Design and implementation of evolutionary computation algorithms for volunteer compute networks
Evolutionary Design and Optimization Group, CSAIL
Otitochi Mbagwu

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Signature redacted

Department of Electrical Engineering and Computer Science
May 23, 2014

Certified By:

Signature redacted

Una-May O'Reilly, Thesis Supervisor
May 23, 2014

Certified By:

Signature redacted

Erik Hemberg, Thesis Co-Supervisor
May 23, 2014

Accepted By:

Signature redacted

Prof. Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee

Copyright by MIT. All rights reserved.
Abstract

We implemented a distributed evolutionary computation system titled EvoGPJ Star (EGS) and deployed the system onto Boinc, a volunteer computing network (VCN).

Evolutionary computation is computationally expensive and VCN allows more cost-effective cluster computing since resources are donated. In addition, we believe that the design similarities between EGS and our chosen VCN (Boinc) would allow for easy integration of the two systems.

EGS follows a centralized design pattern, with multiple engines communicating with a central coordinator and case server. The coordinator synchronizes up engines to run experiments and also stores and distributes individual solutions among engines. The engine-coordinator model creates a scalable (engines can be easily added) and robust (can continue to operate if nodes fail) system.

For our experiment we chose rule-based classification. We saw the distributed EGS solutions (standard and Boinc) outperform the single-engine system. Deploying the system to Boinc revealed some design conflicts between Boinc and EGS experimentation. These conflicts stemmed from the asynchronous and asymmetric nature of VCNs.
1 Introduction

The number of problems involving large data sets has been increasing. These problems require scalable algorithms that can utilize large clusters of resources. One scalable algorithm is Evolutionary Computation (EC). A common and powerful evolutionary computation method is genetic programming. Genetic programming (Batenkov, 2010) works by generating a group of random solutions at the start, called individuals. Each individual contains a tree-like representation of its solution containing operators at each node. The program then generates new individuals by mutating or crossing over individuals in the population (by swapping branches of their trees or changing operators at nodes). The individuals are then evaluated on training points and assigned a fitness based on the results. The best individuals are used to breed the next generation and the process repeats. The algorithm terminates after a specified amount of generations has passed, or the best solution’s margin of error has dropped below a specified threshold (indicating a ‘good enough’ answer has been found).

In general, evolutionary computation is a resource intensive process. One way to overcome this hurdle is to use a volunteer compute network (VCN). In a volunteer compute network individual users donate resources. In this paper I describe and assess a new distributed genetic programming system, titled EvoGPJ-Star (EGS), designed to be deployed onto a VCN. This system is similar in principle to ALFA’s evolutionary computation framework EC-Star (O’Reilly, 2012). Unlike EC-Star however, EGS is more flexible and agile (open source) and thus is able to accommodate a more diverse range of problems.

There are several motivations for deploying EGS on to a volunteer compute network (VCN). Firstly, we compare performance of evolutionary algorithms on VCN to a traditional cluster setup. In addition, in the process of building such a system we also expose any possible issues or design constraints related to deploying a distributed evolutionary compute system to a VCN. Secondly, the design of EGS lends itself to deployment on a VCN as both are centralized distributed systems with many engines communicating to a central server. Finally, evolutionary computation algorithms are quite expensive in terms of computer resources. Being able to leverage the compute power of a VCN when performing evolutionary algorithm can allow large-scale computation with lower costs.

Using VCN imposes some constraints which cluster or cloud computing do not need to follow. The VCN is asynchronous and unreliable (nodes can jump in and out of the network) and thus requires a robust design of the EC algorithm. In this paper I answer the questions of how to design and implement an EC algorithm efficiently (Section 2) for VCN and assess its performance (Section 3). In addition I investigate the requirements that the constraints of VCN have on EC experiments and runs (Section 4) and provide concluding remarks and possible future work (Section 5).
2 EvoGPJ-Star Design Overview
The newly constructed EGS system is composed of three main components: the evolutionary engines, the case server, and the evolutionary coordinator. The high level design is very similar to EC-Star (O’Reilly, 2012). The goal of communication between the evolutionary coordinator server, case server, and the evolutionary engines is to decrease the chance of converging to local maxima by generating more diverse solutions, increase the speed in which solution spaces are explored, to generate solutions that are robust (perform well on many training cases), and increase the number of training cases seen by good solutions (Hemberg, 2013).

Figure 1: Basic structure of the distributed EvoGPJ-Star program.
The case server stores training data for a specified problem. The evolutionary coordinator server stores and distributes individuals generated by the evolutionary engines. In the current implementation the evolutionary coordinator and case server are Apache Tomcat servers. The server logic is implemented in a java application that is run on the servers. The evolutionary engines run separate instances of an evolutionary algorithm. This point is crucial with respect to VCNs, as the failure or termination of an engine does not hinder the completion of the algorithm (the coordinator holds the best performing solutions). In a VCN such events are almost guaranteed since nodes can enter or exit the network at any time.

During the execution of their evolutionary algorithms the evolutionary engines will periodically request cases from the case server, request migrants from the evolutionary coordinator server, and submit graduates to the evolutionary coordinator server. See Figure 1 for a visual representation.

