Implementing Fast and Flexible Parallel Genetic Algorithms

Erick Cantú-Paz

August 1997
Revised in August 1998

To be published in *Handbook of Practical Genetic Algorithms.*
Volume 3.

Illinois Genetic Algorithms Laboratory
University of Illinois at Urbana-Champaign
117 Transportation Building
104 S. Mathews Avenue Urbana, IL 61801
Office: (217) 333-0897
Fax: (217) 244-5705

Implementing Fast and Flexible Parallel Genetic Algorithms

Erick Cantú-Paz
Department of Computer Science and
Illinois Genetic Algorithms Laboratory
University of Illinois at Urbana-Champaign
cantupaz@illigal.ge.uiuc.edu

Abstract

As genetic algorithms (GAs) are used to solve harder problems, it is becoming necessary to use better algorithms and more efficient implementations to reach good solutions fast. This chapter describes the implementation of master-slave and multiple-population parallel GAs. The goal of the chapter is to help others to implement their own parallel codes. To this effect, the text discusses some of the design decisions that were made and possible improvements to the code.

1 Introduction

Genetic algorithms (GAs) are making their way from universities and research centers into commercial and industrial settings. In both academia and industry, genetic algorithms are being used to find solutions to harder problems, and it is becoming necessary to use improved algorithms and faster implementations to obtain good solutions in reasonable amounts of time. Fortunately, parallel computers are making a similar move into industry, and GAs are very suitable to be implemented on parallel platforms. The goal of this chapter is to help developers to implement their own parallel GAs. The chapter explains some of the design decisions that were made, and it discusses a few ideas to optimize the code.

The code described here is designed to be used in a research environment, where both flexibility and efficiency are indispensable to experiment. New features can be added easily as the research requires, and special care was

taken to maintain efficiency because experiments must be repeated multiple times to obtain statistically significant results. This quest for flexibility and efficiency should also be appealing to practitioners outside academia who want to use a parallel GAs on their applications.

In the design of software there is usually a tradeoff between making programs very flexible or making them efficient and easy to use. In our design we tried to make this tradeoff as small as possible by incorporating enough flexibility to enable experiments with many configurations, but at the same time trying to keep the code as simple as possible to maximize its efficiency and its usability. In our view, it was pointless to implement a system loaded with features and options, because it would have made the programs too complicated to understand and execute.

Another important concern was portability because access to parallel computers changes over time. For this reason, PVM was chosen to manage processes and the communications between them. PVM is a message-passing parallel programming environment that is available on many commercial (and experimental) parallel computers. The code was implemented in C++, and the design follows an object oriented methodology. The choice of language allows a fast execution and the object-oriented design facilitates maintenance and extensions.

The remainder of this chapter is organized as follows. The next section presents some background information on GAs, parallel computers, and different kinds of parallel GAs. Section 3 contains a detailed description of some critical portions of the software. Section 4 presents the results of tests that show the efficiency of the parallel GA working on different problems. Finally, this chapter ends with a summary and a discussion of possible extensions to the system.

2 GAs and Parallel Computers

How do GAs relate with parallel computers? A very short, but true, answer would be that they get along very well. A long answer would look into the history of GAs and parallel computers and explore their relationship across several decades. However, it is not the intention of this chapter to review the literature on parallel GAs (an interested reader can refer to the survey by Cantú-Paz (1998b)). Instead, this section focuses on the present and the future of parallel computers and on how GAs relate to them.

It is difficult to discuss parallel computers in general when one considers their rich history and the variety of architectures that have appeared over

time. However, there is a clear trend in parallel computing to build systems from components that resemble complete computers interconnected with a fast network. More and more often, the nodes in parallel computers consist on off-the-shelf microprocessors, memory, and a network interface. Parallel and supercomputer manufacturers are facing a reducing market for their computers and commercial success is more difficult. This makes it almost impossible to compete successfully using custom components, because the cost of designing and producing them cannot be amortized with the sell of a few machines.

The trend to build parallel machines as networks of essentially complete computers immediately brings to mind *coarse-grained* parallelism, where there is infrequent communication between nodes with high computational capabilities. There are two kinds of parallel GAs that can exploit modern coarse-grained architectures very efficiently: multiple-population GAs (also called coarse-grained or island model GAs) and master-slave (or global) parallel GAs. We shall review these algorithms in detail in later sections.

There is a third kind of parallel GA that is suitable for fine-grained massively parallel computers, but because these machines are not very popular and it is highly unlikely that they will be in the near future, we shall not discuss fine-grained parallel GAs any further. Instead, the presentation shifts to the problem of tuning GAs to particular domains.

GAs are very complex algorithms that are controlled by many parameters, and their success largely depends on setting these parameters correctly. There is no single set of parameter values that results in an effective and efficient search in all cases, and the fine-tuning of GAs to particular domains is still viewed as a black art by many. But, in reality we know a great deal about adequate values for most of the parameters of a simple GA. For example, a critical factor for the success of a simple GA is the size of the population, and there are accurate models that relate the problem difficulty and population size with the expected quality of the solutions (Harik, Cantú-Paz, Goldberg, & Miller, 1997).

Another critical factor for the success of GAs is the balance between crossover and selection (Thierens & Goldberg, 1993; Goldberg & Deb, 1991). If selection is too intense the population would converge very fast and crossover might not have enough time to build good solutions. When this premature convergence occurs the GA may converge to a suboptimal population. On the other hand, if the selection intensity is too low, crossover might disrupt any good solutions that may have been found, but that have not had time to reproduce.

The problem of choosing adequate parameter values is worse in multiple-

population parallel GAs, because they have more parameters than simple GAs that control their operation. Besides deciding on all the GA parameters one has to decide on migration rates, subpopulation sizes, interconnection topologies, and a migration schedule.

The research on parallel GAs spans several decades, but we are still a long way from understanding the effects of the multiple parameters that control them. For example, should high or low migration rates be used? What are low and high migration rates? What is an adequate way to connect the subpopulations on a parallel GA? How many subpopulations and of what size should be used? Studies are underway to answer at least some of these questions and to develop simple models that may help practitioners through this labyrinth of options.

