Chapter 1
Maintenance of a Long Running Distributed Genetic Programming System For Solving Problems Requiring Big Data

Babak Hodjat, Erik Hemberg, Hormoz Shahrzad and Una-May O'Reilly

Abstract W

e describe a system, ECStar, that outstrips many scaling aspects of extant genetic programming systems. One instance in the domain of financial strategies has executed for extended durations (months to years) on nodes distributed around the globe. ECStar system instances are almost never stopped and restarted, though they are resource elastic. Instead they are interactively redirected to different parts of the problem space and updated with up-to-date learning. Their non-reproducibility (i.e. single "play of the tape" process) due to their complexity makes them similar to real biological systems. In this contribution we focus upon how ECStar and introduces a provocative, important, new paradigm for GP by its the sheer size and complexity. ECStar's scale, volunteer compute nodes and distributed hub-and-spoke design have implications on how a multi-node instance is managed. We describe the set up, deployment, operation and update of an instance of such a large, distributed and long running system. Moreover, we outline how ECStar is designed to allow manual guidance and re-alignment of its evolutionary search trajectory.

Key words: genetic programming, learning classifier system, cloud scale, distributed, big data

1.1 Introduction

Despite executing a variant of island-based evolution with migration, ECStar, previously described in O'Reilly et al (2012); Hemberg et al (2013b); Hodjat and Shahrzad (2012), is a genetic programming (GP) system which is remarkably different from other distributed evolutionary algorithms (EA). First, it demonstrates a number of aspects of scaling which outstrip those of extant GP or EA systems. For

Genetic Finance, CA 94105 USA. and ALFA Group, CSAIL, MIT and Genetic Finance, CA 94105 USA. and ALFA Group, CSAIL, MIT

example, one ECStar system instance learning financial trading strategies executes on nodes which are spatially distributed across the globe and another instance of ECStar is learning forecasting rules in the medical domain. Further, a widely deployed ECStar system instance is very seldom stopped and restarted. This avoids incurring the overhead of deploying and retracting its many compute nodes. It is designed to be able to simultaneously work on different sub-parts of a problem and over time it synthetically integrates partial solutions. Simultaneously, it is able to shift to new aspects of the problem by accepting, without interruption, new learning data or parameter settings.

Another way of contextualizing ECStar is to compare it to artificial life (ALIFE) systems Bedau (2003). Both share an aspect of longevity in that they are run without a goal of termination. Both share the nature that a snapshot of current state, at any time in the extended duration of their execution, is insightful. ALIFE systems are studied for their life-like behavior in a digital setting (i.e. they offer explanations of a biological process or system). In contrast, ECStar is executing toward solving a problem. It can be polled at anytime, at its Evolutionary Coordinator, to provide its latest best evolved solution. In some commonality with ALIFE systems, there is a continuous interactive relationship between ECStar and its developers. A developer is able to "tinker" with the evolutionary direction of the system. The interactions are indirect (e.g. changes to resources, modifications in parameters) and intentionally light handed: they nudge rather than interfere. This guidance is motivated by problem solving, whereas in ALIFE triggering reactions to perturbations are often the goal.

One further analogy is helpful, ECStar's developer's goal and means of interaction, from the perspective of making a system evolve toward something, places the developer in the role of a bench scientist using directed evolution to isolate a biological property of interest in a cell line. The bench scientist aims to coax evolution to biologically generate a collection of cells with a specific property. A starting cell population is allowed to proliferate and evolve, then, iteratively it is filtered (and perhaps divided for concurrent evolution) to focus its trajectory with an end point in mind. Evolution accomplishes the work of search and adaptation with in-vivo selection while the bench scientist adds an external selection pressure to the process.

In this contribution we augment existing descriptions of ECStar which focus on its distributed design and application, see O'Reilly et al (2012); Hemberg et al (2013b); Hodjat and Shahrzad (2012). We comment on the way in which ECStar's scale, resource choice of nodes offering idle cycles and distributed hub-and-spoke design have implications on how a very large, multi-node instance is managed. We describe the entirely new proposition of setting up, deploying, operating, monitoring, harvesting, securing and even software updating an instance of such a large, distributed and long running system. Broadly, we describe how it is possible to minimize the installation of new Evolutionary Engines while enabling wide ranging, tunable behavior that a developer can oversee and control. We describe how ECStar is designed to allow manual guidance and re-alignment of its evolutionary "trajectory" through an assortment of mechanisms.

We proceed as follows. In Section 1.2 we compare ECStar in terms of distributed model, size and use of volunteer compute node idle cycles to related work. Section 1.3 provides a description of ECStar sufficient to visit its design motivation. In Section 1.4 the long running and large scale system features are explained. The operation and direction of ECStar are explained in Section 1.5. In Section 1.6 there is a discussion of how ECStar influences the GP experimental paradigm. Finally, a summary and future work are in Section 1.7.

1.2 Previous work

There are other distributed GP systems documented but none has the same architecture, size, execution duration or particular use of volunteer nodes' idle cycles as ECStar.

In terms of architecture, typically distribution follows two basic models:

- Master-slave where the fitness function evaluation is distributed and a single server executes the main evolutionary algorithm
- Islands with migration where a number of independent EA algorithms each execute on a node, and exchange best solutions regularly (though often asynchronously) using a fixed neighbor topology Cantu-Paz (2000); Tomassini (2005); Crainic and Toulouse (2010); Scheibenpflug et al (2012).

