v_n)$ for $n \in [5, 50, 100]$ 500 samples and then fit a non-parametric distribution to the data. We report the observed data and fitted distributions alongside the predicted distributions (based on our measured $P(u)$) in Fig. 1. While the predicted and empirical distributions for $P(v_5)$ are close, the actual latency distributions for $P(v_{50})$ and $P(v_{100})$ are significantly larger than predicted.

This discrepancy indicates that smaller batch requests achieve closer to optimal latency than larger requests, and so our system ought to emphasize small batch requests over large ones. Futher, because acquiring many (50 or 100) instances may take significantly longer than acquiring the first 10 instances, we should start running GPML on an instance immediately after it boots, long before the entire set of nodes is acquired. Another concern when computing using the cloud is failing nodes. Requested nodes may never be acquired and running nodes may fail. This necessitates an architecture which is resilient to failures.

FlexGP implements a decentralized, peer-to-peer (P2P) startup algorithm in light of these observations. Every FlexGP instance is capable of launching other FlexGP instances. Immediately after booting, every FlexGP instance retrieves parameters from the node which started it. The parameters $\Psi.k$ and $\Psi.p$ indicate the number of nodes to start and the target IP list size (see Sect. 4), respectively. The GPML meta-parameters, Π, are used to determine the parameterization of each GPML learner (see Sect. 3). These steps are detailed in the NODESTART function in Algorithm 1.

Figure 2 left illustrates how FlexGP would launch 7 instances when $\Psi.k = 2$. Node A is launched and runs NODESTART(7, []), where [] indicates an empty list. A then

Algorithm 1. NODESTART(n, R)

n: nodes to launch, R: list of ancestor IP addresses
Ψ: launch parameters, Π: GPML meta-parameters
$ip \leftarrow$ LAST(R)
RETRIEVE(ip, Ψ, Π)
$R \leftarrow$ CAT(R, MYIP())
$n \leftarrow n-1$
if $n \leq \Psi.k$ and $n \geq 1$ **then**
  for $i = 1$ to n **do**
    $c_i \leftarrow$ BOOTNODE(1, R)
else
  for $i = 1$ to $\Psi.k$ **do**
    $k \leftarrow \lfloor \frac{n}{\Psi.k-i+1} \rfloor$
    $c_i \leftarrow$ BOOTNODE(k, R)
    $n \leftarrow n-k$
IPDISCOVERY(R)
GPMLCOMPUTE()

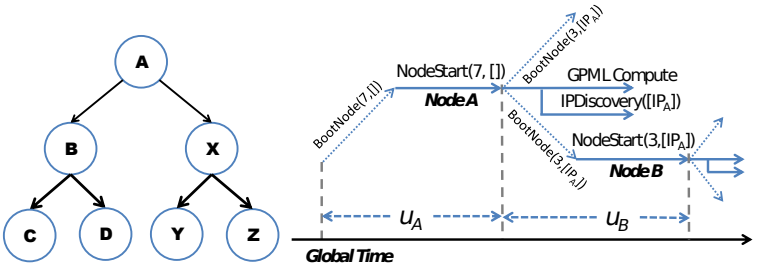


Fig. 2. A view of the launch of FlexGP for 7 nodes. Left: An initial node is launched and it brings up 2 more, which in turn bring up 2 more each, in a cascading fashion. Right: Timeline of booting and launching of instances. After starting more nodes, node A begins computation.

boots nodes B and X, each of which will run NODESTART(3, $[IP_A]$), and will go on to boot 2 more nodes each. Figure 2 right details the timeline of two nodes during startup, illustrating the concurrency present in the FlexGP startup. As soon as node A finishes executing NODESTART and started nodes B and X, it starts a new thread to begin running GPML computation and then continues into the IPDISCOVERY algorithm, as described in Sect. 4. This enables us to run GP concurrent with IP discovery and network discovery.

Figure 3 reports results from our experiments; demonstrating the advantages of the concurrent nature of FlexGP. The plot on the left shows the total progress of GPML as measured by individuals evaluated as global time progresses on the 150 nodes. The figure on the right examines the distribution of completed GP generations across all started nodes as the last node starts (around the 1800th second). The cumulative effect of this is that by the time the last node starts, some nodes have completed as many as 30 generations and some 2.2 million individuals have been evaluated across the FlexGP system.

