Flex-GP: Genetic Programming on the Cloud

Dylan Sherry, Kalyan Veeramachaneni,
James McDermott, and Una-May O'Reilly

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

Abstract. We describe Flex-GP, which we believe to be the first large-scale genetic programming cloud computing system. We took advantage of existing software and selected a socket-based, client-server architecture and an island-based distribution model. We developed core components required for deployment on Amazon's EC2. Scaling the system to hundreds of nodes presented several unexpected challenges and required the development of software for automatically managing deployment, reporting, and error handling. The system's performance was evaluated on two metrics, performance and speed, on a difficult symbolic regression problem. Our largest successful Flex-GP runs reached 350 nodes and taught us valuable lessons for the next phase of scaling.

1 Introduction

Cloud computing has emerged as a new paradigm for commercial, scientific, and engineering computation. A cloud allows an organization to own or rent efficient, pooled computer systems instead of acquiring multiple, isolated, large computer systems each commissioned and assigned to particular internal projects [1, 6].

Cloud computing has substantial advantages. First, it offers *elasticity*. Elasticity allows a software application to use as much computational resources as it needs, when it wants. A cloud also offers *redundancy*. If a server fails, a replacement is swiftly available from the resource pool. However it cannot be automatically incorporated into a long-running computation unless the computation is designed to allow this. A cloud also offers *higher utilization*. Utilization refers to the amount of time a pool of resources is in use rather than idle.

For evolutionary algorithms (EA) researchers, the cloud represents both a huge opportunity and a great challenge. Parallelization has been well-studied in the context of EAs, and has been shown to affect population dynamics and diversity, and to improve performance. With the cloud we can aim to run parallel evolutionary algorithms at a scale never before seen, but we must first make our algorithms and implementations cloud-ready.

In the long term we envisage novel refactoring and rethinking of genetic programming (GP) as the cornerstone of a massively scalable cloud-based evolutionary machine learning system. Our immediate goal is to design, implement, deploy and test a cloud-based GP system, which we call Flex-GP.

In this paper, we adopt an island based parallelization model with communication via sockets, and choose Amazon's EC2 as a computational substrate.

C. Di Chio et al. (Eds.): EvoApplications 2012, LNCS 7248, pp. 477–486, 2012.

We choose symbolic regression as an application. We start modestly with a few nodes and scale to tens and then hundreds of nodes. We encounter some unexpected challanges but achieve parallelization of GP up to 350 islands.

In Sect. 2, we begin with a discussion of related work on the parallelization and scaling of EAs. Sect. 3 briefly describes the elastic compute resource provided by Amazon. In Sect. 4 we present the strategies we employed to scale the algorithm and the challenges that arose. We then present a benchmark problem and experimental results in Sect. 5. We present our conclusions and future work in Sect. 6.

2 Related Work

The simplest EC **parallelization models** are the independent parallel runs model and the master-slave fitness evaluation model. Both are useful in some circumstances. Our research interest is in a different model, the *island model*. Multiple populations or islands run independently and asynchronously, with infrequent *migration* of good individuals between islands. The island model has been studied extensively, with surveys by Cantú-Paz [2] and Tomassini [8].

The topology of an island model may be visualised as a network of nodes (each representing a population) and directed edges (representing migration). Island models typically depend on a centralised algorithm to impose the desired neighbourhood structure. Decentralised algorithms have also been studied, where the network structure emerges in a peer-to-peer or bottom-up fashion [4].

Island models may be expected to deliver performance benefits over single-machine EC, due to their larger total populations. As demonstrated by Tomassini [8], island models can in fact do even better. This happens chiefly because a structured population can avoid premature convergence on just one area of the search space. Vanneschi [9] (p. 199) notes that for each problem there is a population size limit beyond which increases are not beneficial. Tomassini found that isolated populations have an advantage in performance over single large populations, where total population size is 2500, and that communicating islands have an advantage over multiple isolated ones.

A key opportunity in cloud computing is its **massive scale**. Most existing research in island model evolutionary algorithms has not used very large numbers of nodes. A typical value in previous experiments is between 5 and 10 nodes [8,9]. Although we note that each of these projects are now quite old, few specific node-counts are available in the recent literature. The most important exception is the 1000-node cluster used by Koza [7] (p. 95).

The *Hadoop* implementation [`http://hadoop.apache.org/`] of the *MapReduce* framework [3] has been used for genetic algorithms [10]. We started from *ECJ* [5] which is a EC system written in Java. It includes an island model framework which uses sockets for communication in a master-slave arrangement. ECJ is more flexible than MapReduce and can avoid its requirement for a master node which represents a single point of failure and a synchronisation bottleneck; it is actively maintained; and it offers an easy-to-use, but limited, island model. Therefore ECJ was selected for our work.