2.1 Engines

2.1.1 Case Requests

The purpose of case requests is to enable solutions to be evaluated on numerous training cases. A training case is a series of data points from the training set. In the case of symbolic regression it is a sample of points from the target function. We hope that using casing requests will reduce over-fitting on training cases and increase diversity (Hemberg, 2013). After each generation has passed the evolutionary engine is able to query the case server to obtain new training data. In EGS, the frequency of this query is determined by the parameter CASE_INTERVAL. The units of this parameter are defined as iterations of the evolutionary algorithm. By exchanging out cases periodically we avoid generating populations that are over-fitted to a specific case. The query is performed via an http request to the java service running on the case server.

In addition, by splitting up the training sets into arbitrarily small cases, we allow scalability. The training set can be very large, but the case size can still be quite small. Thus the time spent acquiring test cases can be independent of the size of the training set and we only need to worry about how much of the available training cases the solutions should be exposed to. This is detailed further in Section 3.1.
2.1.2 Solution Submissions

Solution submissions allow solutions to migrate from one evolutionary engine to another through the evolutionary coordinator. This should increase the diversity of solutions. Before transitioning to the next generation of the evolutionary algorithm, each evolutionary engine has the opportunity to submit individuals (graduates) to the evolutionary coordinator. The frequency of this submission is determined by the `SUBMIT_INTERVAL` parameter. The units of this parameter are defined as iterations of the evolutionary algorithm. The program can also alter the number of individuals per submission via the `SOL_PER_SUBMIT` parameter. Additionally, another parameter `NO_COPY` was implemented that determines whether the original individuals or copies of them are submitted.
2.1.3 Solution Requests

It follows that evolutionary engines can also request individuals (migrants) from the evolutionary coordinator as well. This should increase diversity by ingestion of foreign solutions into the population. The frequency of this request is determined by the parameter `REQ_INTERVAL`. The units of this parameter are defined as iterations of the evolutionary algorithm. Just as with submissions, evolutionary engines can request multiple individuals, the number being controlled by the `SOL_PER_REQUEST` parameter. Again the request is done via http, with the evolutionary coordinator selecting individuals uniformly at random from graduates currently residing in the evolutionary coordinator. Solution requesting and submitting allows the best individuals from each evolutionary engine to migrate from one evolutionary engine to another, hopefully increasing the range of the solution spaces explored by each evolutionary engine.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE_REQUEST</td>
<td>Boolean value that enables case requests to case server</td>
<td>Evolutionary engine</td>
</tr>
<tr>
<td>SOLUTION_SUBMIT</td>
<td>Boolean value that enables solution submission to the evolutionary coordinator</td>
<td>Evolutionary engine</td>
</tr>
<tr>
<td>SOLUTION_REQUEST</td>
<td>Boolean value that enables solution requests to the evolutionary coordinator</td>
<td>Evolutionary engine</td>
</tr>
<tr>
<td>CASE_INTERVAL</td>
<td>Number of generations between case requests</td>
<td>Evolutionary engine</td>
</tr>
<tr>
<td>SUBMIT_INTERVAL</td>
<td>Number of generations between solution submissions</td>
<td>Evolutionary engine</td>
</tr>
</tbody>
</table>
2.1.4 Aging and Fitness

In the original EvoGPJ program, the fitness of an individual is determined by its last evaluation. However, this may not always be an accurate measure of how ‘good’ a solution is. A high fitness reading is only indicative of how well the individual estimates the latest training case. Even though each training case contains a sizeable number of training points (20 in the experiment trials) that are randomly generated (thus individuals that perform well on one are likely to perform well on others), we can still get ‘unlucky’ and get an unrepresentative fitness rating.

In the EGS program, each individual in the population also has an age, which is equivalent to the number of training cases the individual has been evaluated on. Individuals also contain a total fitness attribute that is the total fitness accumulated over all evaluations.

By using the age and total fitness properties of individuals we can now keep track of a running average fitness computed by calculating:

\[ AVGFITNESS = \frac{total\ fitness}{age} \]
This measurement combined with an individual’s age gives a measure of how robust a solution is (high age and high average fitness indicates an accurate solution). It’s also a useful metric for comparing solutions that have seen varying amounts of test cases. Finally, aging allows a population filtering mechanism, called age layering.

2.2 The Coordinator
The coordinator accepts graduates from the engines and stores them. The coordinator distributes solutions to engines upon request. The coordinator is a limited size, and will replace lower performing individuals with higher performing individuals when the population size limit has been reached. In essence the coordinator acts as an archive of the highest performing solutions. The coordinator is not located within the VCN and is under complete control of the practitioner.

2.2.1 Synchronization of EC Trials
The coordinator is also responsible for coordination of engines during experiment runs. It ensures that all engines run the same trial at the same time. It does this by waiting for a ‘ready’ signal from all the engines. After they have all been received it will clear the coordinator of all stored solutions and send a ‘start’ signal to all the engines, prompting them to start the next trial of the algorithm. Clearing the solutions is important, as you do not want generated solutions of one trial ‘contaminating’ the solutions in another trial. In this design the engines are unaware of the other engines existence and only act upon communication with coordinator via the coordinator’s IP address.