ECStar and just a few others use the model where a pool of individuals are coordinated centrally while a set of client nodes evolve them and pass their best to the central server. Merelo et al (2012) report an initial exploration examining pool versus island based models. Their algorithm IslandSofEA uses separate clients and a pool and scales best of their tested methods.

In terms of size, i.e. number of nodes deployed, a dated but valuable description of larger GP systems and how to build a parallel system can be found in Bennett III et al (1999). This source reports how to build a parallel computer system for $18,000 (1999 prices) that performs a half peta-flop per day. It employed 10 nodes with 533 MHz Alpha processors. Langdon (2012) talks about running GP on the Emerald GPU super computer which has 1008 x86 CPU cores and 372 nVidia M2090 Tesla (In total Emerald has 190,464 stream processors), providing an average of $33GPopS^{-}1$. An instance of ECStar has been run on more nodes than either of these two systems. However it is difficult to quantify a comparison because ECStar uses idle cycles of many different types of CPUs. ECStar's Evolutionary Engine software has not been ported to execute on a GPU because, at the time, GPU computing offers too small a fraction of its resource pool. A GPU, because it is SIMD, is best deployed to execute a lot of data simultaneously. A good strategy is for the GP individual to be evaluated after compilation or with a compiled evaluator downloaded into the GPU with its fitness evaluation cases distributed across the GPU. Empirical design questions as to the breakpoints where this strategy works for ECStar would have to be resolved first. ECStar does not exclude the possibility that individuals from Evolutionary Engines with GPUs can be reported to the Evolutionary Coordinator. One other evolutionary algorithm system also volunteer based has a large number of hosts, Desell et al (2010) execute a distributed genetic algorithm for the MilkyWay@home project which, at time of this article being written, lists $316,902$ hosts (see http://boincstats.com/).

In terms of systems that employ volunteer resources, the ways in which they are used is diverse. Fernández de Vega et al (2012) present a customizable execution environments for evolutionary computation using BOINC (Anderson (2004)) plus virtualization, by running VMWare in the BOINC wrapper. The computing model is focused on institutions, which own their computers, and an end-user application that administers the BOINC resources of an institution and allows existing EAs to run multiple executions. The scale is smaller than ECStar and the system is used for conventional EA runs, with no mention of long running evolution or any description of how to maintain it. Smaoui and Garbey (2013) study how to improve volunteer computing scheduling for evolutionary algorithms. They only use the volunteer compute node clients for computation of fitness function, not for running independent EAs per the master-slave model.

Desell et al (2010), who run a distributed genetic algorithm for the MilkyWay@home project, also use clients for fitness evaluation in the master-slave model. Fault tolerance in a master-slave environment for evolutionary algorithms is discussed in Gonzalez et al (2012). The authors find the parallel EAs to be robust to faults, i.e. loss of fitness evaluations on the slave nodes. In ECStar the Evolutionary Engine executes an EA and sends solutions to the Evolutionary Coordinator, thus the fault tolerance considers churn of Evolutionary Engines instead of only single fitness evaluations.

We next provide an overview of the ECStar system architecture.

1.3 ECStar System Architecture

ECStar is a distributed EC system, see Figure 1.1, that employs a modified decision list representation Rivest (1987). It uses a large number of volunteer compute nodes as "Evolutionary Engines". The individual solutions from the Evolutionary Engines are coordinated by an Evolutionary Coordinator, and data is distributed to an Evolutionary Engine by a Data Server.

1.3.1 Evolutionary Engine

Each Evolutionary Engine runs a completely independent evolutionary algorithm in its client's spare cycles. It has a fixed population size and initially generates the population randomly. In the evolutionary loop, it requests fitness cases (training data) from the fitness case server in the form of a *data package*. Each individual in

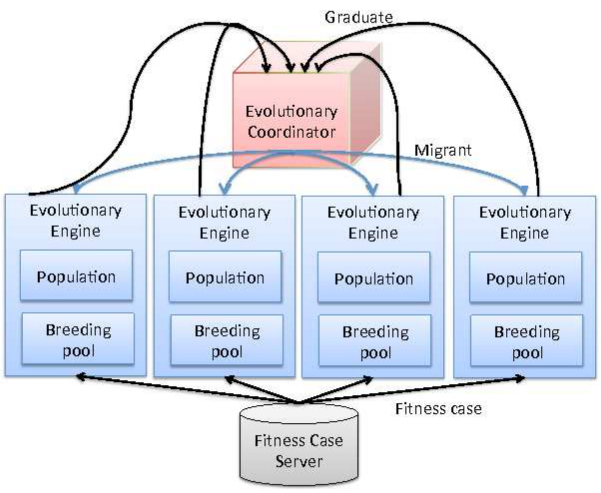

Figure 1.1 The hub and spoke model of ECStar.

the population is evaluated, and after a fixed number of data packages, selection and breeding take place before replacement and the next generation. Periodically local individuals become graduates and are dispatched to the Evolutionary Coordinator and *migrants* are received from the Evolutionary Coordinator.

Evolutionary Engines are unpredictably available with an unknown work and communication channel capacity. These constraints (deriving from the volunteer "contract") mean that Evolutionary Engines are designed to occupy a modest CPU, RAM, storage and network footprint, plus devote added computational time to security. Hodjat and Shahrzad (2012); Hemberg et al (2013b,a) describes in detail ECStar's age layering model for training data access and usage.