3 Deploying Flex-GP

We chose to use the Amazon Elastic Compute Cloud (EC2), a versatile cloud computing service. EC2 provides a simple abstraction: the user is granted as many instances (VMs) as needed, but no access to the underlying host machines. Instances can vary in size, with the smallest costing as little as $0.02 per hour. Users can request instances immediately, reserve instances for a set time period, or bid against the current spot price. An instance's software is specified by a VM image called an Amazon Machine Image (AMI). Amazon provides a selection of off-the-shelf AMIs and also allows the definition of custom images. We found the default Amazon Linux AMI to be a suitable platform for our system. We initially chose to use the cheapest instance size, *micro*, on an immediate-request basis. However, *micro* instances are intermittently allocated more processing power for short periods. To more clearly analyze the performance of our system, we transitioned to *small* sized instances, which are granted a fixed amount of processing power.

Table 1. Notations

Name	Notation
Island q	I_q
Number of Neighbors	N_n
Number of islands	Q
Number of instances	n
IP address of the node i	IP_i
Neighbor Destinations	N_{dn}
Time out for replacing instance	T_o

As described in Sect. 2, we chose to use ECJ's island model. It uses a client-server architecture, where each client hosts one island, and a sole server is responsible for setting up the topology, starting and halting computation. In ECJ's off-the-shelf island models, one of the clients doubles up as the server. We chose to configure a separate server with an eye to scaling. Algorithm 1 presents the pseudocode for the n-island model. Parameters like migration size, M_s, rate, F_m, start generation SG are provided by the user.

Each island consists of two Java threads, as shown in Fig. 1. The first is the main thread, which performs evolutionary computation and periodically sends packets of emigrants to neighboring islands. The second acts as a mailbox, and is responsible for receiving packets of incoming immigrants. The main thread periodically fetches newly arrived individuals from the mailbox and mixes them into the population. Note that this architecture implies that if a node crashes, those nodes to which it sends will not be affected, nor will those which send to it. Although the topology of the network will be damaged, all other nodes will continue calculations. This is a limited form of robustness.

Algorithm 1. The socket based n-island model

1. Pre-process: Create the necessary *params* files
2. **Initialization**
for $d = 0$ to n **do**
  Initialize I_d
  if $d = 0$ **then**
    $IP_s \leftarrow IP_d$
  end if
  if $d \neq 0$ **then**
    Recv_$I_d(IP_s)$
    Send_$I_0(IP_d)$
  end if
end for
3. **Set up communications**
if $d = 0$ **then**
  for $d = 1$ to n **do**
    Send neighborhood information: Send_$I_d(N_n, N_{dn}, N_{dn}^{id}, IP_{dn})$.
    Send Migration parameters: Send_$I_d(M_s, F_m, SG)$.
    Send GP parameters: Send_$I_d(psize, ngens)$.
  end for
end if
4. **Start computation**
if $d = 0$ **then**
  for $d = 1$ to n **do**
    Instructs I_d to start computation
  end for
end if
5. **Stopping computation**
if $d \neq 0$ **then**
  while I_d did not receive stop signal **do**
    Send_$I_0(O_i)$ where O_i is the fitness at the end of i^{th} generation.
  end while
end if
if $d = 0$ **then**
  Recv(O_i)
  if $O_i \geq O_d$ **then**
    for $d = 1$ to n **do**
      Send_I_d(*stop signal*)
    end for
  end if
end if
The server and all islands have exited

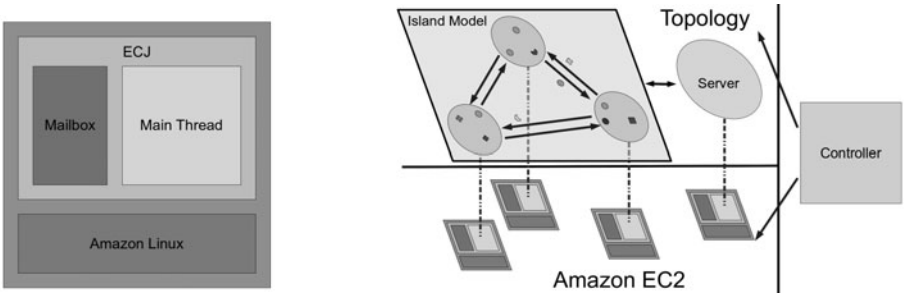

Fig. 1. An ECJ island consists of multiple processes (left). The three-island model on EC2 (right).