2.3 The Case Server
The case server responds to case requests from the engines. The case server will return a test case uniformly at random from all test cases upon request. The case server essentially sub-samples the entire training set and distributes to the engines. This component is important from a design standpoint as the engines do not need to store any training data locally, making the system more scalable. The case server java service running on a tomcat server and communications are done via HTTP.

2.5 Age Layering
In each step of the evolutionary algorithm with aging enabled, individuals are competing against each other, and the individuals with the highest average fitness move to the next generation. However, such a direct method of competition poses an issue when comparing young individuals to old individuals. It’s possible that a young individual got ‘lucky’, and was able to perform well on a small number of training cases but actually isn’t an accurate solution. Such an individual may push out an older individual (with slightly lower fitness that performs more accurately on a wider set of cases) from the population. In an Age Layered population, the individuals are grouped and filtered by age before being crossed over and mutated. Thus, the best individuals in each age group are used to generate the next population. Traits from all age groups are present throughout the newly generated
population. Age Layering can occur within engines or at the coordinator. When age layering occurs in the coordinator, the coordinator protocol should allow solutions from all age groups to be sent out to the engines.

2.6 EvoGPj-Star on Boinc Design Overview
The Boinc volunteer compute platform (Anderson, 2004) is extremely similar to the ‘star’ design of EGS. In a Boinc project a central server sends out work-units to connected engines that perform work and report back to the server.

![Diagram of EGS with Boinc](image)

*Figure 5: EGS with Boinc. The Boinc server distributes the application to the engines to be run*
Unfortunately, Boinc's application API is written in C, and EGS is entirely in java. Thus implementing Boinc distribution and reporting features directly into the program was not possible. Instead we used Boinc as solely a distribution platform, delivering the engine code to participating machines that would then run the algorithms. Reporting of data and program completion is handled through the EGS network infrastructure instead of Boinc.

3 EvoGPj-Star Performance and Tests

Our experiments had several goals. First we want to test that fundamental assumptions of the EGS system were correct and producing the hypothesized results. These assumptions pertained to case coverage and case overlap. Case coverage measures how well the populations are exposed to the training set. Case overlap measures how many cases are shared among the individuals in the population. We hoped to see case coverage and case overlap data align with theoretical calculations in the GECCO paper (Hemberg, 2013).

Second we wanted to compare EGS performance to standard EvoGPJ performance. EvoGPJ would be our baseline and we hope to see EGS match or outperform it. Finally we wanted to compare EGS on Boinc to EGS and the baseline EvoGPJ. We hoped to see EGS on Boinc supersede the baseline EvoGPJ, and match the performance of standard EGS.

3.1 Case Coverage Tests

One important property we wanted to maintain was that high performing individuals are exposed to (covers) the entire training set. Therefore we set up case coverage tests. In these experiments the coordinator sent out special 'reporter' individuals to the engines. These individuals were immune to ejection from the population and are configured to return the coordinator after a specified amount of evaluations (mimicking a high performing individual). A single trial ended when the coordinator obtained a solution that had been evaluated against every test case. The coordinator logged the amount of evaluations it took for the individual to visit every test case.

In these experiments the case server distributes the test cases uniformly. Each case has an equal chance of being distributed for every request. Therefore it is possible a reporting individual is evaluated more than once on the same case before coming across a new one.

We expected as the number of cases in the training set increased so would the amount of evaluations it would take to view them all. For each training set size we ran 10 trials and took the average. Figure 6 shows the results.
We see that the populations are exposed to the entire training set and as the number of training cases increase the more generations it took for an individual to visit the entire training set. Moreover, the relationship between these two variables was linear and closely resembles the theoretical values in the GECCO paper (Hemberg, 2013).

3.2 Case Overlap Tests
Case overlap is a measure of how comparable a group of solutions are in terms of what training cases they have seen. It is calculated using the following equation:

\[
\frac{1}{n^2} \sum_{i}^{n} \sum_{j}^{n} \text{overlap}(i, j)
\]

Where \( n \) is the number of individuals in the population and \( \text{overlap}(i, j) \) is given by:

\[
\frac{\# \text{ of common cases between individual } i \text{ and individual } j}{\# \text{ of cases individual } i \text{ has seen}}
\]

As evident from the formula, the more overlap between the cases seen by individuals in the population the higher the case overlap.

In order to measure case overlap, the coordinator recorded the case overlap of the entire graduate population upon every solution submission. The sample problem the algorithm was attempting to solve was approximating a function. We expected to see the case overlap...
among the graduate population increase over time as the solutions converged to being evaluated on every training case. However, the results proved otherwise. Figure 7 shows the results.

![Overlap Data](image)

Over time the amount of overlap continually decreased, not increased. After a quick examination of the graduate population the reason for this behavior was clear. The coordinator was flooded with low age individuals (individuals evaluated on very few test cases). These individuals were able to perform well on a single training case and had thus become graduates. However in the process this would push older individuals out of the coordinator despite these solutions possibly being better overall.