1.3.2 Evolutionary Engine Representation

An individual in ECStar is represented as a set of conjunctive rules. Operating on logical predicates based on expressing conditions of tests on the problem domain features gives the ECStar solutions the advantage of being human readable..

An individual (a.k.a *classifier*) has a header with id, age, fitness and a body with a set of rules. Each rule is a variable length conjunctive set of conditions with an associated action which is a class in a classification problem. Each condition acts as a propositional variable, which is then applied to the discretized real-valued training environment. Apart from conjunction operators each condition can have, a *comple-*

ment operator, negating the truth value and a *lag* which refers to “past" values of an attribute.

```
<rules> ::= <rule> | <rule> <rules>
<rule> ::= <conditions> => <action>
<conditions> ::= <condition> | <condition> & <conditions>
<action> ::= prediction label
<condition> ::= <predicate> | !<condition>  | <condition> [lag]
<predicate> ::= truth value on a feature indicator
```

Each individual in an Evolutionary Engine is evaluated on a number of fitness cases every generation. Each fitness case is in a data package and consists of a number of rows, called events. For each event, the variables in the classifiers rules' conditions are bound to the features of the event. For each rule in the individual, the rule's conditions are evaluated. If all are true, the rule is added to a *Match Set*. Finally, a voting mechanism elects from the *Match Set* a single rule's action as a prediction for the fitness case. When no rule fires, no prediction is made. We track *activity* and use is as a basis for selection into the breeding population. Predictions for all events in a fitness case are verified against the true class labels. A correct prediction increases the fitness and incorrect decreases. Each individual records its fitness in two ways: relative fitness and absolute fitness. (Relative fitness is the fitness per data package evaluated, i.e. fitness normalization)

1.3.3 Evolutionary Coordinator

The Evolutionary Coordinator coordinates migration among the Evolutionary Engines. It maintains a layered *archive* – a sorted set of individuals that are currently the best from all Evolutionary Engines. When a *migrant* query is received, *migrants* are randomly picked from its archive and sent to the requesting Evolutionary Engine. Migration serves two purposes: it allows individuals to be sent out to each Evolutionary Engine in order to be evaluated on more fitness cases (because Evolutionary Engines appraise an individual's fitness on only a fraction of the fitness cases before selecting graduates) and it allows the arriving migrants to mix their genetic material with the host Evolutionary Engine's local population. When a *migrant* message is received, the returning individuals compete for the space in the archive if it is full. The comparisons will be asymmetric, since different individuals have been evaluated on different data packages but the layers maintain individuals evaluated on approximately the same number of data packages to minimize noisy comparison. More details of archive management are mentioned in Section 1.4.2.5.

1.4 ECStar Design Rationale

We split our discussion of the reasons behind ECStar's design choices into two subsections. In Section 1.4.1 we discuss how general properties of volunteer computing and using ECStar's hub and spoke model impose requirements on its distributed evolutionary algorithm and these requirements, in turn, lead to its design choices. In Section 1.4.2 we next discuss how requirements imposed by the sheer large size and running time of ECStar have prompted design choices.

1.4.1 Idle Cycles, EA Requirements and Design Choices

Cost is arguably a primary driver for using idle cycles. Using volunteer compute nodes makes an ECStar system instance inexpensive relative to owning equivalent hardware capacity, or contracting cloud services. The decision to use idle cycles leads to a number of requirements that, in turn, lead to ECStar's design decisions. In this section, we connect the consequences to the requirements then to the design decisions. We accomplish this in a space efficient manner by cross referencing Tables 1.1 to 1.3. The Properties table, Table 1.1, numerically itemized, enumerates properties associated with using large quantities of volunteer compute nodes in a hub-and-spoke model with a many to one relationship between a volunteer client (a.k.a node) and a dedicated server. Recall that in the ECStar system, servers support Evolutionary Coordinators and Data Servers. The Requirements table, Table 1.2, alphabetically itemized, presents design requirements of an evolutionary algorithm system which ensue from various properties in the Properties table. It cross references requirements to properties using the Property table's item numbers. Finally, Table 1.3, the Design Choice table, enumerated with roman numerals, cross references design choices to multiple requirements in the Requirements table.

Table 1.1 Properties associated with using many volunteer compute nodes.

1. Large scale geographic distribution of nodes implies, at a macro-scale, that resource availability follows diurnal cycles across time zones. At a micro-scale, volunteer clients are available unpredictably.
2. RAM and permanent storage at each client (a.k.a Evolutionary Engine) are small.
3. Clients are not allowed to communicate with each other to ensure privacy of each volunteer client.
4. Extended time is required to enlist, deploy or shut down large quantities of clients.
5. Communication channels are insecure and noisy and their capacity is modest.
6. In many cases, clients should run while the host machines are in use.

