Design and Implementation of a Framework for Performing Genetic Computation Through a Volunteer Computing System

Eamon Walsh

Research Science Institute
Mentor: Luis F. G. Sarmenta

Laboratory for Computer Science
Massachusetts Institute of Technology

August 5, 1998

Abstract

This project designed and implemented a transparent software framework that allows any genetic algorithm to be run in parallel over the Internet. The framework distributes the population of the genetic algorithm among computers which have been volunteered to assist in the task. However, the parameters provided to the genetic algorithm are independent of the number of processors being used. The framework makes use of Project Bayanihan, a volunteer computing package written in the Java programming language. A sample genetic algorithm was created and executed using the framework, and consistent, accurate results were obtained. Benchmark tests indicate that there is a significant improvement in execution speed when using the framework as compared to a single machine.

1 Introduction

1.1 Genetic Algorithms

One of the most promising methods for solving problems computationally is the genetic algorithm. Genetic algorithms function by maintaining a population of chromosomes and performing operations on that population, simulating the biological evolutionary process. Each chromosome encodes a possible solution to the problem being solved. By manipulating the chromosomes much like strands of DNA – through combination, crossover, and mutation, the genetic algorithm produces new chromosomes from the old ones. By selecting the chromosomes with the best solutions from successive generations, the genetic algorithm eventually reaches a final, optimized result [2].

The following pseudo-code describes the work cycle of a basic genetic algorithm [6].

```
initialize a (random) population.

test for termination criteria (time, fitness, etc.)
while not done do

        give each chromosome an evaluation (fitness)
        select parent chromosomes for reproduction
        produce children through genetic recombination
        perform other genetic operations (mutate, etc.)
        evaluate the children
        select survivors to advance to the next generation

end loop
```

Genetic algorithms have numerous real-world applications. Their high degree of adaptibility and versatility makes them ideal for problems with sudden or unexpected conditions.

They are used to construct exam timetables, manage network routing, find maxima or minima, and play complex games [6].

One weakness of genetic algorithms is their need for significant computing power. The cycle described above is slow because operations must be performed on each chromosome every generation, and typical genetic algorithms maintain populations of thousands of chromosomes. However, because the same operations are performed on each chromosome, genetic algorithms can be made to run in parallel, and have been described as "highly parallelizeable" [4]. Thus, one way to provide the necessary computing power is to adapt genetic algorithms to a parallel computing environment.

1.2 Project Bayanihan

Mr. Luis F. G. Sarmenta of the MIT Laboratory for Computer Science has written a parallel computing framework in the Java programming language. This project, Project Bayanihan, attempts to take advantage of a computing resource already in existence – millions of personal computers in homes and offices [7].

When performing computations on a single machine, the typical flow of operations proceeds as shown in Figure 1.2a. A certain amount of work is performed, the results are displayed, and the next batch of work is started. The flow of operations over a Project Bayanihan application is similiar, as shown in Figure 1.2b. In this case, however, the work is first divided into many parts by the "server." The parts are then made available to "worker" client computers who have volunteered to assist in performing the task. Each worker completes a part of the work and sends the result back to the server. Finally, these results are