The next two sections review some important aspects of two kinds of parallel GAs that are suitable for the architecture of current parallel computers. We review first the simpler master-slave GA and then proceed to examine multiple-population parallel GAs.

2.1 Master-Slave Parallel GAs

Master-slave GAs are probably the simplest type of parallel GAs and their implementation is very straightforward. Essentially, they are a simple GA that distributes the evaluation of the population among several processors. The process that stores the population and executes the GA is the master, and the processes that evaluate the population are the slaves (see figure 1).

Just like in the serial GA, each individual competes with all the other individuals in the population and also has a chance of mating with any other. In other words, selection and mating are still *global*, and thus master-slave GAs are also sometimes known as global parallel GAs.

The evaluation of the individuals is parallelized by assigning a fraction of the population to each of the slaves. The number of individuals assigned to any processor can be constant, but in some cases (like in a multiuser environment where the utilization of processors is variable) it may be necessary to balance the computational load among the processors using a dynamic scheduling algorithm (e.g., guided self-scheduling).

Regardless of the distribution strategy (constant or variable) if the algorithm stops and waits to receive the fitness values for all the population before proceeding into the next generation, then the algorithm is *synchronous*. A synchronous master-slave GA searches the space in exactly the same manner as a simple GA. However, it is also possible to implement an *asynchronous* master-slave GA where the algorithm does not wait for

any slow processors. Slaves receive individuals and send their fitness values at any time, so there is no clear division between generations. Obviously, asynchronous master-slave GAs do not work exactly like a simple GA. Many global parallel GA implementations are synchronous, because they are easier to implement, but asynchronous GAs might exploit better any computing resources that might be available.

Master-slave GAs were first proposed by Grefenstette (1981), but they have not been used too extensively. Probably the major reason is that there is very frequent interprocessor communication, and it is likely that the parallel GA will be more efficient than a serial GA only in problems that require considerable amounts of computation. However, there have been very successful applications of master-slave GAs like the work of Fogarty and Huang (1991), where the GA evolves a set of rules for a pole balancing application. The fitness evaluation uses a considerable amount of computation because it requires to simulate a cart with a pole attached to its top with a hinge. The cart moves back and forth on a straight line, and the goal is to make the pole stand straight. Other successful applications of master-slave GAs are the works of Abramson and Abela (1992) and Abramson, Mills, and Perkins (1993) on finding schedules for schools and trains. Hauser and Männer (1994) shows successful implementations of a master-slave GA on three different computers.

As more slaves are used, each of them has to evaluate a smaller fraction of the population and therefore the computation time is reduced. However, it is important to note that the performance does not improve indefinitely, and adding slaves after a certain point can slow down the algorithm. The reason is that as more slaves are used the time used to communicate information between processes increases, and it may become large enough to offset any gains that come from dividing the fitness evaluations. There is a recent theoretical model that predicts the number of slaves that maximizes the parallel speedup of master-slave GAs depending on the particular system used to implement them (Cantú-Paz, 1998a).

It is also possible to parallelize other aspects of GAs besides the evaluation of individuals. For example, crossover and mutation could be parallelized using the same idea of partitioning the population and distributing the work among multiple processors. However, these operators are so simple that it is very likely that the time required to send individuals back and forth will offset any possible performance gains. The communication overhead is also a problem when selection is parallelized because several forms of selection need information about the *entire* population and thus require some communication.

One straightforward application of master-slave GAs is to aid in the search for suitable parameter values to solve a particular problem. Even though there is enough theory to guide users to choose parameter values, it is still necessary to refine the parameters by hand. A common way to fine tune a GA empirically is to run it several times with a scaled-down version of the problem to experiment and find appropriate values for all the parameters. The parameters that give the best results are then scaled to the full-size problem and production runs are executed. With a master-slave GA both the experimental and the production runs should be faster and there can be considerable savings of time when solving a problem.

In conclusion, master-slave parallel GAs are easy to implement and it can be a very efficient method of parallelization when the fitness evaluation needs considerable computations. Besides, the method has the advantage of not changing the search strategy of the simple GA, thus all the theory available for simple GAs can be applied directly.

2.2 Multiple-Population Parallel GAs

The second type of parallel GAs which are suitable to coarse-grained computer architectures are multiple-population GAs. Multiple-population GAs consist on a few subpopulations that exchange individuals infrequently (see figure 2). This is probably the most popular type of parallel GAs, but a complete understanding of the effect of their parameters on the quality and speed of the search still eludes us. However, there have been recent advances to determine the size and the number of subpopulations that are needed to find solutions of a certain quality in some extreme cases of parallel GAs. It is well known that the size of the population is critical to find solutions of high quality (Harik, Cantú-Paz, Goldberg, & Miller, 1997; Goldberg, Deb, & Clark, 1992) and it is also a major factor in the time that the GA takes to converge (Goldberg & Deb, 1991), so a theory of population sizing is very useful to practitioners.

It has been determined theoretically that as more subpopulations (or *demes*, as they are usually called) are used, their size can be reduced without sacrificing the quality of the search (Cantú-Paz & Goldberg, 1997a). Assuming that each deme executes on a node of a parallel computer, smaller demes directly reduce the time dedicated to computations. However, using more demes also increases the communications in the system. This tradeoff between savings in computation time and increasing communication time entails the existence of an optimal number of demes (and an associated deme size) that minimizes the total execution time (Cantú-Paz & Goldberg,

1997b). The deme sizing theory and the predictions of the parallel speedups have been validated using the program described in this contribution.

One of the major unresolved problems in multiple-deme parallel GAs is to understand the role of the exchange of individuals. This exchange is called *migration* and is controlled by: (1) a migration rate that determines how many individuals migrate each time, (2) a migration schedule that determined when migrations occur, (3) the topology of the connections between the demes, and (4) a method to choose the emigrants and how they replace individuals at the receiving deme.