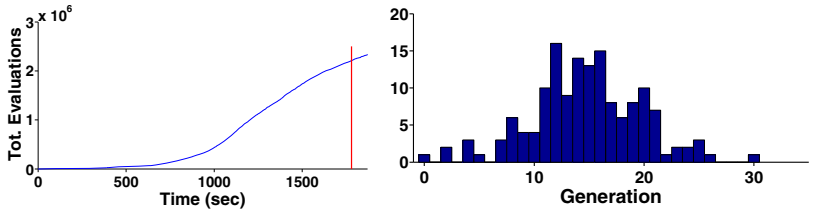


Fig. 3. Progress of GP on each node as distributed startup and IP discovery progress. Left: Cumulative fitness evaluations completed by all FlexGP nodes as launch proceeds. Right: Histogram of number of generations finished before the *time of last launch* (marked as red line on left figure).

Since each evaluation requires a pass through the training cases, this corresponds to at least 2.2 million passes through the problem dataset.

In Fig. 4 we compare how long it takes to start up 50 nodes for $\Psi.k \in [2,4,8,16]$. As expected, the time decreases as $\Psi.k$ increases, until $\Psi.k = 8$. Then the time gets worse for a value of 16. This is likely due to the wider variation in latencies for larger batch request sizes, as seen in Fig. 1. Note that the specific tradeoff point at $\Psi.k = 8$ is largely dependent on the properties of our cloud and the load it is under at the time of measurement and we expect this point would change over time or if measured on a different cloud.

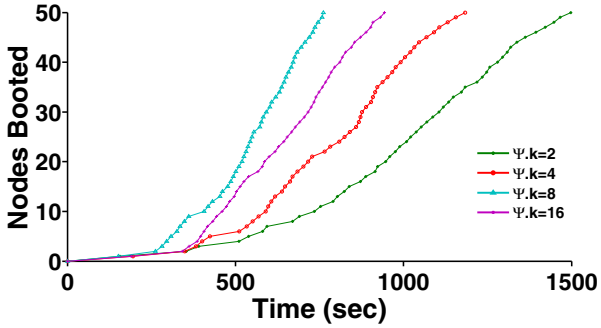


Fig. 4. Time to acquire 150 node as we change $\Psi.k$. Values reported are averages taken over 30 trials at each value.

The protocol is tolerant of node failures: the failure of one node interrupts the acquisition of further instances by that node, but does not hinder launches by other running nodes. For example, in Fig. 2, if node X failed to launch properly, nodes Y and Z will never be requested, but there is no affect on the acquisition of nodes B, C or D. In general, while the actual number of acquired nodes may not meet the requested N, GPML (and IP discovery) can execute on all nodes that have been acquired. We have taken the view that N will usually be large enough and failure will be sufficiently infrequent that we do not need to be concerned about any reporting, tracking and explicit recovery of node failures.

There may still be cases where the launch did not acquire a sufficient proportion of N instances. This may occur in the unlikely event that a node crashes very early on in the launch or in the face of intermittent cloud service interruptions. If such a scenario arises, we can simply tap an existing node and have it run the startup with new parameters which will try to populate the network with more resources. This same strategy can also be used to increase the number of running instances after startup. We might want to do this at night, when cloud instances become cheaper to run.

3 Factored Learners

The availability of massive on-demand expandable compute resources in the cloud enables us to learn many models in parallel. Bagging, boosting or simple parameter search are now feasible at a scale never seen before. FlexGP generates a large set of diverse models for ensemble learning. This is achieved by varying the parameters each GPML learner starts with. This leads to a set of factored learners, working in parallel to learn a diverse set of models.

We define the parameters for a GPML learner as $\{L, O, D, F\}$. L is the operator set provided to GP. O is the objective function used for normalizing fitness evaluations in GP. D is the set of training cases presented to the learner and F is the set of data features used in each training case in D. Note that while L and O are specific to GP, D and F are generic parameters of the data. Different options are available for these parameters, which are summarized in Table 1.