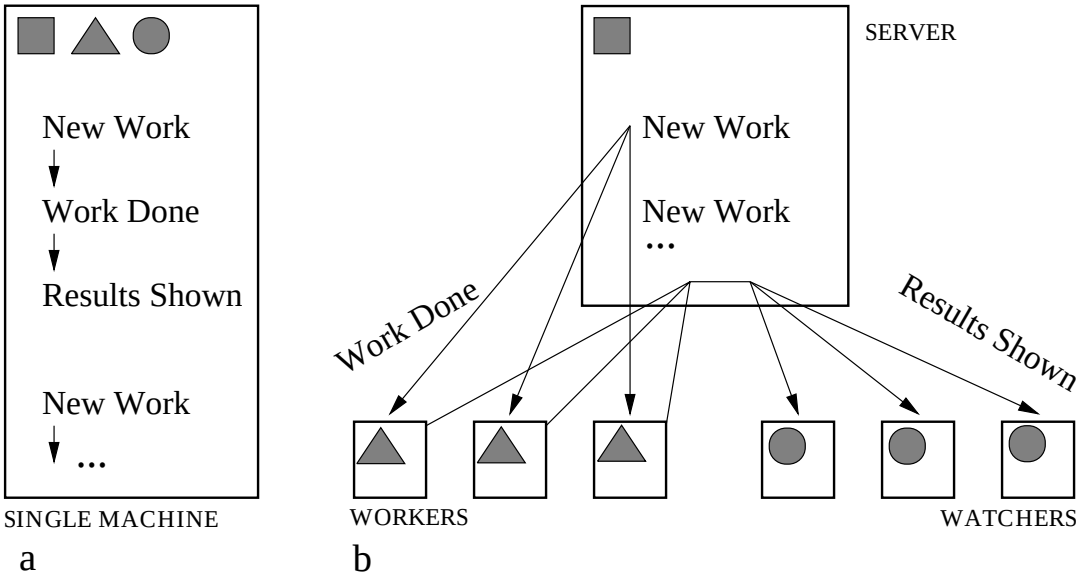

Figure 1: Flow of computing operations: a)single machine b)Project Bayanihan

sent to "watcher" clients that display them to the user.

Through Project Bayanihan, computers can be volunteered over the Internet to assist in a large computing problem. This framework can be used by organizations to "recruit" computing power from interested Internet users. Such a technique has already been used by distributed.net to solve large-scale computing problems [1]. Companies can also use Project Bayanihan over local area networks to take advantage of existing personal computers [7].

1.3 Purpose

The ability to run genetic algorithms using Project Bayanihan would provide a way for an organization or company to take advantage of genetic algorithms without the limitation of providing the necessary computing power. In addition, Project Bayanihan would provide the advantages of a parallel processing environment, making the genetic algorithm run faster.

This project designed and built a sub-framework using the Bayanihan system. The sub-

framework defines methods and data objects which can be used to run genetic algorithms through Project Bayanihan.

2 Design

2.1 Adapting Genetic Algorithms to Project Bayanihan

When running a genetic algorithm on a single machine, the work cycle flows as shown in Figure 2a. The single machine must process each chromosome in the population.

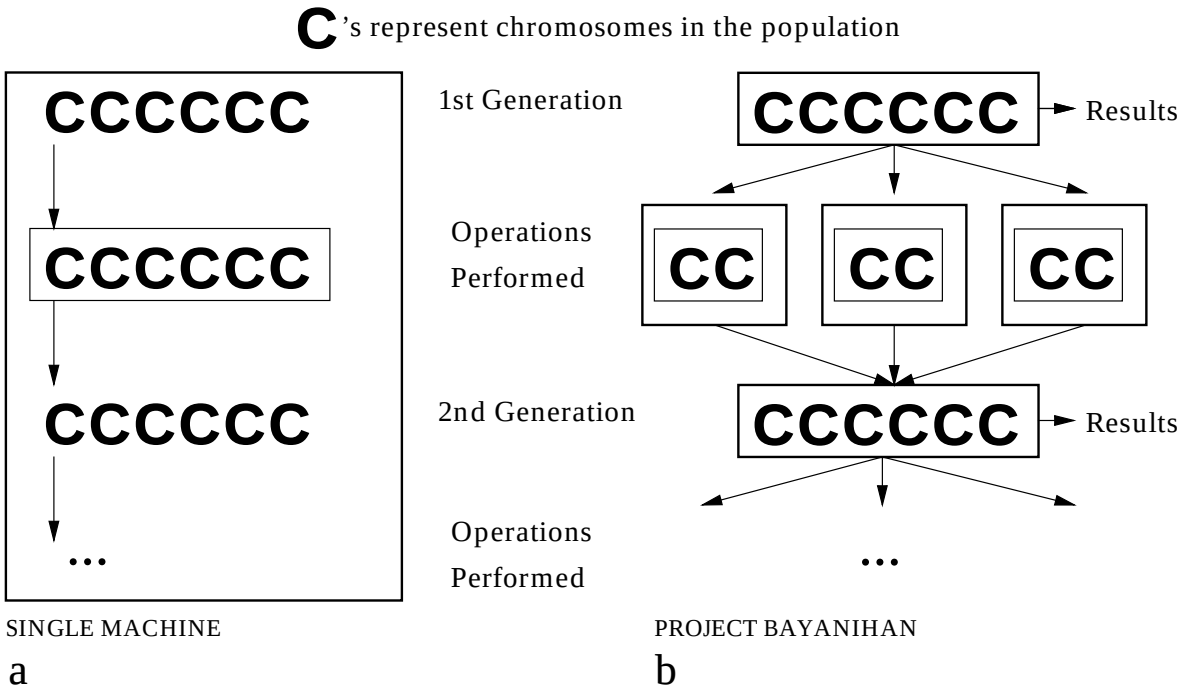

Figure 2: Work flow of a genetic algorithm: a)single machine b)Project Bayanihan

Under Project Bayanihan, however, the population can be split up and sent to many separate computers for processing, as shown in Figure 2b. Under this design, the server initializes the population. Then, for every subsequent generation, the population is split up