Migration affects the quality of the search and the efficiency of the algorithm in several ways. For instance, frequent migration results in a massive exchange of potentially useful genetic material, but it also affects the performance negatively because communications are expensive. Something similar occurs with densely-connected topologies where each deme communicates with many others. The ultimate objective of parallel GAs is to find good solutions fast, and therefore it is necessary to find a balance between the cost of using migration and increasing the chances of finding good solutions.

The program was designed to experiment with multiple-deme GAs, so most of the discussion in later sections centers on the features that permit to experiment with this type of algorithms. The description of some critical parts of the implementation of parallel GAs begins in the next section.

3 Implementation

As was mentioned in the introduction, the parallel GA is written in C++ and it has an object-oriented design. This section first discusses the overall design of the program and then gives a detailed description of the most important objects in the system.

Object-oriented design is well suited for genetic algorithms. Many elements of GAs have the three main elements that characterize an object: a defined state, a well-determined behavior, and an identity (Booch, 1994). For example, a population consists—among a few other things—on a collection of certain individuals (state), it has some methods to gather statistics about them (behavior), and it can be differentiated from all the other objects in the program (identity). An individual is also a well-defined object. Its state is determined by the contents of a string and a fitness value, it can be identified precisely, and it has a very well-defined behavior when it interacts with other objects.

There are many ways to design an object oriented GA and there are

many programming styles. For example, one could implement all the genetic operators as methods of the individuals and use a population object to control the global operations like selection. Another option is to design individuals and populations as container objects with a very simple behavior, and to implement the GA operations at a higher level. The latter was the approach used in the current implementation.

The general idea in the design is to view the simple GA as a basic building block that can be extended and used as a component in a parallel GA. After all, the behavior of the GA is very similar in the serial and the parallel cases. In particular, to make a master-slave parallel GA it is only necessary to modify the evaluation procedure of a simple GA. Also, a simple GA can be used easily as a building block of coarse-grained parallel GAs, because it is only necessary to add communication functions to turn it into a deme.

As was mentioned before, the design of the system emphasizes simplicity, so that the code could be modified easily. To simplify the code, many of the supporting activities (like memory allocation and management) were hidden in the `Individual` and `Population` classes. This makes the code related with the GA and the parallel GA much more readable and easy to modify.

Another aspect of interest during the implementation of the code was portability. The language that was used (C++) is available on most computers today and all the parallel programming aspects were handled with PVM, which is available in commercial and free versions for many platforms.

PVM implements a Parallel Virtual Machine (hence its name) from a collection of (possible heterogeneous) computers. It can be used to simulate parallel computers, but it can also be used as a programming environment on top of an existing parallel machine. PVM provides a uniform message-passing interface to create and communicate processes making the hardware invisible to the programmer. This last feature enables the same parallel program to be executed efficiently on a multitude of parallel computers without modification.

3.1 Simple Genetic Algorithms

The first object that we shall discuss in detail is the class `GA`. It implements a simple genetic algorithm and it is the base class for `GlobalGA` and `Deme`, which we shall treat later. The class `GA` uses the class `Population` to store and obtain statistics about its population. The `GA` class can be configured to use different selection methods (currently roulette wheel and tournament selection of any size are implemented) and crossover operators (uniform and

multiple-point).

The main loop of the GA is in the method `run()`, which simply executes the method `generate()` until the termination condition is met. We normally run a GA until its population has converged completely to a single individual, but the termination condition can be changed easily to a fixed number of generations or evaluations, for example, by modifying `GA::done()`.

A single generation of the simple GA consists on evaluating the population, generating and reporting statistics on the state of the GA, selecting the parents, and applying crossover and mutation to create the next generation. The code for `GA::generate()` is in figure 3.

The GA reports some statistics about the progress of the run every generation and it can be instructive to see how the population diversity, its average, or its variance change over time. Many times in research we use artificial test functions and we know what are the building blocks that are needed to find the global solution. To aid in our research, the program can also report statistics about the average and variance of the building blocks in the population.

Some of the methods of this class are implemented as virtual functions. This is to allow the classes that are derived from `GA` to extend some of the basic functionality that is provides. For example, there is a `ConstrainedCostGA` class that interrupts the run of the GA when a certain number of function evaluations have been executed, even if the GA is in the middle of a generation. The purpose of this class is to be used in time constrained applications, where there is a fixed amount of time available to find a solution. Most of the virtual functions are overridden by the classes that implement the parallel GAs and will be discussed in later sections.

How could the GA code be made more efficient? The GA operators are very simple and there is no need to optimize them to squeeze a one percent improvement in the performance. The supporting classes (`Individual` and `Population`) are also very simple and it is unlikely that any change in them will reduce the execution time. Probably the only optimization performed on this code was to avoid copying memory unnecessarily. In particular, the GA class has a pointer to the current population, so when a new temporary population of individuals is created only a pointer needs to be updated.

3.2 Master-Slave Parallel GAs

The class `GlobalGA` implements a master-slave (or global) GA. Recall that a master-slave GAs is identical to a simple GA, except that the evaluation of the individuals is distributed to several slave processes. `GlobalGA` is the

class that defines the master process and it inherits all the functionality of a simple genetic algorithm from class `GA`. The main differences between these two classes are on their initialization and on the evaluation of the individuals. This section describes those differences in detail, but first it briefly discusses the slaves processes.

The slaves are implemented by the class `Slave`, which basically executes a loop with three operations: (1) receive some number of strings that represent individuals, (2) evaluate them using the user-defined fitness function, and (3) returns the fitness values to the master. When a `GlobalGA` is created it launches the slave processes using PVM. Launching processes requires a considerable amount of time and, to avoid excessive delays, slaves are created only once at the beginning of a series of experiments. The `GlobalGA` keeps track of the number of repetitions and instructs the slaves to terminate when the series of runs is completed. The code for the main loop of the class (`GlobalGA::run()`) is in figure 4.