In order to prevent the ejection of the older individuals the coordinator was age layered. Thus, individuals would only compete against individuals of similar age. After enabling age layering in the coordinator we expected to see the higher (older) layers to have much higher overlap than the lower levels. Figure 7 shows the results of the age-layered trials.
These results agree with theoretical calculations presented in the GECCO paper (shown in Figure 8), which state that the more evaluations a pair of individuals has the more likely they are to share a training case.
3.3 Symbolic Regression

Next, we wanted to compare the performance of various configurations of the EGS program versus the original EvoGPJ program. We hypothesized that the full system with evolutionary coordinator and case server enabled would lead to the best solutions on average.

We tested our programs by running them on the problem of function fitting a quintic function, namely:

\[ f(x) = x + x^2 + x^3 + x^4 + x^5 \]

For each EvoGPJ configuration the program was run 30 times and the fitness of the best solution was logged from each run. Below is the configuration table for each of the test programs.

<table>
<thead>
<tr>
<th>Name</th>
<th>Dummy Equalizer</th>
<th>Dynamic Equalizer</th>
<th>Limited Equalizer</th>
<th>Case Request Interval</th>
<th>Submission Interval</th>
<th>Request Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Baseline_D</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Case-1_D</td>
<td>Yes</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Case-5_D</td>
<td>Yes</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
The following parameters previously present in the non-distributed EvoGPJ program were used for all experimental configurations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
</tr>
<tr>
<td>Number of generations</td>
<td>50</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>.1</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>.7</td>
</tr>
<tr>
<td>Elite group size</td>
<td>3</td>
</tr>
<tr>
<td>Initial tree max depth</td>
<td>6</td>
</tr>
<tr>
<td>Bin width</td>
<td>5</td>
</tr>
<tr>
<td>PNorm</td>
<td>2</td>
</tr>
<tr>
<td>Brood size</td>
<td>1</td>
</tr>
<tr>
<td>Koza function rate</td>
<td>.9</td>
</tr>
<tr>
<td>Tree crossover max depth</td>
<td>17</td>
</tr>
<tr>
<td>Tree crossover tries</td>
<td>10</td>
</tr>
<tr>
<td>Tree mutate max depth</td>
<td>17</td>
</tr>
<tr>
<td>Tourney size</td>
<td>7</td>
</tr>
</tbody>
</table>

For our experiments, the decision process for which individuals are submitted is based solely on fitness. For example, if an evolutionary engine is to submit 5 individuals to the evolutionary coordinator, the 5 fittest individuals will be chosen.

3.3.1 EGS Program Results
The initial testing of the distributed program ran into an unexpected scenario. A majority of program runs seemed to run interminably. The program was unable to generate enough individuals for a new population and thus termination was impossible.

3.3.1.1 The Dummy Equalizer
Upon further inspection, it was determined that the equalization process was unable to finish when case server requests were activated. In the original EvoGPJ program a filtering mechanism called the tree dynamic equalizer (Silva, 2009) was being used to determine
which newly generated individuals would move to the next generation. The equalizer works by placing individuals in bins that correspond to the individual’s size (the size of the individual’s solution tree). If the appropriate bin has space, the equalizer will accept the individual. If the bin is full, the equalizer accepts the individuals only if its fitness is better than all other individuals currently in the bin. The purpose of the equalizer is to create a bias towards smaller solutions by making bins that correspond with smaller individuals have a larger capacity than bins that correspond to larger individuals.

After a new case request, most solutions in the population are tailored towards the previous case. When a new case arrives, all individuals are evaluated on it; thus even though a solution hasn’t changed, its fitness will. This creates high variance of fitness in the population, making it difficult for individuals to fit into the equalizer bins. Since the algorithm requires a specified population size to be reached before moving to the next generation, the equalizer’s repeated rejections of individuals caused the algorithm to stall.

In order to ensure the termination of the test runs, the tree dynamic equalizer was replaced with a dummy equalizer that accepted all generated individuals.

3.3.1.2 Analysis

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean Fitness</th>
<th>Standard Deviation</th>
<th># of Successes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.993468</td>
<td>0.006543</td>
<td>3</td>
</tr>
<tr>
<td>Baseline_D</td>
<td>0.956655</td>
<td>0.041862</td>
<td>0</td>
</tr>
<tr>
<td>Case-1_D</td>
<td>0.972661</td>
<td>0.02067</td>
<td>0</td>
</tr>
<tr>
<td>Case-5_D</td>
<td>0.976249</td>
<td>0.02196</td>
<td>0</td>
</tr>
<tr>
<td>Coord_D</td>
<td>0.954492</td>
<td>0.028933</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 11: Performance summary for dummy equalizer configurations. Bolded numbers indicate statistically significant differences from non-bolded numbers (>95% confidence from t-test).*

Figure 11 gives a quick summary of the dummy equalizer data. The results did not support what was hypothesized. The program with the evolutionary coordinator and case server enabled performed worse than the program with just the case server enabled. Both performed worse than the baseline system. These differences were concluded to be statistically significant by t-tests assuming unequal variances with 95% confidence. The proposed reasoning behind the results was the removal of the tree dynamic equalizer, which guaranteed high fitness individuals pass into the next generation.