and sent to the worker clients. The workers perform all of the operations inside the `while` loop of the genetic algorithm. The survivors are returned to the server, where they are reinserted into the population. The cycle then repeats. The watcher clients are given information each generation, so the user can monitor the evolution process. After the specified number of generations, the server selects the best chromosome from the population and reports it as the final answer.

2.2 Issues Addressed by the Sub-Framework

In the design of the sub-framework itself, two main issues were addressed:

Adaptability. The sub-framework provides support for any genetic algorithm by allowing the programmer to implement the application-specific parts of the algorithm. The programmer must provide the following information to the sub-framework:

- Format for encoding information in the chromosomes
- Population size
- Number of generations before stopping
- Evaluation function for scoring the chromosomes
- Reproduction operators (mutation, crossover, etc.)
- Survival function for performing natural selection

However, the programmer can override more of the sub-framework if he or she wishes to exert more control over its operation. For example, the default user interface can be overridden if the programmer wishes to provide a custom graphical interface.

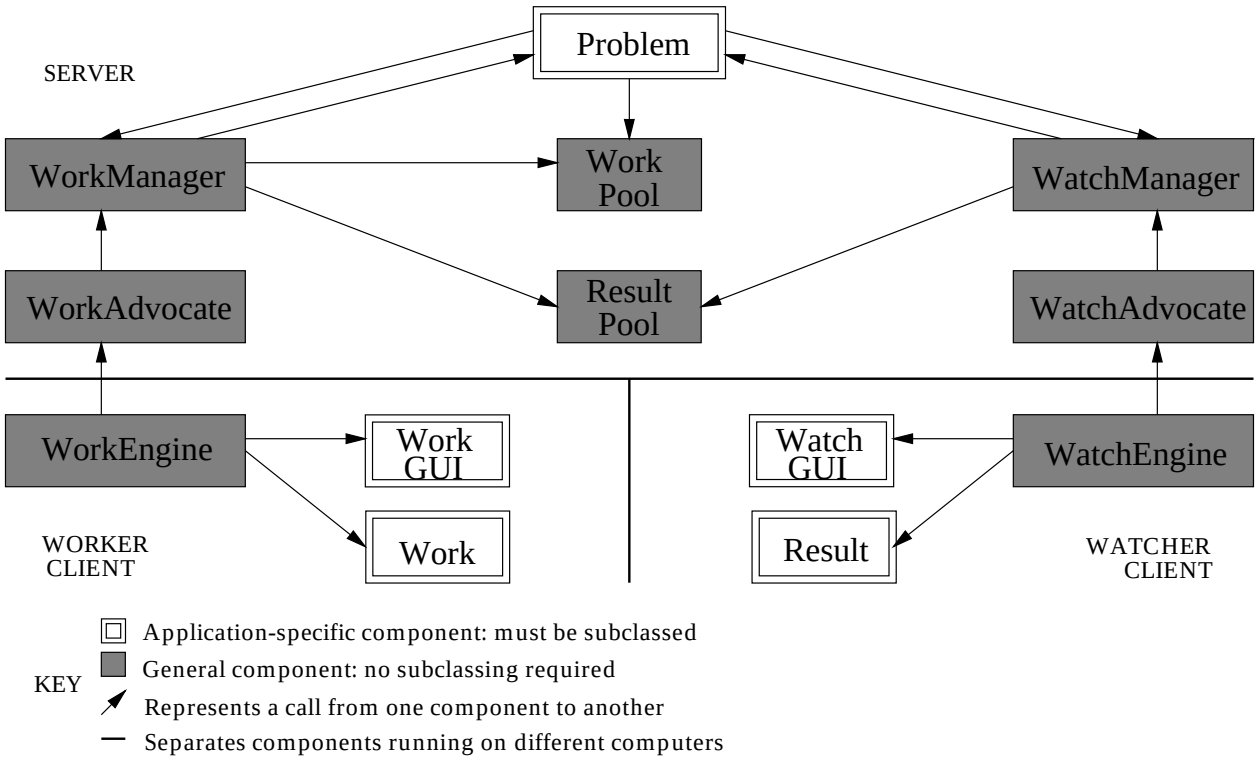

Figure 3: Project Bayanihan components in the server-worker-watcher model. Each rectangle in the figure represents a working part of Project Bayanihan. The arrows represent method calls made from one part to another.