Table 1. GPML parameters and their possible values and definitions

Parameter	Value	Definition
Operator Set (L)	W	$\{+, -, /, *\}$
	X	$\{exp, ln\}$
	Y	$\{sqrt, x^2, x^3, x^4\}$
	Z	$\{sin, cos\}$
Objective Function (O)	Norm	Mean absolute error
	Norm-2	Mean squared error
	Norm-inf	Max error
Training Cases (D)	n	Subset of training cases, of size n
Feature Set (F)	m	Subset of features, of size m

We take Π, the set of meta-parameters retrieved from the parent (see Sect. 2), to define distributions over these options, guiding how FlexGP selects the values for each parameter. We will use the notation $p(O)$ to represent the distribution over the options for O. $p(L)$ gives probabilities for each of the 8 possible values of L.[2] $p(O)$ gives probabilities for each of the 3 *Norm*s defined. L and O are each chosen as a

[2] These values being W, $W \cup X$, $W \cup Y$, $W \cup Z$, $W \cup X \cup Y$, $W \cup X \cup Z$, $W \cup Y \cup Z$, and $W \cup X \cup Y \cup Z$.

single sample from $p(L)$ and $p(O)$, respectively. $p(D)$ defines probabilities of selecting each training case (for the set of all training cases). D is constructed by sampling without replacement n times from $p(D)$. The distribution for F is split into two parts. $p_1(F)$ gives probabilities for the number m of features to use. $p_2(F)$ defines the probabilities of using each feature. F is constructed by sampling m from $p_1(F)$ and then drawing m samples from $p_2(F)$ without replacement. For example, one learner could use the parameters $\{W, Norm2, \{x_1, x_2\}, \{d_{1...3000}\}\}$, while another learner might use $\{\{W \cup X\}, Norm2, \{x_1, x_4\}, \{d_{1...3000}\}\}$.

In our current implementation, we set all distributions in Π to be static, uniform distributions, for simplicity. However, since Π is retrieved from the parent node, the distributions over parameters may change dynamically as new nodes are launched. For example, a node could modify $p(D)$ to decrease the chance their children (the nodes it started) will select the same training cases it did. Or if new nodes are to be brought up after the system has been running, $p(D)$ could be modified to weight difficult training cases more and $p(F)$ could be modified to weight features with little apparent informative power less. We hope to explore such possibilities in the future.

4 Distributed IP Discovery

A key requirement of any cloud-based ML application is the support for communications between learners. Further, cloud-scale systems need an established network to robustly extract information and results from the system. For GPML, such communications might include the migration of individuals (models) or data-dependent summaries. Our work focuses on the establishment of such a network, but not on how the GPML learners use it.

The commonplace centralized architecture for network establishment (the so-called "master-slave" model) is not sufficient to meet the requirements of a cloud-based compute system [9]. In such an architecture, the nodes cannot begin computing until they receive parameters and IP lists from the master. This allows for the creation of arbitrary network topologies by the master. However, on a cloud the master cannot know the IP addresses until all instances are acquired. As we observed in Sect. 2, the latency for a many-node acquisition is quite large. Further, some of the requested nodes might fail before reporting to the master, complicating matters further.

To avoid this latency while still achieving the networking requirements of FlexGP, we design a distributed IP discovery protocol. Note, we focus here on the initial bootstrapping of the network - the "IP discovery" problem. This is separate from the problem of creating particular topologies in P2P networks [4]. Recall that as part of startup a parent node shares its IP list with all its children. A node at level i therefore has i IP addresses at startup. We then use a *gossip* protocol to populate the neighbor list at each node. First, we set a lower limit, $\Psi.p$, for the number of IP addresses a node needs to acquire. It generally is a function of the total number of nodes. We then follow an address passing protocol per Algorithm 2. In this protocol's active phase, each node selfishly tries to increase its IP addresses up to its limit by requesting more IP addresses from its neighbors while it shares with its neighbors its IP addresses in exchange. After it meets or exceeds the limit, in its passive phase, it serves any request it receives in exchange for their IP addresses.

