Waffle Mixer: A Library for Distributing Genetic Algorithms

Ariel Rideout
arideout@mit.edu

Ryan Williams
breath@mit.edu

David Ziegler
david@ziegler.ws

May 6, 2004

Abstract

We describe a toolkit that enables simple parallelization of genetic algorithms. Genetic algorithms are favored for optimization problems because they lend themselves to both continuous and discrete combinatorial problems and are less susceptible to becoming trapped at local optima in comparison to gradient search methods. Unfortunately, GAs are typically computationally expensive. The toolkit enables writers of genetic algorithm applications to choose between local (one machine) operation or distributed computation at runtime. The distributed approach achieves the same result or better than a local computation, but significantly diminishes the required computation time. An analysis of the system's performance is described.

1 Introduction

Genetic algorithms are one of many approaches used to solve optimization problems, such as the traveling salesman problem. In a genetic algorithm, strings of bits called *individuals*, are generated, each representing a solution to the problem. Individuals are evaluated, and the best individuals survive to the next generation. Individuals can *mate* to create *offspring*, new individuals that share characteristics of their parents.

Genetic algorithms are favored for many optimization problems because they are less susceptible to local optima. A genetic algorithm typically has an element of randomness built in, that allows the algorithm to work its way from such local optima. Although a genetic algorithm typically does not provide a deterministic optimal solution, the solution is often "good enough."

One difficulty with genetic algorithms is that for many problems, the performance of the algorithm is hampered by the time taken to evaluate an individual. In this paper, we present a toolkit, the Waffle Mixer, for distributing the computation of a genetic algorithm across a network of machines to speed computation. Such a distributed computation adds complexities; crossovers and mutations are based on both local individuals and individuals seen from the network.

The Waffle Mixer described addresses issues of network topology, host and individual naming, genome storage/announcement, and fault tolerance. The addition of new nodes to the network is done with a minimum of effect on other nodes, and every client, once it has joined, is able to contribute equally to the computation.

2 Related Work

There has been a good deal of research into the parallelization of genetic algorithms (GAs) [3]. Genetic algorithms offer the possibility of speeding computation of optimization problems signifi-

cantly when the optimal solution is not necessary, but often the cost of evaluation is a significant portion of the computation. Therefore, the common approach to parallel genetic algorithm (PGA) problems is to divide the computation such that the evaluation of individuals is spread out.

There are three prevailing methods of parallelization. The first is do perform global parallelization. In this form of PGA there is only one population of individuals, as in a typical GA. However, the evaluation of individuals is expressly parallelized. Selection is unmodified, and all individuals have the chance to mate with all others. This type of parallelization is often straightforward to implement, and can provide a significant speedup as long as the communication costs do not dominate the computation costs.

A coarse-grained PGA divides the individuals into several subpopulations, or demes. Each deme evolves individually from all others. To exchange information between demes, individuals occasionally *migrate* between demes. Coarse-grained PGAs are not as straightforward to implement as globally parallelized PGAs, as they fundamentally change the way in which the population(s) evolve.

The third mechanism for parallelizing GAs is fine-grained parallelism, where the number of demes is very large, and the size of each is quite small. The extreme (and ideal) case is to have precisely one individual for every processing element in the system, and therefore demes with only one individual, but this is not a strict requirement. This model is typically implemented on massively parallel computers, but can easily be adapted to any multiprocessing system.

2.1 Global Parallelization

In global parallelization, the distributed work is usually the evaluation of individuals, as this operation is very computationally expensive in most GAs. (Mutation and cross-breeding could also be farmed out.) Typically one master maintains the entire population of individuals and assigns slaves to evaluate some subset of the population. Once the fitness scores are returned, the master creates a new population and repeats the process.

Typically, globally parallelized PGAs are synchronous; that is, all individuals in the population are evaluated before the next generation is created. Asynchronous globally parallelized PGAs are also possible, but in practice have different characteristics than traditional GAs.