The most interesting part of the `GlobalGA` is the `evaluate()` method. It sends the individuals to the slaves and waits for the results to come back. This is a straightforward procedure, but here we had to make an important decision about the size of the messages used to send the individuals to the slaves and collect the results. In general, there are two options when one needs to send information across a network: (1) use many small messages, or (2) use a few large messages. In the first case, each individual would be sent in a separate message; and in the second case all the individuals assigned to a slave would be sent in one message. Usually, the best alternative is to use few large messages and only this option is implemented in the current version (the first alternative was implemented in an earlier version, but the tests on different platforms confirmed that it was very inefficient).

To understand why few large messages are usually preferable we have to realize that the time needed to send a message across a network has two components. The first is an overhead from the operating system and the message-passing interface each time a message is sent. This overhead is called latency and is independent of the size of the message. The second component of the communication time depends on the bandwidth of the network (a measure of the amount of information that the network can transmit in a unit of time) and the size of the message. Overall, the time needed to send a message of certain size across a network is: $T = Bsize + L$ where B is the inverse of the bandwidth and L is the latency. For example, if we need to communicate x bytes using N messages the total time would be $B\frac{x}{N} = Bx + LN$, but if we only use one message the total time is only

[1] $Bx + L$.

Avoiding excessive overhead by using fewer messages is not the only way to improve performance of this class of parallel GAs. In the current implementation, the master process sits idle as the slaves evaluate the population. A simple—and potentially effective optimization—is to change the code so that the master process also evaluates a few individuals. Although this involves rather trivial changes in `GlobalGA::evaluate()`, the code distributed with this chapter was not changed for the sake of clarity. Besides, it is possible to instruct PVM to spawn a slave on the same node that the master is running, and in this way the processor will not remain idle. Our tests confirm that the performance is not affected negatively by having an extra slave perform the work that the master could do, and the code remains remarkably simple (see figure 5 for the complete code of `GlobalGA::evaluate()`).

3.3 Multiple-Deme Parallel GAs

This type of parallel GAs is more complex than the master-slave type and its implementation uses two classes: `PGA` and `Deme`. The class `PGA` creates, initializes, and collects results from a set of `Demes`. Most of the interesting (and complicated) aspects of the coarse-grained GA are implemented in the class `Deme`, so this section focuses on it.

The class `Deme` is derived from the class `GA`, and it adds the communication capabilities to the simple GA. The most important difference between these two classes is that after each generation a deme checks to see if it is time to migrate. If migration does not occur, then the execution proceeds as usual. But, if migration is due then the deme sends a predetermined number of individuals to each of its neighbors. Then, the deme waits to receive the migrants from its neighbors and incorporates them into its population.

The main cycle for a deme is straightforward: execute for a few generations, send some individuals, receive others, incorporate the newly arrived, and execute again (see figure 6). The complications are not at this (high) level, but in the particular workings of each of these steps. The remainder of the section looks at each step in detail and explains some of the options that designers have to face.

First, there has to be a schedule for migrations: How often should migrations occur? In the current implementation the user specifies an interval

[1] Of course, this is just a coarse example and it may be possible to send the same amount of information faster using N messages in a system with a small latency and low bandwidth.

of generations between migrations. To implement this requires to modify the main loop of the simple GA to check every generation if it is time to send and receive individuals. There is also an option to migrate only after the population has converged completely, and in our tests the quality of the solutions is similar as with frequent migrations. Others have obtained the same results empirically (Grosso, 1985; Braun, 1990; Munetomo, Takai, & Sato, 1993).

We must also decide which individuals migrate. The number of migrants is determined by the user, but which individuals should be sent away? Migrants could be chosen at random or preference could be given to the best individuals of the current generation. Selecting randomly has the advantage of disseminating more diversity and the chances of exploring new regions of the search space may be improved. On the other hand, selecting the best individuals may help disseminate genetic material that has already been tested and shown to be useful. Because the two options may be beneficial the program implements both.

The topology used by the demes to communicate is also specified by the user as a parameter to the program, and its implementation is probably the most interesting part of the class `Deme`. In theory, any arbitrary topology can be used, but there are some patterns that are very common and are already implemented: uni- and bi-directional rings, hypercubes of varying dimensions, rectangular grids of any size, stars, fully-connected, and isolated. The class `PGA` creates and sends to each deme the logical links that specify the topology. The demes only receive the links that correspond to them, so they know where to send migrants and from where to receive them, but they have no knowledge about the global topology. For example, in a ring topology each deme receives only one incoming link (to receive migrants from the deme behind) and one outgoing link (to send migrants to the deme ahead).

Each link could have a different migration rate associated to it, but since it is not clear that this feature is very useful, only a uniform migration rate is supported. However, it would not be too difficult to implement this option in the future.

A common problem in the design of concurrent systems is to avoid deadlocks. One form of deadlock occurs when one process is blocked waiting to receive data from another, which is itself blocked waiting to receive data from the first. It is very complicated to detect deadlocks and correct them as the system is running, and so it is much better to avoid them altogether. In the parallel GA there is a possibility of a deadlock during migration, because one deme may be blocked waiting to receive individuals from another

deme that is also blocked. One way to avoid deadlocks is to use non-blocking send commands, and send all the migrants before attempting to receive any. With non-blocking sends a process that initiates the communication does not have to wait until the recipient finishes reading data from the communication channel. The data is stored in buffers that can be read at the recipient's convenience.

When migrants arrive they are stored in a temporary `Population` object before they are incorporated into the current population. There are several alternatives to incorporate migrants and there are two implemented in the system: random replacement of current members by migrants or a form of elitist or competitive replacement. In the latter method, the newly arrived compete in tournaments with randomly selected members of the current population and the winner stays in the population. In our research we only use random selection and random replacement of migrants, but the other options could be very useful in real-life optimization problems.

Another interesting part of the implementation is concerned with the termination of the algorithm. When the termination condition becomes true in a deme it cannot simply send its best individual to the process that collects solutions and disappear, because other demes might attempt to communicate with it. Sending a message to a nonexistent process is not really a problem, because PVM would signal the error and discard the information. The real issue is that a process would be locked forever it it waits to receive a message from a deme that has already terminated. This is another form of deadlock problem and it is also very difficult to detect and solve at run time.