Performance. Under the framework, the population of chromosomes is divided into groups, and each group is sent as a separate work object. If the size of the groups is too small, network overhead causes a loss in performance. If the size is too large, the worker clients become overloaded, and the benefits of parallel computing are lost. The population size and group size, as well as the number of generations to be produced, must be specified explicitly by the programmer. In this way, the framework can be modified to provide optimal performance for any genetic algorithm.

3 Implementation

3.1 Internal Structure of Project Bayanihan

Project Bayanihan is written in Java, and each working part of the project is a separate Java class (a collection of methods and data which is treated as a single object). Low-level data transfer is handled by Hirano Object Resource Broker (HORB), a distributed object package which allows Java objects to communicate remotely [5]. Figure 3 shows the structure of the server-worker-watcher model used by Project Bayanihan [7].

To write an application using Project Bayanihan, the following problem-specific information must be provided to the model:

- The type of work to be done (Work)
- The type of results to be returned (Result)
- Methods for creating and distributing work (Problem)
- Graphical user interfaces for the worker and watcher clients (WorkGUI, WatchGUI)

This information is provided by extending, or subclassing, Project Bayanihan components. The outlined rectangles in Figure 3 indicate those parts of Project Bayanihan which must be subclassed to create an application. These parts are *abstract* classes; they contain undefined methods and thus cannot be instantiated. The subclasses written by the applications programmer implement the required methods and hold the appropriate data for the specific problem being addressed [7]. The shaded rectangles in Figure 3 indicate generic components which work for most problems and do not require modification.

3.2 Implementing the Sub-Framework

In building the sub-framework for genetic computation, the set of possible problems was reduced from any conceivable computational work to any genetic algorithm. Thus, most of the information required by Project Bayanihan is the same for all cases:

- The work is always the sequence of operations in the `while` loop shown on page 3.
- The data being worked with and the results are always groups of chromosomes.
- The method for dividing the work is always the same: divide the population into groups, send the groups to the workers to have the operations performed, and then recombine them into the population. Finally, report the results to the watchers and repeat.
- A standard user interface can be provided for all genetic algorithms, because in all cases the data being shown to the user is chromosome evaluation.

The sub-framework provides the required subclasses for the server-worker-watcher model. It also introduces three new classes: `Chromosome`, `ChromosomeGroup`, and `ReproductionOperator`.

3.3 Structure of the Sub-Framework

Table 1 summarizes the components of the genetic algorithm sub-framework.

The abstract `Chromosome` class contains undefined methods for initializing and scoring chromosome data. Each genetic algorithm must provide a subclass of `Chromosome` which implements the methods and encapsulates the algorithm-specific chromosome data. The generic `ChromosomeGroup` class stores arbitrary-sized groups of `Chromosome`s, and needs no special customization.

The `GeneticWork` class is an abstract class which maintains a small `ChromosomeGroup` and has abstract methods associated with the operations in the `while` loop of a genetic

Component	Is a Subclass of	Purpose
Chromosome	—–	Contains genetic data
ChromosomeGroup	—–	Stores a group of chromosomes
ReproductionOperator	—–	Performs a single genetic operation
GeneticWork	Work	Performs evaluation, reproduction, and survivor selection
GeneticResult	Result	Returns surviving chromosomes after work is performed
GeneticProblem	Problem	Maintains the population; creates and distributes work
GeneticWorkerGUI	WorkGUI	Displays work status information on the worker clients
GeneticWatcherGUI	WatchGUI	Displays results to the user on the watcher clients

Table 1: Summary of framework components

algorithm. When the `doWork()` method is called by the work engine, these operations are performed on the chromosomes in the `ChromosomeGroup`, and the resulting recombined chromosomes are returned in a `GeneticResult` object. Each genetic algorithm must provide a subclass of `GeneticWork` which implements the required methods.