The communication overhead in globally parallelized PGAs can become a problem, especially in cases where the evaluation of one individual requires information about the entire population. Additionally, sometimes the chromosomes that comprise individuals are large, and the cost of transferring many individuals between computation nodes is prohibitive.

2.2 Coarse-Grained Parallelization

Coarse-grained parallelization is the most common approach to GA parallelization, and much research has been done. The approach is popular for several reasons. First, the conversion of a simple GA to a coarse-grained PGA is straightforward: run the original GA on many machines, and exchange individuals occasionally. Additionally, coarse-grained parallel computers are widely available, and can easily be simulated with available software, such as PVM[2] and MPI[1].

Importantly, the performance of coarse-grained PGAs is good. Research has shown that the parameters used to determine the rate of migration are critical. Specifically, with too low a rate of migration, the individual demes tend to be unaffected by arriving individuals migrating from other demes. Several different approaches to migrating individuals have been studied, and the

specifics used to determine how an individual chooses the deme to migrate to seem to be relatively unimportant; it is number of individuals that migrate and how often migrations occur that are the significant factors.

Some research into communication topologies has been done, and it appears that topologies with dense connectivity (short diameters) allow good individuals to propagate faster and produce better solutions than those with sparse connectivity (large diameters).

2.3 Fine-Grained Parallelization

Relatively little research into fine-grained parallelization has been done. The small research that has been performed has produced some interesting results. Notably, for fine-grained PGAs, it appears the the topology used is a significant factor in the performance of the PGA. Unfortunately, the specific problem the PGA is trying to solve seems to have an important effect on *which* topology is the best, leading to the need to adapt the topology to the specific problem being solved, once the problem has been thoroughly analyzed.

Between coarse- and fine-grained PGAs, it is unclear which is better performing, and at this point, it appears that no generalization may be made. The main reason behind this is that different applications may care about different measures of performance — most often either elapsed time to find a solution or quality of the best individual. Coarse- and fine-grained PGAs each excel in different applications.

For this reason, our toolkit has been designed to support either coarse- or fine-grained parallelization. An application writer can easily create a PGA with any granularity, and can tune the relevant parameters as desired. The mechanisms by which this occurs are discussed in section 5.

3 Genetic Algorithm Structure

Genetic algorithms are notable in their approach to optimization through their use of randomness to overcome local minima. For a problem that has an unknown solution space, genetic techniques can explore the space efficiently [4]. For a trivial problem such as finding the maximum of an unknown function, a genetic algorithm finds the peak through trying a variety of random values, then hill-climbing to the peaks.

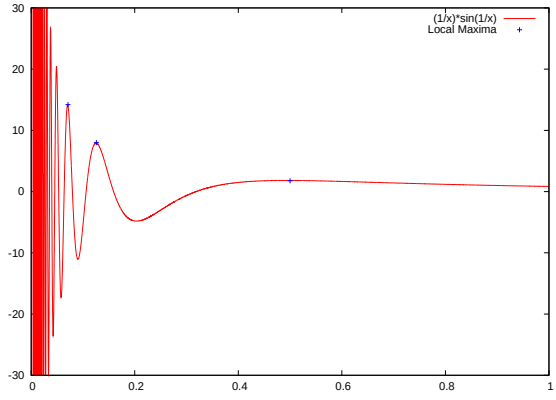


Figure 1: Function $\frac{\sin 1/x}{x}$ and some maxima

In Figure 1 note that there are several local maxima. The GA approach to find the highest peak tries a wide variety of values in the neighborhood of the local peaks as well as random points all over the map. The points in the local neighborhood serve to get closer to the local maximum (in case it is also the global maximum). The random points serve to find new maxima.