To avoid this form of deadlock, each deme notifies all its neighbors when it terminates by sending a message with a special tag. When the neighbors receive this message, they update a list of active links, and do not attempt to communicate with a deme that does not exist anymore. This is a simple and effective solution for avoiding deadlock when processes terminate, but there is another complication at the end of a run when there are multiple repetitions of the same experiment.

It is possible that one deme is ready to terminate but its neighbors have already sent migrants to it. The obvious solution would be to ignore the migrants and terminate as usual, but since non-blocking sends are used, the incoming migrants remain in a buffer until the process reads them. These migrants will be read in the next experiment, and they will dominate the population because they come from the end of a previous experiment and likely have a good fitness. A solution to this problem is to read and discard the buffers of the incoming links before terminating the execution of an

experiment.

The last question examined in this section is: How could we make a faster implementation of the multiple-deme GA? Something that can affect the performance greatly is that demes remain idle while they wait to receive individuals from their neighbors. To reduce this idle time we can resort to a common trick in parallel computation: overlap communications and computations. In our case, a simple way to accomplish this would be to evaluate the population after the migrants have been sent and before waiting to receive others. Of course, migrants could only be chosen randomly because the fitness values are unknown. With this scheme the migrants have plenty of time to arrive from other demes while the population is being evaluated. So, when the deme attempts to receive migrants it is more likely that they have arrived.

A less obscure way to improve the performance of multi-population parallel GAs is to combine them with the master-slave model. This is a good choice when there are many processors available, but only a few demes are used. With this "hybrid" approach, the evaluation of the individuals of one deme can be distributed to several processors and the computation power of the parallel machine would be used more efficiently. To implement this optimization we would need to derive the class `Deme` from `GlobalGA`, instead of from the class `GA`.

4 Test Results

The code described in the previous section has been tested extensively on a variety of parallel computers and this section shows the results of some of those tests. Most of the results presented here have been used to validate theoretical results about parallel GAs. They represent good evidence of the accuracy of the models and also of the performance of the code.

We only show results of the code executed on a network of workstations, as this is the most difficult environment in terms of achieving good performance. The same code has been executed on a Connection Machine CM-5, a Silicon Graphics Power Challenge, and an IBM SP without any problems.

In general, a fair comparison of parallel and serial programs involves running the two versions and measuring the execution times. There is an implicit condition that the two versions must give the same results, otherwise the comparison is meaningless. Of course, the same condition has to be enforced in the particular case of GAs, so we compare the execution times of serial and multiple-deme parallel GAs when they find solutions of the

same quality.

Most of the fitness functions used in these tests are artificial problems of known difficulty. We created the fitness functions by concatenating several copies of fully deceptive trap functions like the one shown in figure . Fully deceptive functions are bimodal and the deceptive optimum has a wide attraction basin, while the global optimum is completely isolated. The difficulty of the fitness function depends on three factors: (1) the relative lengths of the deceptive and global basins, (2) the fitness difference between the global and the deceptive optima, and (3) the number of copies of the trap function that are concatenated.

The first tests are of a master-slave parallel GA. In this case the communications are more frequent and the system can only be expected to yield good results if it takes a considerable time to evaluate the fitness function. We added artificial variable delays to the trap functions to simulate the execution time of difficult functions. The results of these tests are in figure 8 along with a theoretical model that predicts the performance of master-slave parallel GAs (Cantú-Paz, 1998a). Note that the efficiency of the code increases as it takes longer to evaluate the fitness, and the speedups can be proportional to the number of slaves for problems that take a very long time to evaluate (see figure 9 for an example).

The next set of tests are of multiple deme GAs. Figure 10 shows the parallel speedups obtained when the demes are executed in complete isolation. There is also a plot of a theoretical prediction of the speedup in the figures (Cantú-Paz & Goldberg, 1997b), and it is evident that for both functions the speedups are very low. The importance of these results is that they do not depend on the particular hardware (as there is no communication and the computation time gets normalized when the speedups are computed) or on the particular test functions. The theory shows that in general there are not significant gains in performance when the demes are isolated.

The next set of tests are also of multiple-population GAs, but in this case each deme is connected to all the others (fully-connected topology) and the migration rate is set to the highest possible value. The migration strategy was to migrate only once after the demes had converged completely. We can see in figure 11 that there is a maximum speedup for each test function and that the theory predicts the actual results very accurately. Notice that in the case of the easier function (composed of 4-bit traps), the speedup is not as significant as for the harder function (composed of 8-bit traps). The reason is that the savings on computation time brought by the use of multiple demes are not enough to overcome the increase in communications.

Recall that these tests are performed on a slow network of workstations, but even under these circumstances the parallel GA performed significantly better when it was used to optimize the harder problem.

The results from the two algorithms seem to point out that it is more efficient to use parallel GAs with more difficult fitness functions. The reason is that in both algorithms the time used in communications offsets the reduction in computation time. As the fitness functions require longer evaluation times, the ratio between computation and communication times is higher and the parallel algorithms become more efficient.

5 Conclusions and Extensions

It is becoming increasingly important to design faster GAs as they are entering industry and are applied to harder problems. Parallel implementations of GAs are a particularly promising method to make GAs faster as parallel computers are becoming more widely spread. GAs are easy to implement on parallel platforms, and they are particularly amenable to parallel computation because the communications to computation ratio can be kept very low.

Users of parallel GAs can benefit from the results that research has produced recently, but the research is by no means complete. The effect of some of the parameters that control parallel GAs is not completely understood, and it is necessary to continue to experiment systematically and develop new models to predict the performance of parallel GAs. A program like the one presented in this chapter is an indispensable tool to continue the investigation on this area.

In this chapter we have discussed the design and the implementation of a parallel GA that is both flexible and simple. The goal was to transmit ideas that should help others to implement their own parallel GAs. We preferred not to implement every possible variation of codings and operators, but instead to produced code that is efficient, easy to understand, and very portable.

The code presented here is a good starting point for new developments, and it can be extended in many directions. In particular, it is not difficult to add other forms of selection or specialized genetic operators that are suitable to a particular application.