The `GeneticResult` class simply encapsulates a `ChromosomeGroup` and needs no extension.

The `GeneticProblem` class maintains a `ChromosomeGroup` object as the population. When its `createWork()` method is called, it divides the population into smaller `ChromosomeGroups` and assigns each one to a `GeneticWork` object. When a `GeneticResult` is returned, its `ChromosomeGroup` is put back into the population, and the results are sent to the watcher clients. When all the work is complete, `createWork()` is called again and the next generation begins. Three parameters are maintained in this class: the population size, the work size, and the maximum number of generations. These are algorithm-specific and are specified by

the programmer.

Each genetic algorithm may implement one or more subclasses of `ReproductionOperator`. These subclasses contain methods for performing genetic recombination and manipulation. They are used in the `recombine()` method of `GeneticWork`.

Finally, the `GeneticWatcherGUI` and `GeneticWorkerGUI` classes implement generic user interfaces for viewing results of the computation. These interfaces can be overridden if the programmer wishes to provide a custom graphical interface, but they will work for any genetic algorithm.

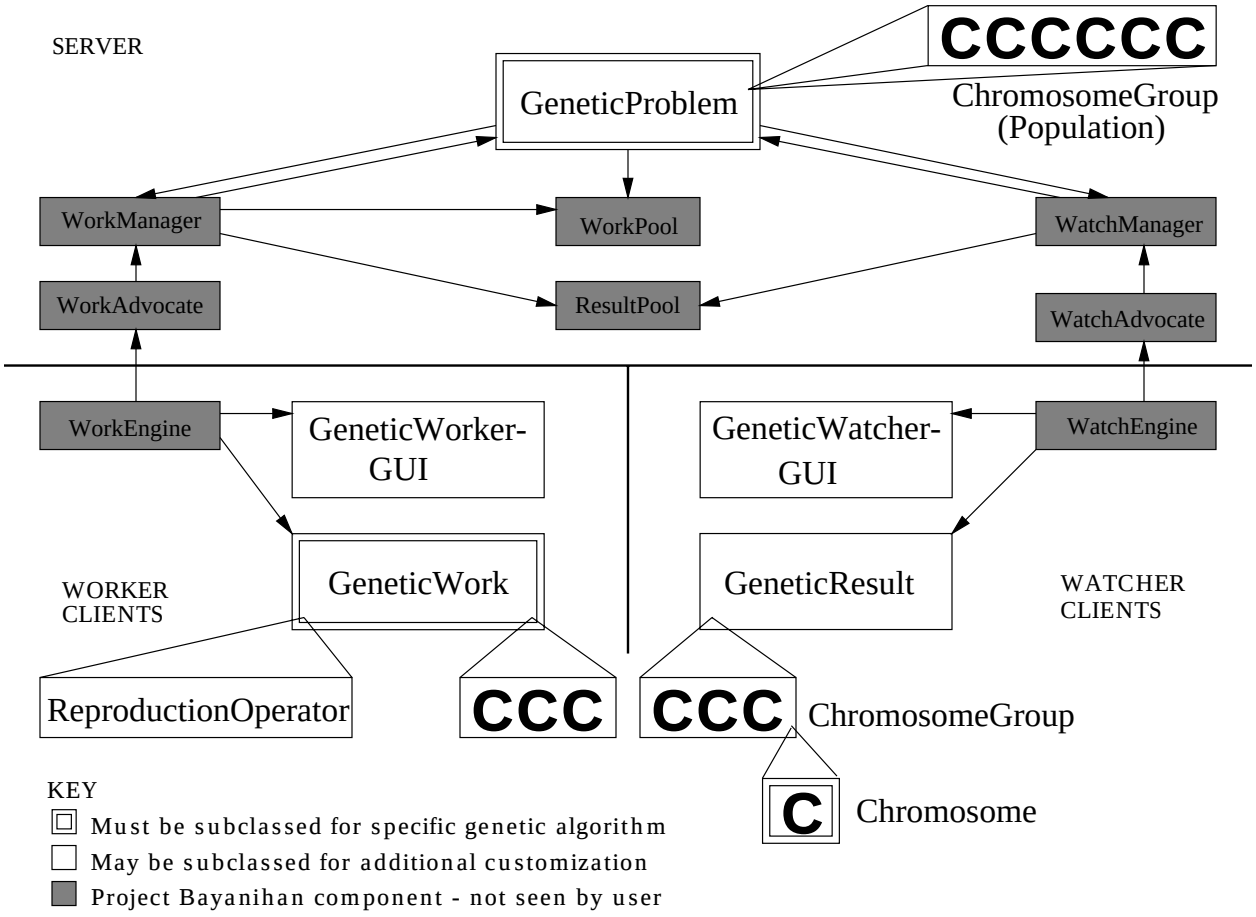

Figure 4: Structure of the genetic algorithm framework

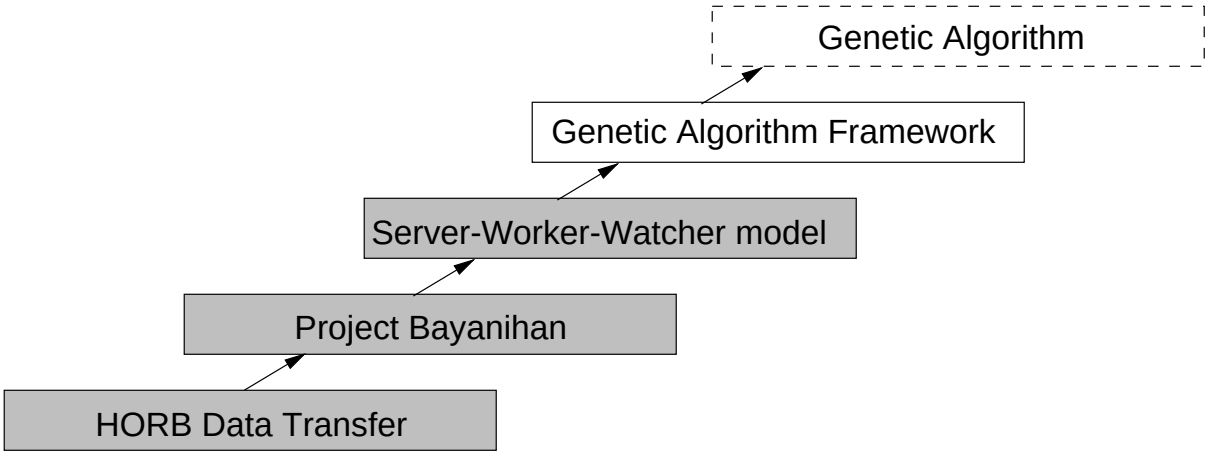

Figure 5: Hierarchy of framework system

Figure 4 shows the structure of the genetic algorithm sub-framework model. It is much like the server-worker-watcher model. However, the only information needed by this model is the specific parameters of the genetic algorithm. The interface to Project Bayanihan is completely contained within the model. Figure 5 shows the complete hierarchy of the framework system, with HORB at the lowest level, and a specific genetic algorithm running at the highest level.

4 Testing