4 Scaling Flex-GP

As we proceeded from an initial run towards large numbers of islands, new issues emerged. We present our progress as a series of milestones: $Q = 3, Q = 20$, $Q = 100$ and finally $Q = 350$ islands. In this section our aim is only to consider scaling, and so the details of the problem and the fitness values are achieved are not reported. For completeness, we note that the experimental setup was the same as that in Sect. 5.

Milestone 1: Three Islands. Our initial goal was a proof of concept. We manually constructed a three-island ring topology on three EC2 instances, with the server hosted on a fourth. This enabled us to understand the steps involved in launching, starting and running a basic island-based GP system on EC2. We found that three key ECJ components were easy to use and ready to run on EC2: socket based communication, an evolutionary loop, and experiment setup. The three-island model ran successfully.

Milestone 2: 20 Islands. As soon as more than three islands were required, the overhead required to manually start each island as above became unachievable. We automated the instance requests using EC2's Python API, *boto*. We avoided the transfer of files by creating a custom AMI containing our code-base. With this setup we achieved the 20-island milestone.

Milestone 3: 100 Islands. During the next phase of scaling, several more issues became apparent.

Instance boot was both unreliable (about 1 in 250 requested nodes simply fails to start) and highly variable in the time required (from 15 seconds to several minutes). The time required for instance network connection was also variable, up to 30 seconds. Since the ECJ island computation does not begin until all islands have reported successful startup, it was therefore essential to provide **monitoring and dynamic control of instance startup**. In Algorithm 2 we present a dynamic launch monitor and control process. It has two parameters, the wait time α and the time for timeout, T_o. All EC2 interfaces have latency of a few seconds, so α

must be long enough to allow for that. We set T_o as the mean time required for an instance to launch and connect. Its main function is to make instance requests and take account of requests which are apparently failing.

Algorithm 2. Dynamic logic for instance startup

```
Generate a cloneable image C
d ← 0
while d ≤ n do
  Request an instance of C via Boto
  while T_o ≠ γ do
    if Instance is running according to API then
      if Instance is connected then
        d ← d + 1
        T_o ← γ
      else
        wait for α
        T_o ← T_o + α
      end if
    else
      wait for α
      T_o ← T_o + α
    end if
  end while
end while
```

Even after an instance is correctly created, a variety of problems can occur at runtime, including software and configuration bugs, network problems, and other unknown errors. **Error tolerance and reporting** is essential. To better handle debugging and post-run analysis we first coded these messages. Once we resorted to a coded catalogue/dictionary for errors, it enabled us to incorporate more log messages throughout our code-base. The coding of messages helped to reduce the bandwidth when transferring logs from the islands.

Milestone 4: 350 Islands. Amazon limits new EC2 users to 20 concurrent instances. Requests for increases may be placed and are usually fulfilled incrementally some days later. After several requests our limit stands at 400 instances. Our next goal was to approach this limit. The main questions to be considered were: would the socket-based model withstand communication among hundreds of islands? Would the fact that the server is a single point of failure prove problematic?

At this level, two major augmentations were required. We added an additional dedicated instance as a monitoring/log server. To do so, we added a *LogServer*, supported by the *Twisted* open source Python networking engine. The *LogServer*'s role was to aggregate and display information about the current status of computation across the network. Two types of information were transmitted to the *LogServer*: performance and migration tracking. In our larger tests

we inadvertently "stress tested" the capacity of this server. During benchmarks with 350 islands, the *LogServer* received and recorded almost 100,000 lines of text over several minutes, successfully receiving 100% of incoming messages.