3.1 Generation

Typical genetic algorithms operate on generations of individuals; a population of individuals competing against each other. The creation of a new generation of individuals that is likely to outperform the previous generation requires a set of tricks. The top-performing individuals are often preserved intact, on the chance that they will remain so for a few generations; this ensures that the best score is monotonically increasing. The remainder of the individuals who are neither best nor worst will be mutated, combined, or otherwise mangled in an attempt to keep their good aspects and improve their weak aspects. Usually a few random individuals are created in order to spread the search space out.

3.2 Scores

To choose which individuals should survive, some objective measure of their quality must be determined. Most genetic algorithms rely on an absolute score that measures the quality of the individual. Some problems allow only the comparison of one individual to another, to determine which of the two is better. This trait makes it difficult to establish an ordering on individuals, as the ordering may be intransitive (for example, rock, paper, scissors). For this reason, our toolkit requires that individuals must have an absolute score.

For those problems that are scorable, the ease of scoring an individual varies. Some problems are quite simple to score such as curve-fitting, while others such as walking are more difficult. For harder problems, the individual evaluation stage is typically computationally intensive, representing a long simulation of each individual.

It is also conceivable that each iteration represents a timestep in some long-running simulation. This model lends itself easily to implicit scoring systems where an individual's score is merely the measure of time it existed before being killed. Since there is little change between cycles, there is much less turnover in the population of individuals per cycle.

4 Design

An application using our toolkit instantiates an object which spawns a network thread. The constructor of the object takes in either a hostname, in which case it attempts to join a network through the specified host, or nothing, in which case it waits for incoming connections from other hosts. After initializing the network thread, the application has access to input and output queues. The network thread will keep the input queue full and the output queue empty to the best of its ability.

When an application wants a individual from the network, it pulls it from the head of the input queue. It may then take any action it desires — we leave that to the application writer. We envision applications using pulled individuals for both mutation and mating.

Of course, it is possible that the input queue is empty; no new individuals have been received from the network. In this case, the application must determine the appropriate behavior; the

network does not automatically generate or mutate new individuals. In many applications, falling back to the single-threaded model of computation may be sufficient; mutate and mate current individuals to produce a new population. Because the appropriate behavior may vary depending on the application, the toolkit does not impose any style of individual generation on the application.

When an application has produced and evaluated a new individual, it may push it to the output queue. The toolkit determines what should be done with the individual; whether it is good enough to announce to the network. This occurs transparently to the application.

The network thread maintains the input queue from which the application will pull, and the output queue with generated individuals. From the network thread's perspective, these are not queues, but rather lists of individuals sorted by their scores.

When an advertisement is heard, the network thread checks whether it wants to insert it into the queue. If the network thread does want to insert the individual, it contacts the node that advertised the individual and requests it directly. The advertiser transmits the individual, or replies that it is no longer available. Because nodes only remember a finite number of their best outgoing individuals, the advertiser may no longer have that individual available. Once the requester receives the individual, it adds it to the queue in sorted order, and removes the worst individual if the queue has grown too long. This ensures that the best individuals are always available to the application.

Similarly, when an individual is pushed to the output queue, the network thread decides if it is good enough to be transmitted. If there is an open space in the output queue, or if the individual has a higher fitness than any individual in the output queue, an advertisement is transmitted. Later, when a request for the individual is received, the node transmits it if it is still in the output queue.

4.1 Network

The design of the network topology is critical to the scalability of the application. With the goal of scaling to large network sizes, a single managing node is insufficient. Therefore, the network used must be self-organizing — any node in the system should be able to provide instructions to an arriving node or a departing node. Additionally, as the network grows, symmetry is a desired trait. A node that joins at the beginning of the computation should have no greater duty than a node that joins later on.

Our network topology is based on HyperCuP[5, 6], a system for organizing a peer-to-peer network. This system provides a set of protocols for a node joining a network, departing a network, and failing. Importantly, it provides strong expectations on network properties. The characteristic path length is approximately $\frac{1}{2}\log_2 n$, where n is the number of nodes in the network. Broadcast messages may be transmitted with no message overhead, reaching all nodes in $n-1$ messages, without any need for path vectors. The network is also very resilient to node failures; it is difficult to partition the network or disconnect a node, without many nodes failing.