Table 1.2 EA Requirements imposed by Volunteer Compute Nodes' Properties. () indicates row in Table 1.1

A. The EA must handle irregular compute availability, be able to cope with clients suddenly becoming ready or unavailable and be able to cope with asynchronous data communication. (1,4)
B. The Evolutionary Engine footprint must be small. (2)
C. The EA must be able to incorporate new islands while executing. It must continue to execute and utilize available nodes while not blocking to wait for unavailable others. (3)
D. Migration of information between EA islands (Evolutionary Engines in ECStar) must occur without the need for island to island identification. (1,5)
E. The EA must robustly run with many islands exchanging information.
F. The EA must not rely on running iterated experiments. (4)
G. The EA, while still using all fitness cases, must only reference a few at a time and all transactions with the servers should be lightweight. (5)
H. The EA should minimize its CPU (and memory) utilization so as not to disrupt host machine's main processing priorities. (6)

Table 1.3 ECStar's Design Choices following from its Requirements. () indicates row in Table 1.2

I. The Evolutionary Engines, coded in C, occupy a small footprint. Their stats can be quickly saved and restored. (B)
II. The Evolutionary Engines save their state frequently, they obtain fitness cases from the fitness data server in small quantities and execute the entire population on them before requesting more. (A,B)
III. There is no sequence related bottleneck in the Evolutionary Coordinator logic. It never blocks which allows it to always respond to any Evolutionary Engine graduate or migrant report. If it is busy, it harmlessly rejects the report and the reporter backs off before retrying.(A).
IV. The Evolutionary Coordinator uses a relative fitness metric when comparing individuals during archive management to address the fact that individuals are evaluated at different speeds. It also uses archive layering to impose a (reasonably) fair comparison between migrants in terms of fitness case exposure. (A).
V. The Evolutionary Engines are booted with configuration information that informs them of the Evolutionary Coordinator's and Data Server's IPs. They use this information to integrate with the current system. They do not need IPs of the other Evolutionary Engines because they communicate with them through the Evolutionary Coordinator. (C,D).
VI. Random migration is directed by the Evolutionary Coordinator. (C,D)
VII. Distributed, random fitness case evaluation defines interim solutions in terms of partial fitness as well as evolutionary progress. Neither the Evolutionary Engine or Evolutionary Coordinator have to maintain exact knowledge of which fitness cases an individual has been evaluated. (E)
VIII. No restart is required to run parallel experiments and they are integrated using federated resources. (F)
IX. The Evolutionary Engines use age layering and an elite pool to handle migrant guests of different ages. This prevents a migrant from being eliminated from providing genetic material too soon. (D)
X. Evolutionary Engines run at lower priority than host processes. The Evolutionary Engine can be stopped, paused, or restarted, and state files can be deleted from a container runner (GFRunner) or remotely by the server.

1.4.2 Scale Related Design Rationale

We now discuss the features which are key in allowing ECStar to execute continuously at a very large scale.

1.4.2.1 Resources federation

The Evolutionary Coordinator is a source of two potential bottlenecks. The archive may become too large to manage if there are a lot of Evolutionary Engines and the bandwidth of the channels into the Evolutionary Coordinator used by the Evolutionary Engines may become over-subscribed. While it would be possible to distribute the Evolutionary Coordinator itself to address load balance and archive management, this solution would create another bottleneck due to numerous reads and writes to the distributed archive.

Instead, ECStar *federates* the functions of the Evolutionary Coordinator, see Figure 1.2. An Evolutionary Coordinator can be down-chain to another Evolutionary Coordinator, acting as an Evolutionary Engine minus the evolutionary process itself. Down-chain Evolutionary Coordinators control and interact as Evolutionary Coordinators with their down-chains, which can be Evolutionary Engines, or Evolutionary Coordinators themselves, but they act as Evolutionary Engines to their up-chains. Each down-chain Evolutionary Coordinator maintains its own local archive, reducing the load on the top Evolutionary Coordinator's archive, as well as reducing bandwidth requirements. Note that Evolutionary Engines and down-chains of Evolutionary Coordinators need to exclusively communicate with their assigned up-chains. This is because the archives are federated themselves, and material received from these archives need to be reported back to them once aged, and would be missing from other archives.

1.4.2.2 Evolutionary Engine state preservation with state migration

A large computation executing over a long duration needs to be able to preserve its state if interrupted. This state must be quick to restore (and even modify before a restart). It needs to withstand a reboot or even prolonged shut-down of its host. While results are inevitably lost in these circumstances, they should be minimized. In this respect, ECStar can serialize and save the Evolutionary Engine population and the whole state of the evolution at intervals that are specified in an Evolutionary Engine's configuration in what is called the *state file*. Further, this state can be remote from the Evolutionary Engine, a local state-server, making it *stateless*. This makes the large, long computation is robust to lost nodes. When a Evolutionary Engine comes online for the first time, or if it is ordered to restart by its up chain Evolutionary Coordinator, it can request a state file from the local state-server and continue processing from that state onwards. This exploitation of a state-server also provides more management control of the Evolutionary Engines, especially for adding or removing them.