Some extensions that can improve performance were also discussed. In the system there are times when processors remain idle as they wait for others to finish their work. For master-slave GAs, we discussed asynchronous

implementations, where the master does not wait for slow slaves. A better implementation of multi-population GAs should overlap communications and computations, and section 3 discussed one strategy to accomplish this.

An interesting extension is to combine coarse-grained and master-slave GAs. This "hybrid" parallel algorithm can be implemented easily by changing the base class for `Deme`. This is an example of the flexibility that results from the object-oriented design.

Of course, there are many other possible extensions to the code, but we have also shown in this contribution that gains in efficiency do not come only from better implementations, but from a more rational utilization of resources. As the theory for parallel GAs advances, better guidelines are emerging to help users exploit the potential of parallel computers.

Acknowledgments

I wish to thank David E. Goldberg for his guidance during my research on parallel GAs. I was supported by a Fulbright-García Robles Fellowship.

This study was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grants number F49620-94-1-0103, F49620-95-1-0338, and F49620-97-1-0050. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

References

Abramson, D., & Abela, J. (1992). A parallel genetic algorithm for solving the school timetabling problem. In *Proceedings of the Fifteenth Australian Computer Science Conference (ACSC-15)*, Volume 14 (pp. 1–11).

Abramson, D., Mills, G., & Perkins, S. (1993). Parallelisation of a genetic algorithm for the computation of efficient train schedules. *Proceedings of the 1993 Parallel Computing and Transputers Conference*, 139–149.

Booch, G. (1994). *Object-oriented analysis and design with applications.* Menlo Park, CA: Benjamin Cummins.

Braun, H. C. (1990). On solving travelling salesman problems by genetic algorithms. In Schwefel, H.-P., & Männer, R. (Eds.), *Parallel Problem Solving from Nature* (pp. 129–133). Berlin: Springer-Verlag.

Cantú-Paz, E. (1998a). Designing efficient master-slave parallel genetic algorithms. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K. Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., & Riolo, R. L. (Eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference* (pp. 455). San Francisco, CA: Morgan Kaufmann Publishers. Also available as IlliGAL Report No. 97004).

Cantú-Paz, E. (1998b). A survey of parallel genetic algorithms. *Calculateurs Parallèles, Reseaux et Systems Repartis*, *10*(2).

Cantú-Paz, E., & Goldberg, D. E. (1997a). Modeling idealized bounding cases of parallel genetic algorithms. In Koza, J., Deb, K., Dorigo, M., Fogel, D., Garzon, M., Iba, H., & Riolo, R. (Eds.), *Genetic Programming 1997: Proceedings of the Second Annual Conference* (pp. 353–361). San Francisco, CA: Morgan Kaufmann Publishers.

Cantú-Paz, E., & Goldberg, D. E. (1997b). Predicting speedups of idealized bounding cases of parallel genetic algorithms. In Bäck, T. (Ed.), *Proceedings of the Seventh International Conference on Genetic Algorithms* (pp. 113–121). San Francisco: Morgan Kaufmann.

Fogarty, T. C., & Huang, R. (1991). Implementing the genetic algorithm on transputer based parallel processing systems. *Parallel Problem Solving from Nature*, 145–149.

Forrest, S. (Ed.) (1993). *Proceedings of the Fifth International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann.

Goldberg, D. E., & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*, *1*, 69–93. (Also TCGA Report 90007).

Goldberg, D. E., Deb, K., & Clark, J. H. (1992). Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, *6*, 333–362.

Grefenstette, J. J. (1981). *Parallel adaptive algorithms for function optimization* (Tech. Rep. No. CS-81-19). Nashville, TN: Vanderbilt University, Computer Science Department.

Grosso, P. B. (1985). *Computer simulations of genetic adaptation: Parallel subcomponent interaction in a multilocus model.* Unpublished doctoral dissertation, The University of Michigan. (University Microfilms No. 8520908).

Harik, G., Cantú-Paz, E., Goldberg, D. E., & Miller, B. L. (1997). The gambler's ruin problem, genetic algorithms, and the sizing of populations. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation* (pp. 7–12). Piscataway, NJ: IEEE.

Hauser, R., & Männer, R. (1994). Implementation of standard genetic algorithm on MIMD machines. In Davidor, Y., Schwefel, H.-P., & Männer, R. (Eds.), *Parallel Problem Solving fron Nature, PPSN III* (pp. 504–513). Berlin: Springer-Verlag.

Munetomo, M., Takai, Y., & Sato, Y. (1993). An efficient migration scheme for subpopulation-based asynchronously parallel genetic algorithms. See Forrest (1993), pp. 649.

Thierens, D., & Goldberg, D. E. (1993). Mixing in genetic algorithms. See Forrest (1993), pp. 38–45.

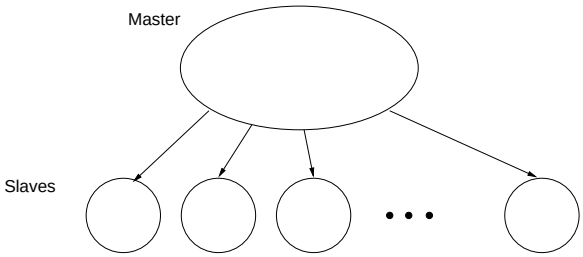


Figure 1: A schematic of a master-slave GA. The master process stores the population and each slave evaluates a fraction of the individuals.