At a high-level, the network is organized as a hypercube, where nodes are vertices in the hypercube. The edges between nodes are labeled according to a set of rules, allowing efficient communication. All nodes have the same role in the network; there are no "super nodes" that are responsible for managing the roles of other nodes after they join. The interested reader is directed to the cited works for further details.

Although the hypercube topology was chosen for the network, it is not critical that such a topology is chosen. Other topologies, such as a random graph, could also work well. The hypercube topology provides uncommon features that are a benefit, particularly the zero overhead of

broadcasting messages. This allows for very simple routing of messages without network overhead.

The code managing the network is designed to be independent from the application-layer code above it (in this case, the genetic algorithm code). The library has a simple interface that provides the necessary actions: join a network (given a node to contact), leave the network, transmit a message (with different options for whom it is transferred to), and receive a message.

5 Implementation

The implementation of the Waffle Mixer was done in Java, due to the availability of example code, ease of memory management, and the new non-blocking input/output library.

5.1 Application Interface

The application writer using our toolkit is offered a simple bidirectional queue interface, represented by a DStream object. The DStream, once created, handles the network setup and connections to other hosts without further intervention from the application writer.

The only restriction the DStream places on the writer of the application is that the objects to be passed over the network must implement the Waffle interface, which contains a `score()` and `marshall()` method. This interface was chosen to allow the application to efficiently compress its data representation for network transport. Marshalling/unmarshalling methods are also useful to the application writer because they aid in debugging, and enable system state to be saved to and restored from files. These methods are typically very easy to write, since the data representation format is fully known to the application writer.

At the moment these interfaces do not offer the application writer the ability to tweak the parameters of the network; this is a feature under discussion.

6 Experiments

We performed several experiments to examine whether the toolkit provided any benefit to the process of evolution. In order to be judged worthwhile, our toolkit must perform better than a single machine running for a long time (or a very fast single computer), or many machines running in parallel but not communicating over the network.

Of the several programs we wrote to test the functionality of the Waffle Mixer, we elected to use the Robot Arm program as the basis for most of our testing. This genetic algorithm attempted to find a sequence of joint angles for a simulated robot arm to touch a goal without colliding with itself or various walls. Other programs we tested included a function maximizer, which was too quick to find a solution to be of use, and an image matcher, which was too slow to render.

6.1 The Robot Arm Problem

The robot arm is a many-jointed two-dimensional appendage with various constraints. Each of its joints is limited to rotating $\pm 90^\circ$ relative to its predecessor. The arm's origin is fixed at the coordinate $(0, 0)$, though its rotation about that point is unconstrained. Though the number of links and their lengths was a configurable parameter, for all of the tests we computed here we elected to use an arm with 16 links of half a unit in length each. It is not allowed to touch the

walls in any way — the arm's motion was simulated by the evaluation routine and any collisions are immediately penalized. The arm can very easily touch itself (by bending three or more joints in the same direction) so we early on made the decision to penalize the arm for self-collision.

The goal of the arm is to touch one of the goal lines set up in the playing area without touching either itself or any of the walls. The way we encouraged this behavior was through giving higher fitnesses to arm individuals that did not touch any walls and which got closer to the goal lines. Setting up the playing field such that this task is neither too easy nor too hard is often a matter of guesswork.

The fitness score of each individual falls into one of three ranges:

score < 0**:**
: Individuals that hit walls are scored below all others, ordered by how long they lived before colliding. $\left[\frac{-1}{time}\right]$.