After the framework was developed, a sample genetic algorithm was written using the framework and run over the Athena network at MIT. The genetic algorithm finds the absolute maximum value of a function $f(x, y)$ by maintaining a population of chromosomes which contain encoded x and y values. The genetic material is a string of 44-bits, which is divided into two 22-bit numbers. The numbers are scaled from the range $[0, 22^2 - 1]$ to the range $[-100, 100]$ and used as inputs to the function.

The score of a chromosome is equal to its function value. Thus, since the higher-scoring chromosomes come to dominate the population, the population eventually evolves to the point where a maximum is found [2].

The sample genetic algorithm provides the required subclassing of the `GeneticWork`, `GeneticProblem`, and `Chromosome` objects. It implements two `ReproductionOperators`, a mutator and a crosser. The mutator has a 5 in 1000 chance of flipping a single bit in the chromosome data. It serves to introduce random change in the x or y value, possibly producing a higher-scoring chromosome. The crosser exchanges x and y data between two chromosomes, producing a new (x, y) point which has potential for a higher score.

In addition, the standard `GeneticWorkerGUI` and `GeneticWatcherGUI` were overridden to provide a custom graphical interface, displaying a graph of scores and a radar screen showing the x and y points encoded in each chromosome. A screen shot of the running algorithm is shown in Figure 6.

Two trial runs were performed. In the first trial, the algorithm was run with a single worker client on one machine. In the second trial, four worker clients on separate machines were used. In each trial, the algorithm was given three functions to maximize, which are shown in table **??**. Timing measurements were taken each generation using Project Bayanihan's built-in timer. After five generations, the highest-scoring chromosome in the population was returned and it's score recorded.