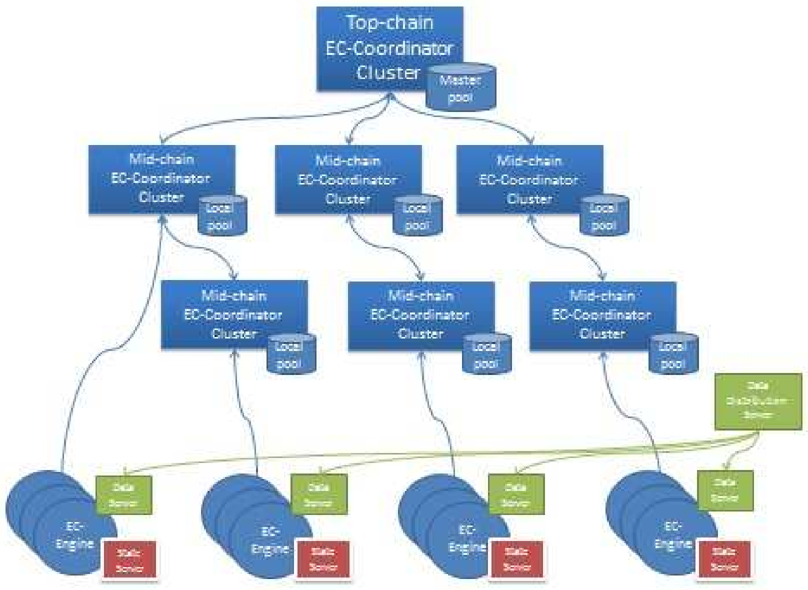

Figure 1.2 Resource federation in ECStar.

1.4.2.3 Unique IDs and Snapshots

Recall that the Evolutionary Coordinator cannot send an individual to only one Evolutionary Engine in case that Evolutionary Engine never responds (it is a volunteer after all). Therefore, it sends a copy of an individual while keeping it in the archive in order to maintain selection pressure (more details on this follow in Section 1.4.2.3). Scaling introduces the issue of tracking these replicated individuals as they return in an unpredictable order. To address this, the Evolutionary Coordinator, for each individual, creates unique Evolutionary Engine identifiers and prefixes them to unique identifiers assigned by the original Evolutionary Engine. This makes all individuals distinguishable both locally and globally. When an Evolutionary Engine receives a migrant, it creates a copy called a *snapshot*. Then after evaluating the original individual on a number of fitness cases, it reports back both the individual with updated fitness and the snapshot to the Evolutionary Coordinator. This allows the Evolutionary Coordinator to easily recognize and consolidate the differences between the two with its own current version of the individual. As a result, when ten Evolutionary Engines evaluate a gene on ten fitness cases, the Evolutionary Coordinator merges those reports so that the individual's fitness is based on the hundred cases.

1.4.2.4 Fitness-age normalization and layering

No ECStar Evolutionary Engine fully evaluates an individual on all the fitness cases. This partial evaluation implies a need to compare individuals that more or less match in terms of being evaluated on the same number of fitness cases. This is addressed through age-layering, see Hodjat and Shahrzad (2012); Hemberg et al (2013b). Layering (at either Evolutionary Engine or Evolutionary Coordinator allows individuals to only compete with others of the same level of fitness case evaluation, based on their relative fitness.

1.4.2.5 Archive Selection Pressure Through Alive/Dead State

It is important for the Evolutionary Coordinator to direct selection pressure among the individuals it receives. From Hemberg et al (2013a) we know that the coverage of the training fitness cases and the symmetry of the training fitness cases of individuals due to the evaluation of random training fitness cases increases as the individual solution is exposed to more cases. Parameterization supports changing the size of the layers on the Evolutionary Coordinator, to tune the pressure and address the issue of asymmetric comparisons (noisy) between individual solutions that have seen only a few training fitness cases. These parameters can be adjusted as ECStar executes.

Individuals that are dispatched by the Evolutionary Coordinator as migrants to Evolutionary Engines are not deleted from its archive. Retaining them maintains selection pressure in the context of returning migrants. However, these individuals may be displaced by new migrants with superior fitness. This creates a timing glitch: when the migrants of an individual return, it might have been displaced. The fitness of the displaced individual must be updated and it must be allowed to compete for entry into the archive. To resolve the glitch, displaced individuals are retained but marked as “dead” and called “ghosts”. Ghosts are kept in the archive until the ratio of “dead” genes is too high, then the archive is flushed of them.

1.4.2.6 Application Specific Interpreter in Evolutionary Engines

Each Evolutionary Engine is deployed with a very small, application specific, language interpreter. This allows instructions to be passed, via a configuration message passed by the Evolutionary Coordinator, to be executed and avoids needing to replace the software on the Evolutionary Engine. It makes the Evolutionary Engine to be a general executable which in turn is 100% configurable by the Evolutionary Coordinator via the configuration messages. The interpreter supports signals to restart, flush the Evolutionary Engine’s population, or stop before a backward incompatible software upgrade is loaded. A configuration message can also carry:

- new evolutionary algorithms settings, such as mutation rate, population size, maturity age, etc.

- a meta description of a new fitness objective
- a definition of new discrete intervals for feature discrimination
- a definition of new features in data packages

1.4.2.7 Integrated Data Feature Server

Because it executes uninterrupted for a long time, ECStar incorporates the ability to perform online feature selection. This function is controlled by the Data Feature Server.

Two feature sets are managed by a data base table which counts the number of handshakes (migrants reported to the Evolutionary Coordinator to be added or merged) which were requested and the number of handshakes which were fulfilled. The Data Feature Server looks at these ratios, an absolute threshold and improvement in fitness. When the improvement rate declines and the number of requested handshakes is more than C_1, the feature sets are ready to merge and a new set will be created. The merged feature set uses the union of features of both sets. The individuals are merged in the Evolutionary Coordinator and marked as coming from the merged set. The new set uses a fix collection of features and chooses others randomly. Merging and new feature set creation is coordinated via a properties file referenced by the Evolutionary Coordinator.