Algorithm 2 IPDISCOVERY(R)

```
Λ ← R
loop
  λ ← set of new messages received
  for m in λ do
    if m.type is REQUESTIPLIST then
      Λ ← MERGE(Λ, m.Λ)
      RESPONDIPLIST(m.ip, Λ)
    else if m.type is RESPONDIPLIST then
      Λ ← MERGE(Λ, m.Λ)
  if LEN(Λ)< Ψ.p then
    ε ← RANDOM(Λ)
    REQUESTIPLIST(ε)
```

4.1 Empirical Study

We ran launch and IP discovery experiments with $N = 150$, and $\Psi.p = 25$. The plots of Fig. 5 show global time progressing along the x-axis. In Fig. 5 left, the y-axis denotes the number of IP addresses each node acquires. Each plotline is a node and, for clarity, we only show a subset of N. We observe that all nodes eventually acquire IP addresses of a significant number ($> \frac{N}{2}$) of nodes and in some cases almost all other nodes. A plotline changes from solid to dashed when a node switches from active to passive phase. It is interesting that the latter nodes acquire addresses of a very large number of nodes by gossiping with just one or two nodes arbitrarily. Interestingly, the last node acquires as many as 150 addresses. This ensures connectivity if we add more nodes later. Finally, note that all nodes have at least $\Psi.p$ nodes before the last one is booted.

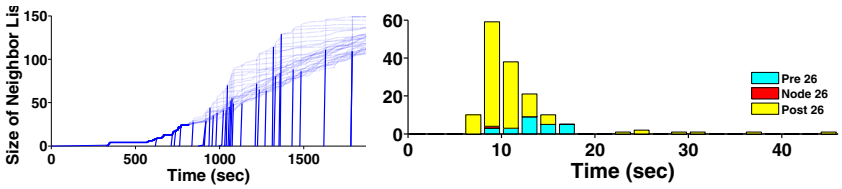


Fig. 5. IP discovery through gossip. Left: Progress of IP discovery as a function of global system time. Each line represents the number of IP addresses a node accumulates as time progresses. Right: Time it took for each node to acquire $\Psi.p$ IP addresses.

Figure 5 middle shows the distribution of delays for nodes to enter the passive phase of IP discovery. This delay for the i^{th} node is calculated as follows. Let S_i be the time at which the i^{th} node started and let T_i be the time at which the i^{th} node discovered its 25^{th} IP address. Then the latency for node i is given as $T_i - max(S_i, S_{26})$. We notice that almost 130 nodes take less than 25 seconds to enter the passive phase.

5 Related Work

There is a large body of distributed EC research which focuses exclusively on the design of distributed, algorithmic models, like island-based GP, instead of designs which take advantage of a particular resource type or communication layer. Much of this work is only tangentially related to FlexGP, as we are focused on writing an EC platform which takes advantage of the cloud platform. The systems in [1, 3, 10–12] rely upon MapReduce for parallelization. MapReduce is a powerful platform for distributed computation, but its dependence upon a separate distributed file system, single point of failure in the master and synchronization bottlenecks are not a good match with the cloud.

FlexGP's IP discovery is like other EC peer-to-peer systems. For example, the EvAg system [5, 6] also relies upon gossiping for node discovery. Little information is available on its startup method. It is not specialized to run on particular resource types whereas it is designed to investigate topology and a fine grained distribution model. EvAg and FlexGP differ in how they introduce evolutionary diversity: EvAg employs different operators across randomized neighbourhood whereas FlexGP factors each island with differentiation of data, GP objective function, operator set and input variables. Folino introduced peer to peer based design for building classifier ensembles [2].

6 Conclusions and Future Work

In this paper we have described the development of FlexGP, a large-scale, distributed system using GP for machine learning on the cloud. Our goal was to establish a large network of independent GPML leaners on the cloud with an eye towards minimizing latencies and bottlenecks while maximizing resource utilization. FlexGP is an improvement over its precursor FlexGP-ECJ, avoiding its bottlenecks in starting up and establishing a network topology. For a modest sized experiment of 150 nodes attempting to solve a small scale regression problem using FlexGP we showed the efficiency gains by introducing parallelization schemes and concurrency at multiple levels.