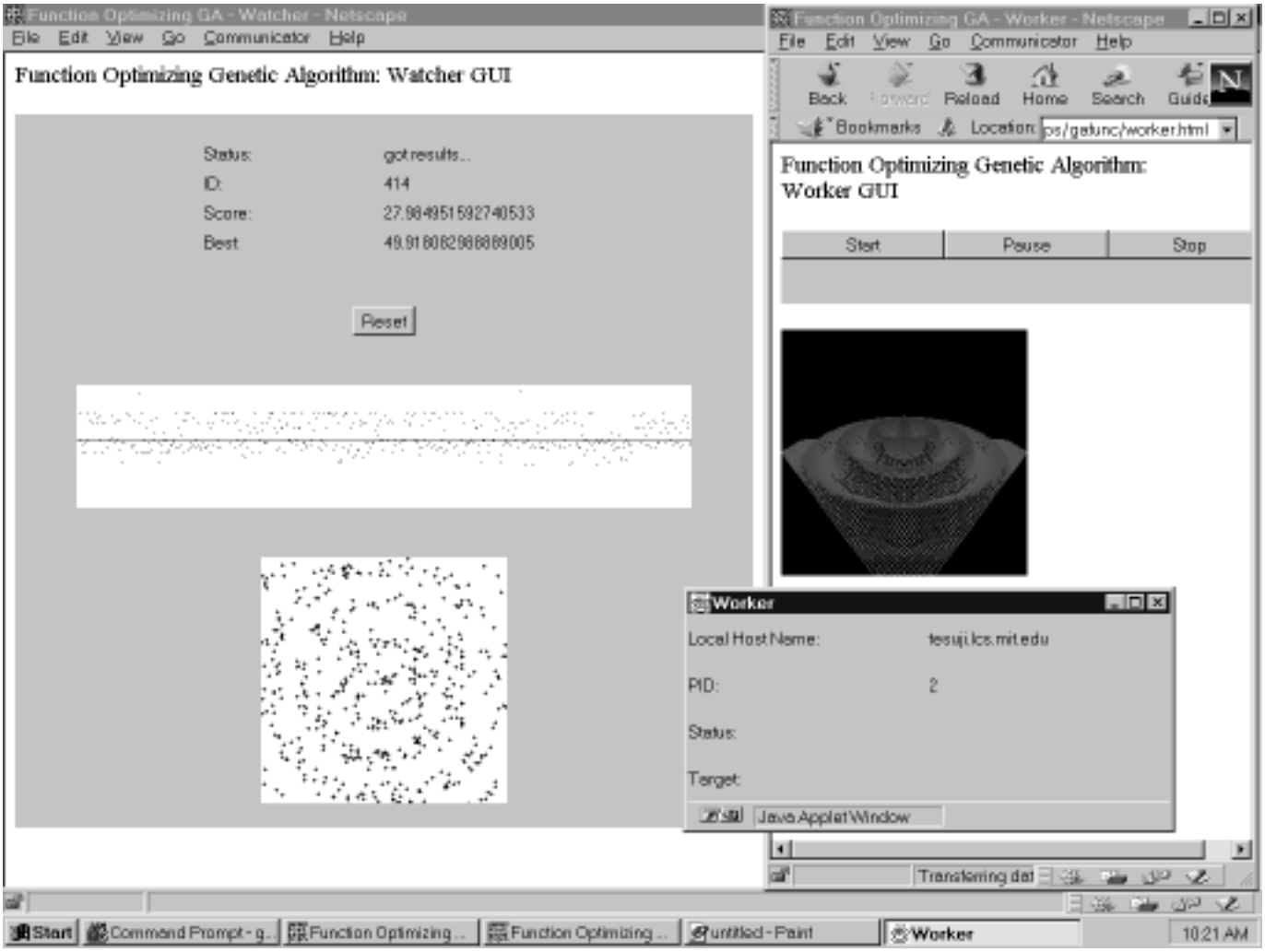

Figure 6: Sample genetic algorithm running on the framework

Function 1	$z = 10 - \lvert x^2 + y^2 \rvert$
Function 2	$z = -\frac{x^2+y^2}{1000} + 10$
Function 3	$z = \frac{1}{10}(100 - \sqrt{x^2+y^2})cos\sqrt{x^2+y^2}$

Table 2: Functions which were maximized in the trial runs. Each function has a maximum value of z = 10 at the origin.