The Evolutionary Engines receive the definitions of the new feature set from the Evolutionary Coordinator via a configuration message. This minimizes the effort spent on updating the Evolutionary Engine. Configuration messages propagate through the normal Evolutionary Engine Evolutionary Coordinator communication and take a while to be completely reach all Evolutionary Engines.

1.4.2.8 Security

An ECStar system instance is running on public networks and the integrity of Evolutionary Engines is not assured. Ensuring security increases the computational cost of data transfers but is vital. Detecting injection of erroneous solutions is handled by consulting the identifiers on solutions and by executing out-of-sample tests. Sensitive data is encrypted and serialized. The Evolutionary Coordinator, Data Server and client-state servers are also required to reside behind firewalls.

1.4.2.9 Summary

Some of these scaling design choices function below the level of the attention of the engineer supervising an ECStar system instance. Others are exposed via parameterization or for provisioning. This becomes clear in the next section on System Instance Management.

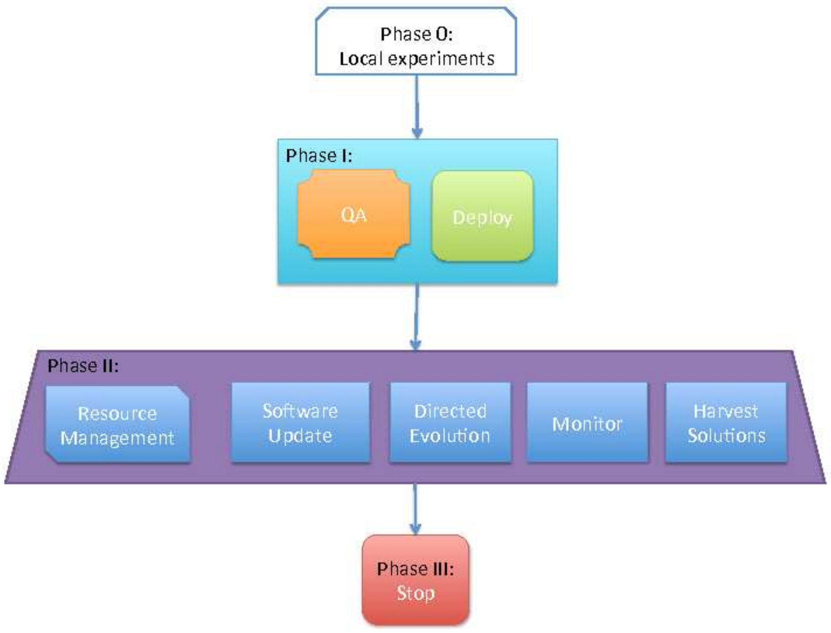

Figure 1.3 ECStar System Management Phases

1.5 ECStar System Instance Management

For an example of a use case for a deployment and maintenance of a run with 10,000 Evolutionary Engines, see Figure 1.3. ECStar executes over three phases:

Phase 0: Local experiments The initial step:

Local experiments On a cluster local experiments are completed before in preparation to scale. This includes iterating to decide upon an efficient format for data packages, features, representation parameters, objective function and execution parameters in Evolutionary Engine and Evolutionary Coordinator, see O'Reilly et al (2012). The output of the process are source and configurations for the Evolutionary Coordinator and Evolutionary Engine.

Phase I: Large-scale deployment preparation There are two preparation steps:

Quality Assurance The binaries for the Evolutionary Engines must pass a Quality Assurance (QA) checking whether it is able to run within the limitations of the client's footprint, and not disrupt the volunteer while it is not idle. This includes 24-48h execution and profiling of the binary.

Deployment After the QA the deployment is initiated to an controlled environment of 1000 Evolutionary Engines for a couple of days. The Evolutionary Coordinator servers and Data Servers are configured to handle the initial Evolutionary Engine load. Finally, the Evolutionary Engine code is distributed and deployed globally. This can take a week. The management of resources continues during Phase II.

Phase II: Long-term operation As soon as an Evolutionary Engine has received the binary code and has idle cycles it can start to compute. When the network capacity allows, it can request training cases from the data server. After enough time processing with idle cycles it tries to report results as snapshots of individual solutions to the Evolutionary Coordinator server. If the network capacity allows the Evolutionary Coordinator receives them. When there is high throughput the Evolutionary Coordinator is usually configured to tolerate a wave of 10000 requests per minute. There are a number of steps:

- Software update In order to update the Evolutionary Engine a new binary or parameter settings would need to be distributed. This is discouraged since it could take a week. Moreover, when updates are done on the servers it must be taken into account that there are Evolutionary Engines which are using data from previous versions, thus the turn around for updates is the same as for a new deployment. See Section 1.5.2 for more details on how this is nonetheless possible.
- Resources management By adding or removing Evolutionary Engines the resources can be managed during the run.
- Monitor We are continuously monitoring the state of the hardware and the network as well as the solutions at the evolutionary coordinator.
- Harvesting The harvesting is performed by filtering the solutions at the Evolutionary Coordinator or out-of-sample tests. See Section 1.5.2 for more details.
- Directed Evolution The design of ECStar allows a directed evolution of the system. Promising combination within the system can be allowed more resources, while worse can be terminated.