3.3.1.3 The Limited Reject Equalizer

To solve the stalling problem, instead of replacing the tree dynamic equalizer, the amount of rejections it could give was limited. This is controlled by the `MAX_EQUALIZER_REJECTS` parameter. Once this limit is reached, the best individual that has been rejected so far is passed to the next generation and the reject counter is reset.
Performance data for all experimental configurations

Figure 12: Results from all experimental configurations side-by-side.

3.3.1.4 Analysis

The following tests (except the baseline configurations) were run with the new limited reject equalizer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mean Fitness</th>
<th>Standard Deviation</th>
<th># of Successes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.993468</td>
<td>0.006543</td>
<td>3</td>
</tr>
<tr>
<td>Baseline_D</td>
<td>0.956655</td>
<td>0.041862</td>
<td>0</td>
</tr>
<tr>
<td>Case-1</td>
<td>0.991139</td>
<td>0.013168</td>
<td>4</td>
</tr>
<tr>
<td>Case-5</td>
<td>0.993410</td>
<td>0.004479</td>
<td>1</td>
</tr>
<tr>
<td>Coord</td>
<td>0.994733</td>
<td>0.003589</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 13: Performance summary for limited equalizer configurations. Bolded numbers indicate statistically significant differences with non-bolded numbers (>95% confidence).

Looking at the results (Figure 12 and Figure 13) the benefits we see that by using the limited reject equalizer we retain the pruning effect from the original tree dynamic equalizer while having program that runs in a satisfactory amount of time. Compared to the dummy equalizer runs, the performances of the distributed programs were improved across
the board (deemed statistically significant by t-tests with >95% confidence for each configuration). We also saw that the full implementations of the program with the case server and evolutionary coordinator enabled achieved higher average fitness than implementations with only the case server. However, these differences in performance were only validated by t-tests with 80% confidence. Thus, we could not reject the null hypothesis that EGS outperformed the EvoGPJ program.

3.3.2 Exposed Design Points of EvoGPJ-Star
While testing the EGS system we discovered that the use of the tree dynamic equalizer could cause the algorithm to interminably search for new solutions. To resolve this we created a new limited reject equalizer that limited the length of this search.

In addition, the relative ease of transforming EvoGPJ into EGS shows the modularity of EGS. The communication modules for transferring data between the engine and case/coordinator servers are mostly self-contained. Thus, creating an EGS-like system from another currently existing evolutionary compute algorithm would likely not require massive changes in code structure.

3.4 Rule-Based Classification with EvoGPJ-Star
Our final experiment was designed to test both the performance of EGS and its scalability. For this experiment, the target problem was rule-based classification.

We set up 3 candidate programs to compare: the non-distributed configuration, the distributed configuration (standard EGS), and the distributed Boinc configuration (EGS on Boinc). We hypothesized the EGS and EGS Boinc configuration to at least perform on par with the non-distributed configuration.

3.4.2 Multi-Objective Fitness
In the classification tests, there are some differences in the algorithm compared to the symbolic regression builds. First, the fitness is now multi-objective. When choosing graduates, engines will prioritize the simplicity of a solution as well as the accuracy. Solutions dominate another when one solution is both simpler and more accurate. Second, the equalizer mechanism has been replaced by the multi-objective fitness evaluation.

3.4.3 Data Sets, Small and Large
The data we are trying to classify in these experiments are cardiotocography (CTG) readings (Bache, 2013). Each data point consists of 14 characteristics. The evolutionary trials would attempt to classify the data points. For our experiments we tested against data from two sources. One was from actual CTG data. This is the small data set of ~3 MB. The second data set was synthetically generated CTG data. This data was a portion of a 0.5 terabyte computer generated data set.
3.4.4 Folds and Data Size
Each data set was split into 11 folds – 10 training folds and 1 test fold. In the small data set, each training fold contains 15 training cases, each containing ~1-2 KB of data. The size of the entire small data training set is 90 KB. The size of the test fold data is 19 KB. In the large data set, each training fold contains 20 training cases, each containing approximately ~16 KB of data. The size of the entire large data training set is 3.12 MB. The size of the test fold data is 1.5 MB.

3.4.5 Testing Strategy
For each configuration (Single engine EGS, EGS, and EGS Boinc) we ran the algorithm for 250 generations for each trial. We performed thirty trials for each training fold and took averages over all the trials for our data points. We did the same for both the small and large data sets.

Trials consisted of running the algorithm for 250 generations and logging the following data for each generation:

- Best Fitness – the fitness rating of the top individual in the population
- Average Fitness – the average fitness of the population
- Average Complexity – the average solution complexity of the individuals in the population
- Average Evaluation Time – the average evaluation time for individuals in the population

In addition at the end of each trial we logged Test Fitness – a measure of how well the best individual at the end of the algorithm performed when evaluated on the testing data.

3.4.6 Parameters and Configuration

3.4.6.1 Engine Parameters
In the classification experiments engines submitted 10 individuals every generation to the coordinator and requested 5 individuals every 4 generations. Aging of individuals was enabled.