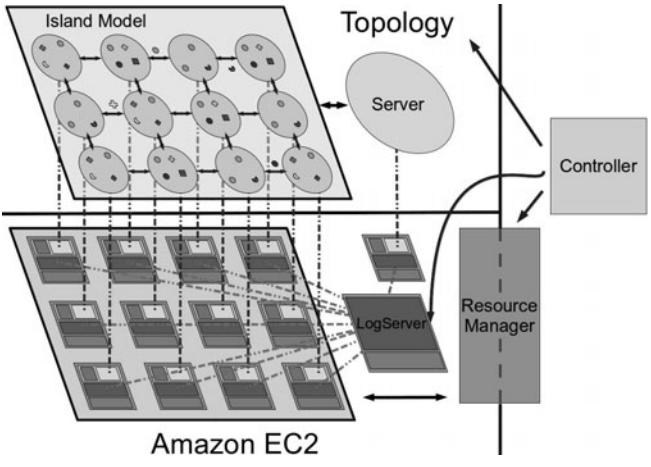

Fig. 2. The 350-Island System

We also wished to take advantage of our large limit of 400 instances by running multiple tests simultaneously, for example almost the entirety of the 1, 2, 4, 16, 128 and 256 island runs mentioned in the next section. We introduce the concept of a *bucket*, that is a set of nodes allocated to a single task. We added a *ResourceManager* to manage instances and buckets, consisting of the following components: a *Connection* class which communicates with EC2; a list of *free instances*; and a set of buckets to which the free instances can be allocated. Each test run requires one bucket. Managing multiple buckets in the ResourceManager means that multiple tests can be run simultaneously. The ResourceManager favoured the serial re-use of buckets to minimise setup time.

5 Experimental Setup

Our next goal was to evaluate our system. The benchmark was a two-variable symbolic regression problem taken from [11]. The aim is to match training data produced from a known target function,

$$f(x, y) = \frac{e^{-(x-1)^2}}{1.2 + (y - 2.5)^2}$$

The training data was 100 points randomly generated in the interval $(x, y) \in [0.3, 4]$, with a different set of points being generated for each island. No separate testing phase was run. Fitness was defined as the mean error over the

training points. The function set was $\{x, y, +, *, -, \%, \text{pow}, \exp, \text{square}\}$, where % indicates protected division (if the denominator is zero, 1 is returned).

The population was initialised using the ramped half-and-half algorithm, with minimum depth 2 and maximum depth 6. Tournament selection was used with tournament size 7. Subtree crossover, biased 90/10 in favour of internal nodes, was used with probability 0.9, and reproduction with probability 0.1. No mutation was used, nor was elitism. The maximum tree depth was 17. The population size at each island was 3000 and the number of generations was 100. The island topology used was a non-toroidal four-neighbor grid. Each island was configured to send 40 emigrants to its destination neighbors every four generations. We ran the benchmark on 1, 2, 4, 16, 64, 128 and 256 islands. For each of these cases we conducted at least 10 runs.We evaluated benchmark performance on two metrics, accuracy and time.

Accuracy is the best fitness achieved, i.e. $1/(1 + e)$ where e is the mean squared error on our benchmark problem. In Fig. 3 we show the improvement in accuracy of the system as a function of number of islands. We plot the average fitness achieved at the end of each generation. We average this number over 10 independent runs. This is shown in Figure 3(right). Fitness generally improves as we add more resources. However, as we add more resources the gain in fitness achieved at the end of 100 generations reduces. For example, the biggest gain in fitness is achieved when we go from 1-island model to 4-islands and the least gain is achieved when we use 256 instead of 128 islands. In the experiment only the number of islands was varied, and all other parameters left fixed. The larger trials thus had a lower *relative* degree of information flow between islands, which may have impaired performance. Finally it is possible that this result reflects Vanneschi's finding that for any problem, there is a limit to total population size beyond which performance does not increase and can even be impaired [9] (p. 199). We also show the variance in the best fitness at the hundred generations for multiple runs as a box plot in Figure 3(left). It is interesting to note that the variance in the best fitness achieved significantly reduces as we add more resources.

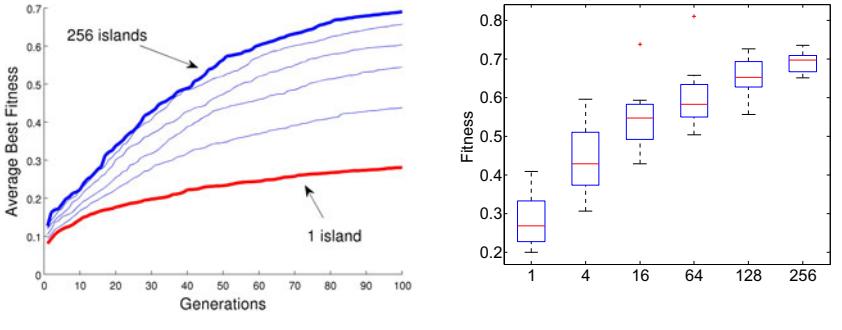

Fig. 3. Results achieved on a benchmark symbolic regression problem

Time is measured as achieving this higher accuracy in the same amount of time that would be used on a single machine. We measure both communications and infrastructure setup time and total compute time, with results as shown in Fig. 4. We plot four different times. First one is the initialization of the *LogServer*. The second is the time taken by our system to set up monitors, and the third is the time taken by the evolutionary server to set up islands and the communications. Both these times see an increase in time from $\tilde{1}0$ seconds to $\tilde{1}80$ seconds. Finally we show the actual computation time of the islands. Even though we add 255 nodes the compute time only increases by a factor of $\tilde{3}$.