$0 \leq$ **score** $<$ **length of arm:**
: Individuals that run out of data without reaching the goal are scored on their final distance. $[\max(\text{length of arm} - \text{distance to goal}, 0)]$

score $\geq$ **length of arm:**
: Individuals that reach the goal are scored above all others, ordered by how quickly they reached the goal. $\left[\frac{1}{\text{size(arm)+time}} + \text{length of arm}\right]$.

6.2 Visuals

To aid in the development, we wrote a companion program that graphically displayed the progression of winning individuals over time. Often this would reveal errors in our assumptions or "illegal shortcuts" that the individuals employed to get higher scores than they really should have. A popular ploy was to whip the tip of the arm around so fast that the collision detection algorithm would miss its collisions.

The scene depicted below is one we called "the lobster trap," and it is very interesting because of the contortions the arms go through to solve it. This scene was a difficult problem for the naive genetic algorithms we wrote. The arm tends to get stuck in one of three places, illustrated in Figure 2. Most of these places represent equipotential fitness lines. Since there is no fitness pressure for the arm to travel towards the lip, only randomness can help the arm into the hole or around the lip. Our fitness evaluation function could clearly benefit from more complexity; writing a good evaluator is a time-consuming art requiring some expertise. However, enough randomness can eventually overcome a mediocre fitness evaluator.

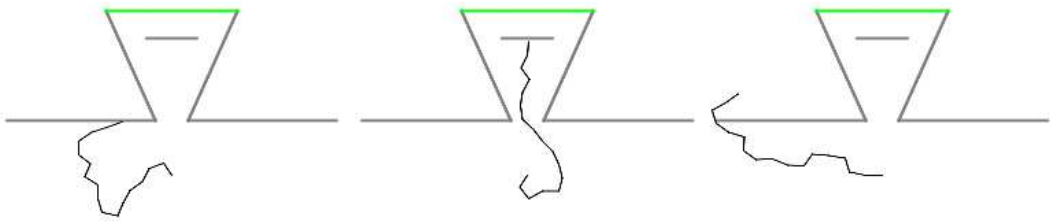
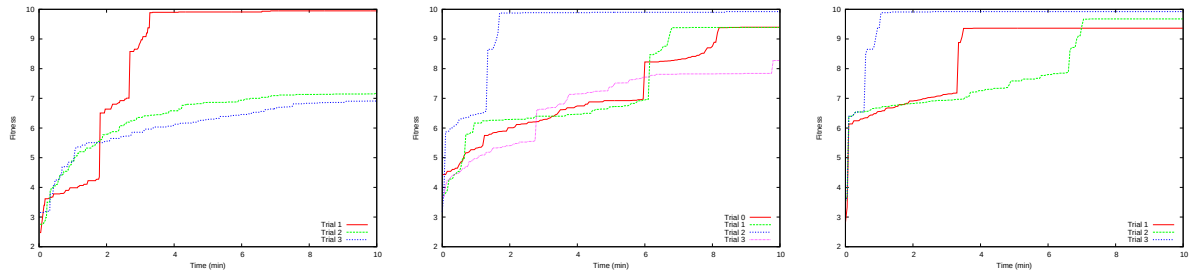


Figure 2: Arms getting stuck

The last stuck arm is particularly pernicious, as it represents a very strong peak in the solution space. We did not anticipate this particular configuration when designing this scene (we would have made the horizontal walls longer), and it is interesting because of our oversight. The framework we have is not especially good at overcoming these obstacles since each individual is a careful series of positions and it is highly unlikely that a random configuration will top the "wrap around" configuration.

6.3 Experiments

The purpose of our experiments was to compare the performance of a genetic algorithm problem using our distributed Mixer against the same problem without the Mixer. The Mixer allows one to increase the amount of computing power devoted to a particular problem, but is it better for the task than simply running the problem on a more powerful CPU? Perhaps running the problem on N computers independently and collecting the best result afterwards will be equivalent or better than involving the network as the Mixer does. Our results indicate that using the Waffle Mixer to share individuals across the network caused the problem to be solved more efficiently than a single powerful computer or many computers computing without communication.