<table>
<thead>
<tr>
<th>Name</th>
<th>Submission Amount</th>
<th>Request Amount</th>
<th>Case Request Interval</th>
<th>Submission Interval</th>
<th>Request Interval</th>
<th>Number of Engines</th>
<th>VCN Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvoGPJ</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>EGS</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>EGSB</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 14: Table showing configurations of the different systems used in the experiments

The other relevant parameters are the same as in Figure 10.
3.4.6.2 Coordinator Configuration

The coordinator in the classification tests was age-layered with a maximum of 100 individuals per layer. Each age range covered two ages, and was layered up to the amount of training cases within a fold. When distributing graduates to the engines, the coordinator first chooses a random layer containing at least one individual and then chooses a random individual from that layer.

3.4.6.3 System Configurations

The following describe the 3 systems we compare in our experiments:

- Single Engine EGS – EGS with a single engine
- EGS – EGS with 8 engines. Engines run trials simultaneously. In our tests engines were run on separate threads on the same machine.
- EGS on Boinc – EGS with 8 engines. Engines are Boinc tasks executing on separate remote machines.

3.5 Performance Results

In the following sections we compare the performance of the 3 system configurations on both the small and large data sets.

3.5.1 Small Data Set Results

Below are the Best Fitness results for 3 configurations on all 10 folds of the small data set.

---

**LEGEND:**

- X-Axis – generations
- Y-Axis – Fitness of highest ranked individual
- Single-engine
- EGS
- EGS w/Boinc

---
In order to make sure the best individuals generated were not over-fitted to the training data, we evaluate the best individual on the testing fold and compared the fitness before and after the evaluation. Figure 16 shows the results.

<table>
<thead>
<tr>
<th>SMALL</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
</tr>
</thead>
<tbody>
<tr>
<td>EG</td>
<td>.134 ± .076</td>
<td>.159 ± .073</td>
<td>.159 ± .083</td>
<td>.129 ± .080</td>
<td>.129 ± .078</td>
</tr>
<tr>
<td>Test</td>
<td>.180 ± .061</td>
<td>.182 ± .045</td>
<td>.205 ± .046</td>
<td>.205 ± .046</td>
<td>.199 ± .030</td>
</tr>
<tr>
<td>EGS</td>
<td>.026 ± .041</td>
<td>.021 ± .041</td>
<td>.086 ± .0876</td>
<td>.108 ± .075</td>
<td>.136 ± .085</td>
</tr>
<tr>
<td>Test</td>
<td>.076 ± .078</td>
<td>.168 ± .053</td>
<td>.177 ± .057</td>
<td>.154 ± .052</td>
<td>.183 ± .053</td>
</tr>
<tr>
<td>EGSB</td>
<td>.064 ± .073</td>
<td>.072 ± .072</td>
<td>.078 ± .072</td>
<td>.068 ± .071</td>
<td>.115 ± .065</td>
</tr>
<tr>
<td>Test</td>
<td>.121 ± .070</td>
<td>.130 ± .067</td>
<td>.087 ± .074</td>
<td>.113 ± .075</td>
<td>.114 ± .076</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SMALL</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F9</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>EG</td>
<td>.143 ± .034</td>
<td>.117 ± .085</td>
<td>.071 ± .064</td>
<td>.133 ± .065</td>
<td>.092 ± .059</td>
</tr>
<tr>
<td>Test</td>
<td>.208 ± .037</td>
<td>.185 ± .030</td>
<td>.175 ± .026</td>
<td>.176 ± .034</td>
<td>.171 ± .028</td>
</tr>
</tbody>
</table>

Figure 15: Best fitness results for the small data set
<table>
<thead>
<tr>
<th></th>
<th>Training</th>
<th>Test</th>
<th>Training</th>
<th>Test</th>
<th>Training</th>
<th>Test</th>
<th>Training</th>
<th>Test</th>
<th>Training</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGS</td>
<td>.132 ± .078</td>
<td>.080 ± .065</td>
<td>.099 ± .069</td>
<td>.097 ± .044</td>
<td>.105 ± .064</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td>.195 ± .036</td>
<td>.161 ± .042</td>
<td>.180 ± .026</td>
<td>.142 ± .037</td>
<td>.158 ± .039</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EGSB</td>
<td>.021 ± .076</td>
<td>.074 ± .065</td>
<td>.087 ± .063</td>
<td>.106 ± .047</td>
<td>.089 ± .062</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td>.093 ± .087</td>
<td>.134 ± .062</td>
<td>.123 ± .061</td>
<td>.110 ± .058</td>
<td>.132 ± .061</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 16: Table comparing the fitness rating (with standard deviation) of the fittest individual in the population at the end of a run to its fitness rating given from the test data set. Numbers are an average over 30 trials. Bolded test results indicate significant differences with the training result above it.

Unfortunately, we see that some degree of over-fitting occurred in most of the folds (validated by 95% t-tests), irrespective of the system used.
3.5.2 Large Data Set Results

Below are the Best Fitness results for the 3 configurations on all 10 folds.