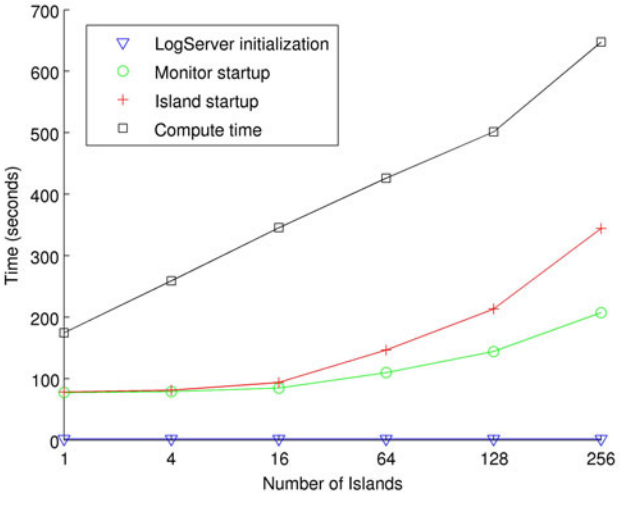

Fig. 4. Time taken by the server to set up island infrastructure and topology for communications

6 Conclusions and Future Work

In this paper we have described development of Flex-GP, which we believe to be the first large-scale cloud GP implementation. This is timely and is made possible by the advances made in virtualization and cloud infrastructure. We chose the Amazon EC2 cloud service. From several pre-existing software packages, we chose *ECJ* which provides a simple off-the-shelf island model with socket communication. We made each island a separate EC2 instance.

In order to scale up to 350 islands we had to develop many additional software features including *cloning*, *dynamic launch*, and a *LogServer*. We made use of some publicly available open source tools like *twisted*, *boto*, *nmap*. We encountered and overcame several problems during scaling. We were able to identify when certain features become critical: for example, automatic launch via *cloning* becomes necessary for island numbers above 5.

Our success is demonstrated through Flex-GP performance on a benchmark problem. Increased resources improve performance, with some cost in time.

Our next goal is to scale Flex-GP to at least 1000 islands. From experience in scaling to 350, we expect to require a number of additional infrastructural features such as *distributed startup*, *visualization tools* and *no single point of failure*. We also aim to modify the Flex-GP island model to introduce features of *elasticity* (add or remove instances when needed) *resiliency* (gracefully handle node failures). After achieving these we aim to examine other distribution models.

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. Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28 (2009)
2. Cantú-Paz, E.: A survey of parallel genetic algorithms. Calculateurs Paralleles 10(2) (1998)
3. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of ACM 51(1), 107–113 (2008)
4. Laredo, J.L.J., Castillo, P.A., Paechter, B., Mora, A.M., Alfaro-Cid, E., Esparcia-Alcázar, A.I., Merelo, J.J.: Empirical Validation of a Gossiping Communication Mechanism for Parallel EAs. In: Giacobini, M. (ed.) EvoWorkshops 2007. LNCS, vol. 4448, pp. 129–136. Springer, Heidelberg (2007)
5. Luke, S., Panait, L., Balan, G., Paus, S., Skolicki, Z., Bassett, J., Hubley, R., Chircop, A.: ECJ: A Java-based evolutionary computation research system (2007), `http://cs.gmu.edu/~eclab/projects/ecj/`
6. Ograph, B., Morgens, Y.: Cloud computing. Communications of the ACM 51(7) (2008)
7. Poli, R., Langdon, W., McPhee, N.: A field guide to genetic programming. Lulu Enterprises UK Ltd. (2008)
8. Tomassini, M.: Spatially structured evolutionary algorithms. Springer, Heidelberg (2005)
9. Vanneschi, L.: Theory and Practice for Efficient Genetic Programming. Ph.D. thesis, Université de Lausanne (2004)
10. Verma, A., Llora, X., Goldberg, D.E., Campbell, R.H.: Scaling genetic algorithms using MapReduce. In: Proceedings of Intelligent Systems Design and Applications, pp. 13–18 (2009)
11. Vladislavleva, E., Smits, G., Den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via Pareto genetic programming. IEEE Transactions on Evolutionary Computation 13(2), 333–349 (2009)