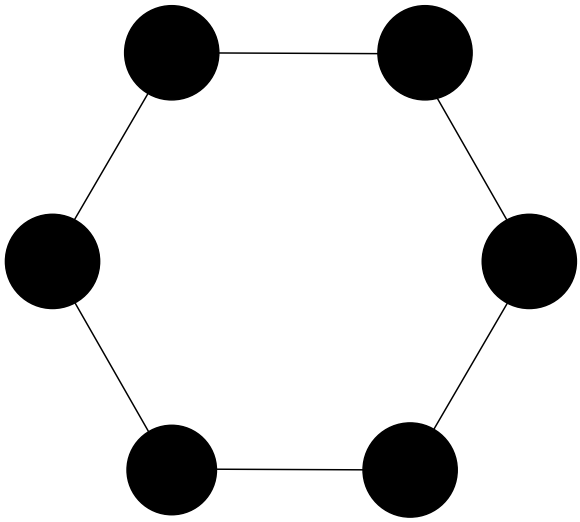


Figure 2: A schematic of a multiple-deme parallel GA. The subpopulations exchange individuals with their logical neighbors on the connectivity graph.

```
// do one generation of the GA
void GA::generate()
{
        evaluate();
        statistics();
        report();

        generation++;

        (*this.*select)();
        Population *temp;
        currentpop;
        currentpop = poptemp;
        poptemp = temp;

        crossover();
        mutate();
}
```

Figure 3: The code for one generation of a simple genetic algorithm. Note that selection is executed using a pointer to a function, this provides flexibility as it is easy to add different selection algorithms to the system and to select between them. For efficiency, the mating pool created by selection (`poptemp`) is not copied directly into the current population, instead just a pointer needs to be swapped.

```
unsigned long GlobalGA::run(char *outfile)
{
        int i;

        for (i=0; i<repetitions-1; i++) {
                GA::run(outfile);
                init_GA();
        }
        GA::run(outfile);

        // tell the slaves to die
        for (i=0; i<slaves; i++){
                pvm_initsend(ENCODING);
                pvm_send(tids[i], END_OF_RUN);
        }
}
```

Figure 4: The main loop of a `GlobalGA`. The only difference with the main `GA` loop is that after repeating an experiment several times, the master instructs the slaves to terminate sending them a specially-tagged message.

```
void GlobalGA::evaluate()
{
        unsigned num_strings;
        double eval_value;
        int i,j;

        assert(size % slaves == 0);

        num_strings = size / slaves;

        // send the strings to the slaves
        for(i=0; i<slaves; i++){
                // send: #strings, length, strings
                pvm_initsend(ENCODING);
                pvm_pkuint(&num_strings, 1, 1);
                pvm_pkuint(&length, 1, 1);
                for(j=0; j<num_strings; j++)
                        pkbyte((*currentpop)[i*num_strings+j].data,length, 1);
                pvm_send(tids[i], STRINGS);
        }

        // get the evaluations back from the slaves
        for (i=0; i<slaves; i++){
                pvm_recv(tids[i], EVALS);

                for(j=0; j<num_strings; j++){
                        pvm_upkdouble(&eval_value,1,1);
                        (*currentpop)[i*num_strings+j].set_fitness(eval_value);
                }
        }
        // update the evaluations counter
        evaluations+=size;
}
```

Figure 5: This is the code for evaluating the population in a master-slave GA. The first loop sends the individuals to the slaves, and each slave evaluates the same number of individuals. The second loop receives the evaluations back from the slaves and sets the fitness values in the population.

```
void DEME::generate()
{
        // do a normal generation
        GA::generate();

        // check to see if it is time to migrate
        if  (generation % migration_interval  == 0)
        {
                SendMigrants();
                ReceiveMigrants();
                IncorporateMigrants();
        }
}
```

Figure 6: The code for one generation in a `Deme`. The difference with a simple GA is that migration occurs every `migration_interval` generations.

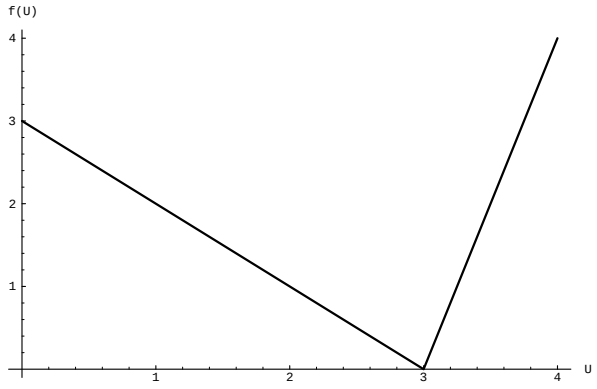
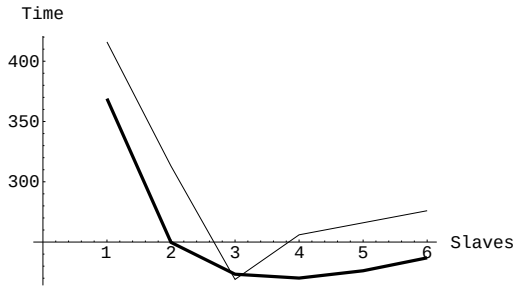
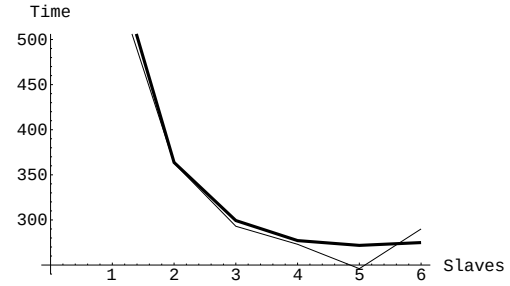


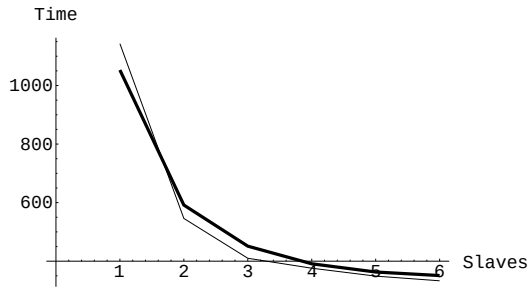
Figure 7: This is a 4-bit deceptive trap function. There is a deceptive optimum at 4 bits with a long attractor, and the global optimum is at the opposite extreme and it has a very small attractor. In some of our experiments we concatenate several trap functions like this to create test problems with known difficulty and building block length.