**LEGEND:**
- X-Axis – generations
- Y-Axis – Fitness of highest ranked individual
- Single-engine
- EGS
- EGS w/Boinc

![Graphs of Best Fitness Fold 1 (Large), Best Fitness Fold 2 (Large), Best Fitness Fold 3 (Large), Best Fitness Fold 4 (Large)]
Again, to make sure our individuals were not over-fitted, we evaluate the best individual against the testing data. Figure 18 shows the results.
Again, we see that some degree of over-fitting occurred in most of the folds (validated by 95% t-tests), with the exception of the single-client configuration.

3.6 Analysis

3.6.1 Small Data Set Tests

Our results from our CTG experiments showed that after 250 generations, the distributed configurations outperformed the single-engine in general. Figure 19 highlights the significant performance differences.
We see that in general the distributed versions of the program outperform the single client version in both best fitness and test fitness scores. We also see that the distributed systems performed similarly (some folds EGS performed better, while in others EGSB performed better). This is expected since they are essentially the same system deployed to different networks.

3.6.2 Large Data Set Tests
The synthetic data experiments yielded much more convincing results. In every fold the distributed systems outperformed the single-engine system. Between the EGS and EGS w/ Boinc implementations, however, results were fairly comparable with the exception of fold 2 and 3 (in which the EGS w/ Boinc system performed significantly better than the standard EGS system). Figure 20 highlights the significant performance differences.

![Table showing significant performance differences between the systems on the small data set. X indicates a significant difference (t-test, 95%), the arrow indicates which system performed better.](image)

We observe similar results as in the small data set. The distributed systems routinely outperformed the single client system, but were on par with each other.

3.6.3 Asymmetric Generation Counts
From the results we see that the distributed versions of EvoGPJ outperformed the single-engine as we hypothesized. However, given that in these configurations it is possible that 8 times the amount of generations were run (albeit in parallel) compared the single-engine version the comparison isn’t entirely valid when taking into account global computation time. However, if compared on per-engine basis, the distributed system yields performance benefits. To make a more valid comparison we allotted the single client configuration the same amount of generations as the global versions (2000 generations). Figure 21 shows the best fitness results for Fold 4.
Figure 21: Best fitness results for the small and large data sets, comparing the distributed EG configurations to the single client configuration run for the same amount of total generations.

We see that despite having the same global amount of generations, the single client configuration yields less optimal solutions on average for the small and large data set (verified by t-test with 95% confidence). This indicates that the distributed nature of EGS helps the algorithm explore the solution space more efficiently.

As before, we also check for over-fitting. Figure 22 shows the results.
<table>
<thead>
<tr>
<th></th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGS</td>
<td>.304 ± .078</td>
</tr>
<tr>
<td></td>
<td>.281 ± .028</td>
</tr>
<tr>
<td></td>
<td>.302 ± .023</td>
</tr>
<tr>
<td></td>
<td>.287 ± .034</td>
</tr>
<tr>
<td></td>
<td>.292 ± .019</td>
</tr>
</tbody>
</table>

Figure 22: Tables comparing the fitness rating of the fittest individual in the population at the end of a run to its fitness rating given from the test data set. Numbers are an average over 30 trials. Bolded numbers indicate significant differences (Fold 4).

Interestingly, in the 2000 generation single-client runs, we do not see any evidence of over-fitting.

3.6.4 Over-fitting
We observed over-fitting in almost every fold and configuration combination, with the exception of the single-client configuration on the large data set and the 2000 generation single-client configuration on both data sets. We suspect that the over-fitting is occurring for three reasons. One is the lack of cross-fold validation and per fold test evaluation. The second is that often the best individual that was chosen to be evaluated on the test data was young and its fitness rating was only suitable for the most recently acquired cases. And finally, the amount of generations was not enough to cycle through the training data sufficiently. This is further supported by the 2000 generation single-client configuration showed no indications of over-fitting. We hope to test these hypothesis in the future.

3.6.5 Solution Complexity and Evaluation Time
We expected to see a positive correlation between average solution complexity and average evaluation time (larger solutions should take longer to evaluate). This held true with the exception of our EGS standard implementation. The EGS standard implementation failed to show a positive correlation between evaluation time and solution complexity (Figure 23). We suspected this was due to the fact that for the EGS system we assigned multiple engines to a single multi-threaded machine. Thus, due to thread scheduling evaluation times could become erratic. This assessment was validated when the single-client and EGS w/ Boinc configuration demonstrated the hypothesized positive correlation.
Additionally, we found that evaluation time in the small data set trials were much smaller than those in the large data set trials. This is reasonable given the cases in the large data set contained more data points.
4 Discussion of EC on VCN

During the implementation and testing of EGS on a VCN, the advantages and pitfalls of the system were made apparent. Some of these were specific to Boinc, and others apply to VCNs in general.

4.1 Boinc Specific Advantages

Boinc allows hassle-free allocation of volunteer resources. An engine that installs the Boinc engine and connects to the project will have a relatively smooth experience. Boinc automatically halts execution of tasks when the CPU, memory, network, or other computer resource is currently being used or attempted to be used. None of this logic needed to be included in our engine code. The single submit feature of Boinc also made it easier to use Boinc to distribute the engine code.