	Max	**Trial 1**	**Trial 2**	**Avg.**	**%Error**
Function 1	10	9.9993	9.9983	9.9988	0.0112
Function 2	10	9.9976	9.9990	9.9983	0.0117
Function 3	10	9.9972	9.9641	9.9807	0.1935

Table 3: Accuracy of the sample genetic algorithm.

	Gen. 1	**Gen. 2**	**Gen. 3**	**Gen. 4**	**Gen. 5**	**Avg.**
Function 1	2302	2254	2212	2233	2265	2253
Function 2	2273	2279	2239	2224	2268	2257
Function 3	2245	2274	2312	2267	2254	2270
Function 1	724	569	598	817	602	662
Function 2	587	656	911	587	674	683
Function 3	620	825	646	709	753	711

Table 4: Timing measurements in milliseconds. The first three rows are the data from the single machine. The last three rows are the data from the four parallel worker clients.

5 Results

The algorithm succeeded in finding the maximum value of the three different functions with an error of less than 0.2% in all cases. Table 2 contains complete results.

In the first trial, one generation was finished approximately every 2250 ms. In the second trial, one generation was finished approximately every 650 ms. Network overhead caused a loss in performance during the second trial. Even so, the four machines working in parallel were three times faster than the single machine. Table 3 contains the complete results of the timing measurements.

6 Discussion

Now that the sub-framework is complete, genetic algorithms can be run in parallel over the Internet, using computers that have been volunteered by their owners to participate. This method of high-performance computing provides a new way for organizations and companies to solve real-world problems using genetic algorithms.

The fact that the genetic algorithm is running over multiple computers is transparent to the programmer using the sub-framework; no data transfer or work control routines need to be provided. Only the parameters of the genetic algorithm itself are required, and the sub-framework controls the execution through Project Bayanihan. The two issues discussed in the design section were adequately addressed: the framework is robust (providing support for any genetic algorithm), and executes quickly (outperforming single machines).

In the future, the sub-framework may be compared with other parallel computing sys-

tems, such as distributed.net [1] or PVM [3]. These systems use other models for distributing work and collecting results. They also implement other user interfaces. Genetic algorithms may be adapted to run on these systems, in which case a comparison of adaptability and performance could be made to identify the optimal distribution model and interface.

7 Acknowledgements

Acknowledgements go first and foremost to my mentor, Mr. Luis F. G. Sarmenta, who provided invaluable assistance in helping me to learn the inner workings of Project Bayanihan, as well as providing me with the necessary tools to complete the research.

I wish to thank my teacher-advisor, Mr. Don Hyatt, for his help with the application process, as well as his constant guidance and support in all of my computer science endeavors.

Acknowledgements go next to Ramesh Johari and various other RSI alumni, who provided assistance in the paper-writing process.

Finally, acknowledgements go to the Center for Excellence in Education for providing me with the opportunity to conduct this research. Their generous gift is greatly appreciated.

References

[1] A. L. Beberg, J. Lawson, and D. McNett. "Distributed.net: The World's Fastesest Computer." Online. Availible `http://www.distributed.net`. 11 Nov. 1998.

[2] Lawrence Davis. Handbook of Genetic Algorithms. New York: Van Nostrand Reinhold, 1991.

[3] A. Geist. PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallelism. Cambridge: MIT Press, 1994.

[4] Joerg Heitkoetter. "The Hitchhiker's Guide to Evolutionary Computation." Online. Availible `http://www.etsimo.uniovi.es/pub/EC/FAQ/www`. 30 Sep. 1998.

[5] S. Hirano. "HORB: Distributed Execution of Java Programs." *Proc. WWCA '97, Lecture Notes in Computer Science.*, Vol. 1274 (1997): 13pp. Online. Availible `http://ring.etl.go.jp/openlab/horb`. 13 Sep. 1996.

[6] Mark Kantrowitz. "Frequently Asked Questions: Genetic Algorithms." Online. Availible `http://www.cs.cmu.edu/Groups/AI/html/faqs/ai/genetic/top.html`. 10 Aug. 1997.

[7] L.F.G. Sarmenta and S. Hirano. "Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java." *Proc. ACM Workshop on Java for High-Performance Network Computing.* (Palo Alto, 1998).