Phase III: Stop The final step:

- Stop Halting the run takes a while as well, a couple of days. It is important to make sure that all the Evolutionary Engines stop sending solutions to the Evolutionary Coordinator.

1.5.1 Monitoring a Deployed ECStar System Instance

When ECStar is up and running there are multiple automatic monitors in action. There is basic IT monitoring of data base load and connections. This mainly preventive and can identify where the Evolutionary Engines have slowed down.

Even more important is the monitoring of the Evolutionary Coordinators. The improvement of fitness is constantly monitored to warn when it trails off. In addition, individuals are constantly filtered and harvested from the Evolutionary Coordinator and tested on out-of-sample data for generalization. The convergence of the clients are monitored by looking at the ratio of number of handshakes between

Table 1.4 Components of ECStar and their access and maintenance

Component	Access	Maintenance
Evolutionary Engine	Unpredictable	Sporadic & Delayed
Evolutionary Coordinator	Predictable	Continuous & “hot swap”
Data Server	Predictable	Continuous & “hot swap”
Data Feature Server	Predictable	Continuous & “hot swap”

the Evolutionary Engine and the Evolutionary Coordinator and the accepted handshakes (individuals) at the Evolutionary Coordinator. The fitness function is used to drive the evolution, but there can also be other domain specific properties in the individual which are interesting. The monitoring can be configured to find individuals with these properties in the Evolutionary Coordinator archive as well.

1.5.2 Updating a Deployed ECStar System Instance

The state of a Evolutionary Coordinators, Evolutionary Engine, Data Server and Data Feature Server are: inactive (Off) and active (On). When a component is active the versions of all the components need to be compatible. Changing the active version of the components can be done interactively during runtime or with a restart. If the component is restarted, a new parameter file or a new binary can be used. The runtime changes require the fewest steps, but they will still result in the system being in a state with hetereogenous versions.

Table 1.4 shows the accessibility and the mode of maintenance of each component. Each of the components in ECStar has a configuration file. This reduces the need to recompile the components and instead a restart of the component is sufficient to make changes.

The Evolutionary Coordinator, Data Server and Data Feature Server are the components which are the most accessible, since they are currently not part of the volunteer compute. When there are changes made that requires feedback from the Evolutionary Engines then there will always be the turn around of all the clients being available and updated. The simplest change is when it is pushed out via the servers without stopping the run. It becomes slightly more complex when the clients need to be stopped and updated.

ECStar’s design deals with the appearance of Evolutionary Engines after long periods of idleness. This situation is handled by the check for version comparability between Evolutionary Engine and Evolutionary Coordinator when the Evolutionary Engine restarts. When incompatible Evolutionary Engines contact a Evolutionary Coordinator they are terminated by a kill signal sent from the Evolutionary Coordinator.

The federation of Evolutionary Coordinators allows the archive size to be scaled according to the number of Evolutionary Engines. In addition, federation allows a tiered evolution where different tiers control the selection pressure and supply of individuals. Finally, the federation allows more control for ECStar, since the Evolutionary Coordinators are under direct control.

The servers can be "hot swapped" by simply redirecting the IP of the server to a duplicate, while the original server is updated. The Evolutionary Engines are not so easy to change, since they cannot be accessed at will, but there is access to the state server. The loss of the Evolutionary Engine local data is therefore limited to the state file save interval.

The multiple objectives in the fitness function can be altered when ECStar is deployed. This has to be done with care because of the interaction of the old population under the new fitness objectives. It could wipe out the old population with no genetic legacy left behind or it may allow the population to converge to local minima.

1.6 Directed Evolution

The properties which imply ECStar is impressive and exemplary by contemporary measures of scale and computational evolutionary complexity also make ECStar a square peg in the evolved round hole of GP research expectations. This is because it is impractically prohibitive to repeatedly execute ECStar at least 30 times to obtain performance characterization which carries statistical significance. Like nature, ECStar won't pass a t-test; its scale, inherent asynchronicity and complexity dictate that its"tape" can only be played once.

Instead, a directed evolution approach can be taken when studying ECStar. This involves changing parameter settings during the run, adding and removing Evolutionary Engines and updating the software. The reset is never complete as with a standard GP system, it is more of a nudge of the current population. This requires either the population to be sufficiently diverse, or the ability to inject diversity, see Hemberg et al (2013a).

In addition, due to the robust setup ECStar allows investigation of what happens in the system when a change is not yet propagated to all the parts of the system, i.e. the components are hetereogenous. In other words, there will be intermediate effects of the system heterogeneity.

1.7 Summary

In summary, we believe it inevitable that more and more GP (and EC) systems will become distributed on much larger scale than is presently the case. This transition will be driven by operational costs of clouds and grids decreasing plus increasingly bigger applications with training data from so called "Big Data" repositories. This move from somewhat small-scale distribution in the order of ten's or hundred's to thousands through to millions is around 3 orders of magnitude in terms of computing nodes. ECStar addresses this scale up in the following ways:

- It has a hierarchical federated resources design which creates a hierarchy of Evolutionary Coordinators to handle the scale of the volunteer compute node and uses the Evolutionary Engine as the base. (Section 1.4.2.1)
- It explicitly deals with the inevitable and higher risk, introduced by running so many nodes, of losing computed information by means of Evolutionary Engine state preservation, state migration and a strategy of polling for required data in frequent but small quantities. (Section 1.4.2.2)
- It has an explicit means of resolving asynchronously computed work on the same datum (i.e. rule set) that occurs because of the unpredictable availability of its Evolutionary Engines by employing snapshots and unique IDs. (Section 1.4.2.3)
- It uses fitness normalization and layering to deal with the different types and value of fitness cases that have been evaluated by each solution. (Section 1.4.2.4)
- It maintains archive selection pressure by means of ghosts in the archive and flushing periodically flushing. (Section 1.4.2.5)
- It integrates a meta language to simplify deployment to the Evolutionary Engines. (Section 1.4.2.6)
- it incorporates the ability to perform feature selection on the data. (Section 1.4.2.7)
- It deals with security via encoding data, compiled binaries, firewalls and recalculation of fitness at the Evolutionary Coordinator, as well as out-of-sample evaluation of the top solutions. (Section 1.4.2.8)

ECStar also introduces explicit operational means of addressing its execution complexity which arises from using so many resources in an asynchronous way the time cost to deploy so many resources the need to update deployed resources. It has facilities for: monitoring, see Section 1.5.2, harvesting, see Section 1.5.1 and software updating, see Section 1.5.2.

ECStar offers a digital version of directed evolution. It has strong commonality with Artificial Life systems because, like them, it is a single execution process, rather than a restart-rerun platform. It is NOT an ALIFE system because it is applied to solving a problem, for purposes much more typical of evolutionary computation. It is applied and GP in practice rather than a digital experiment being studied for its behavior to validate and explicate a biological system.

We embrace a new "zen" of design with this unique evolutionary system. ECStar allows us to try to do direct evolution in a system which you do not completely turn off.

Acknowledgements

We acknowledge the generous support of Li Ka-Shing Foundation.

References

Anderson D (2004) BOINC: a system for public-resource computing and storage. In: Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on, pp 4 – 10, DOI 10.1109/GRID.2004.14

Bedau MA (2003) Artificial life: organization, adaptation and complexity from the bottom up. Trends in cognitive sciences 7(11):505–512

Bennett III FH, Koza JR, Shipman J, Stiffelman O (1999) Building a parallel computer system for $18,000 that performs a half peta-flop per day. In: Proceedings of the Genetic and Evolutionary Computation Conference, vol 2, pp 1484–1490

Cantu-Paz E (2000) Efficient and accurate parallel genetic algorithms, vol 1. Kluwer Academic Pub

Crainic TG, Toulouse M (2010) Parallel meta-heuristics. In: Handbook of Metaheuristics, Springer, pp 497–541

Desell T, Anderson DP, Magdon-Ismail M, Newberg H, Szymanski B, Varela CA (2010) An analysis of massively distributed evolutionary algorithms. In: 2010 IEEE world congress on computational intelligence, Barcelona, pp 18–23

Gonzalez DL, Laredo JLJ, Vega FF, Guervas JJM (2012) Characterizing fault-tolerance in evolutionary algorithms. In: Fernandez de Vega F, Hidalgo Perez JI, Lanchares J (eds) Parallel Architectures and Bioinspired Algorithms, Studies in Computational Intelligence, vol 415, Springer Berlin Heidelberg, pp 77–99, DOI 10.1007/978-3-642-28789-3_4, URL http://dx.doi.org/10.1007/978-3-642-28789-3_4

Hemberg E, Wagy M, Dernoncourt F, Veeramachaneni K, O'Reilly UM (2013a) Efficient training set use for blood pressure prediction in a large scale learning classifier system. In: Sixteenth International Workshop on Learning Classifiers Systems, ACM, New York, NY, USA

Hemberg E, Wagy M, Dernoncourt F, Veeramachaneni K, O'Reilly UM (2013b) Imprecise selection and fitness approximation in a large-scale evolutionary rule based system for blood pressure prediction. In: Proceedings of the fifthteenth international conference on Genetic and evolutionary computation conference - GBML, ACM, New York, NY, USA, GECCO '13

Hodjat B, Shahrzad H (2012) Introducing an age-varying fitness estimation function. In: Genetic Programming Theory and Practice X, Springer

Langdon WB (2012) Distilling genechips with gp on the emerald gpu supercomputer. ACM SIGEVOlution 6(1):16–22

Merelo J, Mora A, Fernandes C, Esparcia-Alcazar AI, Laredo JL (2012) Pool vs. island based evolutionary algorithms: an initial exploration. In: P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on, IEEE, pp 19–24

O'Reilly UM, Wagy M, Hodjat B (2012) Ec-star: A massive-scale, hub and spoke, distributed genetic programming system. In: Genetic Programming Theory and Practice X, Springer

Rivest R (1987) Learning decision lists. Machine learning 2(3):229–246

Scheibenpflug A, Wagner S, Kronberger G, Affenzeller M (2012) Heuristiclab hive-an open source environment for parallel and distributed execution of heuristic optimization algorithms. In: 1st Australian Conference on the Applications of Systems Engineering ACASE'12, p 63

Smaoui M, Garbey M (2013) Improving volunteer computing scheduling for evolutionary algorithms. Future Generation Computer Systems 29(1):1–14

Tomassini M (2005) Spatially structured evolutionary algorithms. Springer

Fernández de Vega F, Olague G, Trujillo L, Lombraña González D (2012) Customizable execution environments for evolutionary computation using boinc+ virtualization. Natural Computing pp 1–15