4.2 Boinc Specific Drawbacks

The drawbacks of Boinc mostly stem from its asynchronous nature. Coordinating engines can be quite tricky since engines running programs may arbitrarily be interrupted, terminated entirely, or may run multiple programs at once.

To compare EGS on Boinc to standard EGS we needed to make sure that each engine was only running one instance of an engine at a time. However, if there is enough CPU headroom Boinc is designed to grab as many instances of a program as possible from the Boinc server without overloading the allotted CPU resources. In order to overcome this issue, we manually configure each engine such that the CPU headroom was only enough for one engine program at a time.

In addition, by default Boinc will terminate applications that have not progressed after a period of time. In our EGS system it is common for an engine to be waiting to start its next trial (the other engines in the system are not done with the current trial yet). If the amount of time spent waiting passes Boinc’s timeout parameter, the engine will kill the program, effectively ruining the experiment. Thankfully, digging around in Boinc’s configuration files yielded the option to increase the timeout to a substantially larger amount of time.

4.3 Drawbacks of VCNs in General

When using a VCN many problems simply stem from the inability to fully control client machines. For example, a client’s firewall may disable communication to the coordinator or case server.

However, the main drawback present in all VCNs, is the ability for nodes to exit the network mid-execution or delay execution interminably. This makes it difficult to coordinate comparative experiments and causes synchronization inefficiencies. Engines must wait until the slowest engine completes a trial before advancing. In a traditional
cluster this wait time exists, but usually cannot be interminably delayed. This is not the case in a VCN.

These fixes for these asynchronicity issues are not ideal. Handling such problems requires a more complex system. A system that could possibly allow handoff of work from one engine to another in the case of halt or termination. This would likely require substantial modification to the current EGS code base and/or deeper Boinc integration (difficult because of the lack of Java API).

We found that VCNs do not provide an ideal experiment environment due to their asynchronous and asymmetric nature. It is more suitable for enhancing evolutionary computation projects with no strict requirements on when compute power should be available.

4.4 Use Cases of EC on VCN
Here we discuss 3 possible alternative use cases of EC on a VCN and discuss their merits and drawbacks. We hope to look into these models of EC computation in the future and compare them to an EGS-like system.

4.4.1 Strictly Evaluation Compute Power
In this model, the VCN clients are strictly used for evaluating clients but there is only one global evolutionary algorithm running. This is not as robust as EGS since the failure of a node may halt the algorithm.

4.4.2 Non-Coordinated Engines
Simply use VCN nodes to run separate evolutionary algorithms that do not interact with each other via coordinator (i.e. the engines are self-contained). This solution is resistant to node failure but fails to offer the benefits of mixing solutions via migrants.

4.4.3 Non-Synchronized Coordinated Engines
This is the same design as EGS however there are no synchronized trials. Client simply run trials whenever possible and the coordinator clears its solutions periodically. Unfortunately this model is hard to run comparative experiments on. However, in our preliminary experiments it has shown to produce comparable performance to EGS in terms of solution fitness.

5 Conclusions and Future Work
Over the course of this project we successfully constructed a distributed evolutionary computation system and deployed it to a VCN. Our design model, based on EC-Star allows for scaling in both the number of engines and the size of the training data. We confirmed fundamental theoretical assumptions with regards to case coverage and case overlap held true just as in the non-distributed version of the same system. The rate of complete
coverage scaled linearly with the amount of total cases, and layers of older individuals had
greater rates of overlap. In addition, using the single-engine setup as a baseline, we saw that
in general the distributed versions produced healthier solutions on a per-engine basis. In
addition, we exposed design benefits (more computation power) and drawbacks
(asynchronous nature makes it difficult to standardize experiments) that emerged when
deploying EGS (and likely similar evolutionary compute systems) on a VCN. Finally, we
discussed how a VCN produces synchronization inefficiencies when running multi-trial
experiments.

Looking forward it might be interesting to optimize our program parameters to acquire
even better performance. In addition, it would be preferable to compare trials on a per
generation basis. For example, instead of a trial ending when all engines have finished their
allotted amount of generations, the trial ends once a global number of generations have
been completed. At this point, a ‘kill switch’ signal would be sent to the engines, telling
them to begin the next trial.

We would also like to look into determining the source of over-fitting in our solutions by
implementing per-fold testing and cross validation into our experiments.

In addition, we would like to directly compare overlap metrics with the theoretical
calculations in the GECCO paper by comparing pairs of individuals and seeing how likely
they are to share a single training case.

Also, in order to make experimentation within a VCN more feasible, it may be worth
constructing a C version of EGS to utilize Boinc APIs and integrate a more complex
synchronization system.

And finally, we hope to explore alternative models of EC on VCNs and assess their
performance relative to EGS.
References


Erik Hemberg, Kalyan Veeramachaneni, Franck Dernoncourt, Mark Wagy and Una-May O'Reilly. Imprecise Selection and Fitness Approximation in a Large-Scale Evolutionary Rule Based System for Blood Pressure Prediction, to appear in GECCO 2013.