(a) $T_f = 19ms$



(b) $T_f = 38ms$



(c) $T_f = 76ms$

Figure 8: Execution time for master-slave GA using a dummy fitness function with different evaluation times (T_f). The thin line is the experimental results and the thick line is the theoretical predictions. Note that there is an optimal number of slaves that minimizes the execution time.

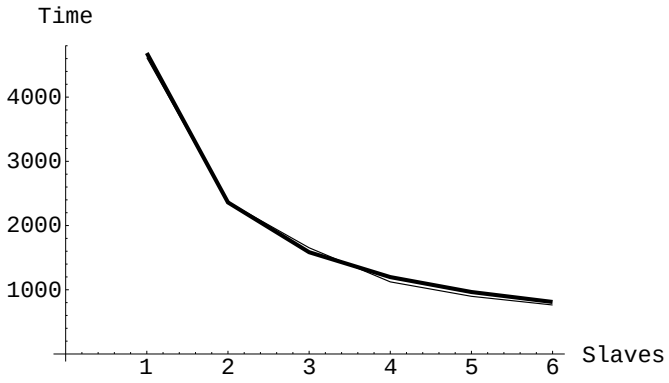
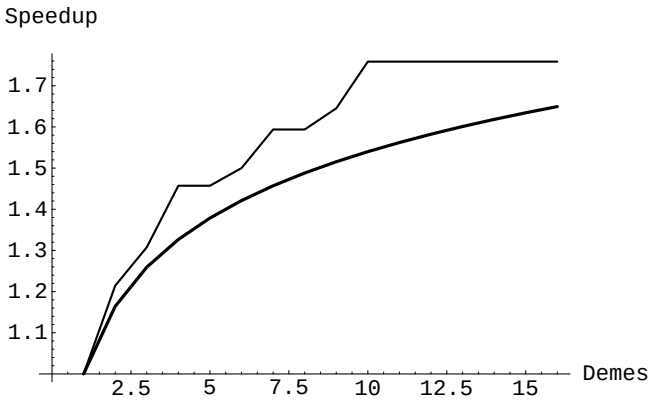
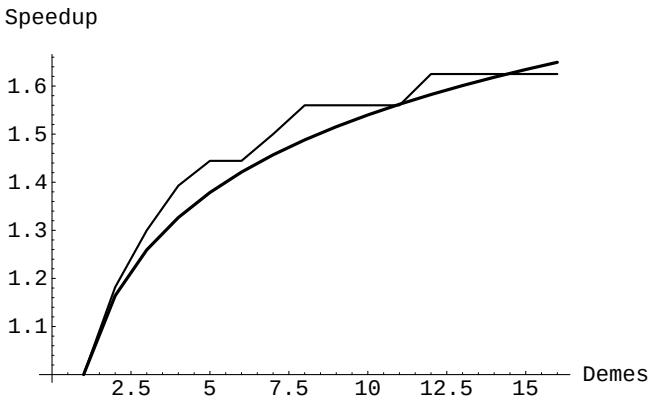


Figure 9: Execution time for a master-slave GA using a NN fitness function. The speedup in this case is almost linear as slaves are added.

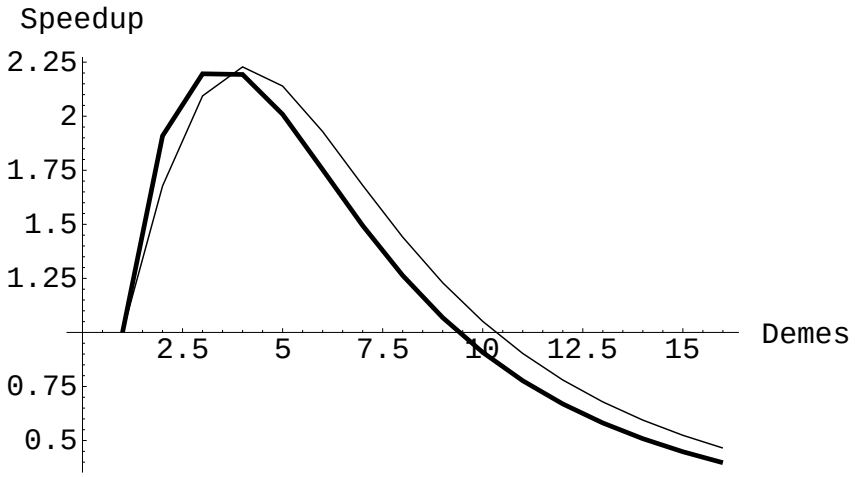


(a) 4-bit trap function

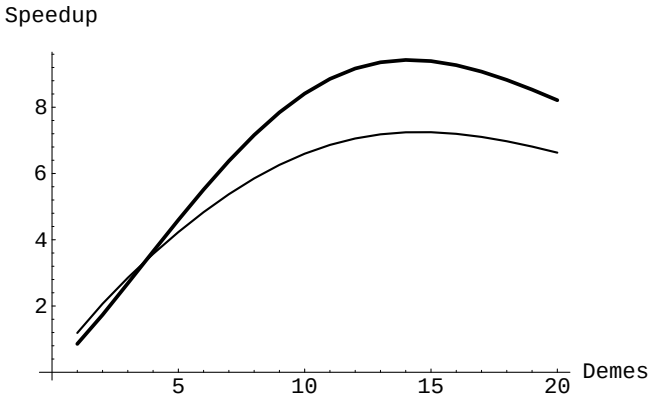


(b) 8-bit trap function

Figure 10: These are the parallel speedups for two test functions of varying difficulty using isolated demes. Both test problems are formed by concatenating 20 copies of a 4-bit fully deceptive trap functions. In both cases the speedup is not significant. The bold lines are the theoretical predictions, and the thin lines are the empirical results.



(a) 4-bit trap function



(b) 8-bit trap function

Figure 11: Parallel speedups for two test functions using fully-connected demes and maximal migration rates. For a 4-bit trap function the performance improvement is not very significant, but the plot shows the existence of an optimal speedup. The plot for the 8-bit trap function shows a much better speedup. The bold lines in both graphs are the theoretical prediction of the speedup, and the thin lines are the experimental results.