(a) **(b)** **(c)**

Figure 3: Individual trials of the three computation methods

Single CPU:

We simulated an extremely fast computer by running our trials on a single computer for a longer period of time, about 25 minutes. Since the length of time it took to evaluate an individual varied depending on the individual's complexity, we took wall time as the x axis and plotted fitness against it, leading to Figure 6.3a.

Parallel computers, no Mixer:

Our next experiment involved running ten computers with the simulation. We selected the best individual amongst the ten computers for each time in order to plot a single line for each trial in Figure 6.3b.

Parallel computers with the Mixer:

Our last experiment ran the simulation on ten computers that were all collaborating with each other through the Waffle Mixer. The best individuals were collected at a central location for each time. All of the trials are plotted in Figure 6.3c.

Note that in all three graphs there is a lucky trial that succeeds far sooner than all the other trials. We expect this to happen because of the randomness of the algorithm.

6.3.1 Final Comparison

We averaged the results from the trials of each method in order to generate a graph representing the statistically expected results, Figure 4.

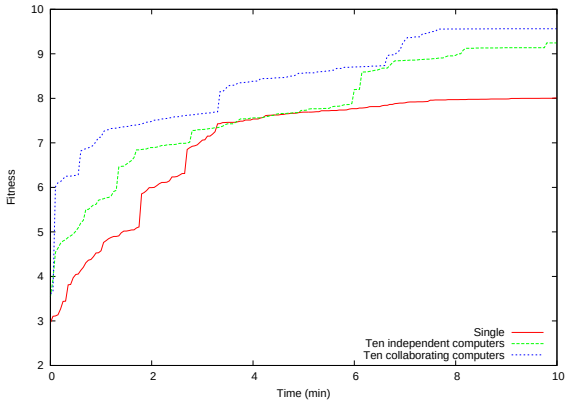


Figure 4: Comparison of the three computation methods

In Figure 4 we see that the average score of the best individual computed with the Waffle Mixer method is always superior to the best individuals computed using the other methods. It is interesting to note that the addition of computers does not result in a linear speedup of the time required to pass a given mark. The following table reports the time at which each of the three methods reached a fitness of 8 (which corresponds to touching the goal, a plausible ending point).

Single	**Independent**	**Mixer**
9.7m	6m	3.35m

This is not to imply that the Waffle Mixer is always three times as good as a single CPU. On the average however, it clearly offers an advantage.

7 Conclusion

We have provided a complete toolkit for easily parallelizing genetic applications. Most importantly, the waffle mixer allows for a wide variety of parallel genetic algorithm implementation styles. The application writer is left in control of mutating individuals to create new generations, pulling individuals from the network, and determining the structure of the code.

The toolkit allows for very robust networks, and eliminates any single point of failure. Importantly, there is no node responsible for managing the responsibilities of other nodes. The topology used ensures that the network of nodes can grow to be very large.

We have additionally demonstrated the the use of the toolkit provides speed advantages over single-threaded genetic algorithms. A network of machines communicating using the waffle mixer typically arrives at a better solution than the same number of machines computing without communication for the same amount of time, or one machine computing for longer.

References

[1] Mpi — the message passing interface standard. `http://www-unix.mcs.anl.gov/mpi/`.

[2] Pvm: Parallel virtual machine. `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[3] Cantu-Paz, E. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis 10*, 2 (1998), 141–171.

[4] Holland, J. *Adaptation In Natural and Artificial Systems.* MIT Press, 1975.

[5] Schlosser, M., Sintek, M., Decker, S., and Nejdl, W. Hypercup — hypercubes, ontologies, and efficient search on p2p networks. *1st Workshop on Agents and P2P Computing* (2002).

[6] Schlosser, M., Sintek, M., Decker, S., and Nejdl, W. Hypercup — shaping up peer-to-peer networks.