We plan to continue development on FlexGP, refining our platform and adding new features. The elegant design of this framework allows seamless expansion of the computation as needed. We intend to use this feature in several ways. If more resources become available, we would like to bias the distributions in Π for the new nodes based on insights from the running computation. We would also like to introduce multiple levels of parallelism to solve big data problems. Each FlexGP node can start a FlexGP system of its own, running an island-based GP with migration in a fully connected network. Additionally, we will develop shared memory, multicore parallelization at each GPML learner. This will allow us to run bigger population sizes and solve larger problems.

Acknowledgements. This work was supported by the GE Global Research center. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of General Electric Company.

References

1. Fazenda, P., McDermott, J., O'Reilly, U.-M.: A Library to Run Evolutionary Algorithms in the Cloud Using MapReduce. In: Di Chio, C., Agapitos, A., Cagnoni, S., Cotta, C., de Vega, F.F., Di Caro, G.A., Drechsler, R., Ekárt, A., Esparcia-Alcázar, A.I., Farooq, M., Langdon, W.B., Merelo-Guervós, J.J., Preuss, M., Richter, H., Silva, S., Simões, A., Squillero, G., Tarantino, E., Tettamanzi, A.G.B., Togelius, J., Urquhart, N., Uyar, A.Ş., Yannakakis, G.N. (eds.) EvoApplications 2012. LNCS, vol. 7248, pp. 416–425. Springer, Heidelberg (2012)
2. Folino, G., Forestiero, A., Spezzano, G.: A jxta based asynchronous peer-to-peer implementation of genetic programming. Journal of Software 1(2), 12–23 (2006)
3. Huang, D.W., Lin, J.: Scaling populations of a genetic algorithm for job shop scheduling problems using MapReduce. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 780–785 (December 2010)
4. Jelasity, M., Montresor, A., Babaoglu, O.: T-Man: Gossip-based fast overlay topology construction. Computer Networks 53(13), 2321–2339 (2009); Gossiping in Distributed Systems
5. Jiménez Laredo, J.L., Lombraña González, D., Fernández de Vega, F., García Arenas, M., Merelo Guervós, J.J.: A Peer-to-Peer Approach to Genetic Programming. In: Silva, S., Foster, J.A., Nicolau, M., Machado, P., Giacobini, M. (eds.) EuroGP 2011. LNCS, vol. 6621, pp. 108–117. Springer, Heidelberg (2011)
6. Laredo, J., Eiben, A., Steen, M., Merelo, J.: Evag: a scalable peer-to-peer evolutionary algorithm. Genetic Programming and Evolvable Machines 11, 227–246 (2010)
7. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: A framework for machine learning and data mining in the cloud. Proc. VLDB Endow. 5(8), 716–727 (2012)
8. Owen, S., Anil, R., Dunning, T., Friedman, E.: Mahout in Action. Manning Publications Co. (2011)
9. Sherry, D., Veeramachaneni, K., McDermott, J., O'Reilly, U.-M.: Flex-GP: Genetic Programming on the Cloud. In: Di Chio, C., Agapitos, A., Cagnoni, S., Cotta, C., de Vega, F.F., Di Caro, G.A., Drechsler, R., Ekárt, A., Esparcia-Alcázar, A.I., Farooq, M., Langdon, W.B., Merelo-Guervós, J.J., Preuss, M., Richter, H., Silva, S., Simões, A., Squillero, G., Tarantino, E., Tettamanzi, A.G.B., Togelius, J., Urquhart, N., Uyar, A.Ş., Yannakakis, G.N. (eds.) EvoApplications 2012. LNCS, vol. 7248, pp. 477–486. Springer, Heidelberg (2012)
10. Verma, A., Llora, X., Goldberg, D., Campbell, R.: Scaling genetic algorithms using mapreduce. In: Ninth International Conference on Intelligent Systems Design and Applications, ISDA 2009, pp. 13–18 (December 2009)
11. Verma, A., Llora, X., Venkataraman, S., Goldberg, D., Campbell, R.: Scaling eCGA model building via data-intensive computing. In: 2010 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (July 2010)
12. Wang, S., Gao, B.J., Wang, K., Lauw, H.W.: Parallel learning to rank for information retrieval. In: Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, pp. 1083–1084. ACM, New York (2011)