This paper appeared in proceedings of the 25th International Conference on Parallel Processing, August 1996.

A STUDY OF A NON-LINEAR OPTIMIZATION PROBLEM USING A DISTRIBUTED GENETIC ALGORITHM

Nuno Neves **Anthony-Trung Nguyen** **Edgar L. Torres**

University of Illinois at Urbana-Champaign

Abstract – *Genetic algorithms have been used successfully as a global optimization method when the search space is very large. To characterize and analyze the performance of genetic algorithms on a cluster of workstations, a parallel version of the GENESIS 5.0 was developed using PVM 3.3. This version, called VMGENESIS, was used to study a nonlinear least-squares problem. Performance results show that linear speedups can be achieved if the basic distributed genetic algorithm is combined with a simple dynamic load-balancing mechanism. Results also show that the quality of search changes significantly with the number of processors involved in the computation and with the frequency of communication.*

1 INTRODUCTION

In the last twenty years a numerical analysis category known as Evolutionary Computation (EC) has emerged as a powerful and broad method for solving optimization problems. The stochastic nature of EC algorithms makes them well suited to solve large and complex problems. However, serial implementations of EC algorithms usually necessitate long execution times to find the solutions. Consequently, many realizations of EC algorithms have been proposed for parallel computers [1, 10, 14, 18, 22]. An alternative approach consists on implementing the EC algorithm on a cluster of workstations. This approach is becoming more appealing to researchers because of the economics of workstations over supercomputers. Most research centers have a large number of interconnected workstations that remain idle during some period(s) of the day. Using this potential computational resource to support the execution of the EC algorithm can save a considerable amount of money, while sacrificing some performance. Depending on how significant this loss of performance is, a cluster of workstations can be a valuable alternative to support optimization problems.

This paper describes the use of a Distributed Genetic Algorithm (DGA) to solve a nonlinear least-squares problem on a cluster of workstations. Measurements show that the execution time of the basic implementation of the DGA depends heavily on load conditions. Speedups are reduced substantially even for small asymmetries in the loads of the nodes. To overcome this problem, a simple dynamic load-balancing scheme was developed. With this new implementation, the DGA exhibited linear speedups in a cluster of workstations with 16 nodes.

This paper describes experiments using two communication patterns among nodes. In the first pattern, nodes are arranged in a logical ring and communication is made only to the nearest neighbor. In the second pattern, information is multicasted from each node to all the other nodes. For each pattern, different ratios of communication were studied by changing the number of computational steps between communications. It was observed that quality of search could vary substantially with the pattern of communication, the communication ratio, and the number of processors involved in the computation. However, the best results on average were obtained with a moderate level of communication, which seems to indicate that independent computation is beneficial.

To make the results as general as possible, a DGA system was developed based on the serial implementation of GENESIS 5.0 [7], a popular genetic algorithm system developed by Grefenstette. The underlying software for the distributed implementation was PVM 3.3 [3], a concurrent computing framework developed at the University of Tennessee.

2 BACKGROUND

Within EC there are two basic methods specific to numerical optimization: simulated annealing and genetic algorithms. Simulated annealing [13] has been studied for its ability to find global optima in large search spaces. This method was inspired by the cooling characteristics of a crystal. Simulated annealing starts in one of the states of the search space and computes the corresponding function value. Then, in each step, it randomly selects one neighbor state and calculates its function value. If the neighbor's function value is better than the current state value (in a minimization/maximization problem a value is better if it is smaller/larger), the neighbor state becomes the new current state. Otherwise, it becomes the new state with a given probability that depends on the current temperature. Although this method has been used successfully in many applications, recent papers have questioned its global convergence rate and accuracy [2]. Another drawback of this method is that several parameters of an algorithm are application dependent. This can often result in a significant amount of time spent on configuring the algorithm.

Genetic algorithms [4, 8, 9, 20] were inspired by the natural

Nuno Neves was supported in part by the Grant BD-3314-94, sponsored by the program PRAXIS XXI, Portugal. This work was supported in part by the Office of Naval Research under contract N00014-91-J-1283, and by the National Aeronautics and Space Administration under grant NAG-1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS). The findings, opinions, and recommendations expressed herein are those of the authors and do not necessarily reflect the position or policy of the United States Government and no official endorsement should be inferred.

evolution of species. They have been studied in the last twenty years as an EC method that can overcome some of the simulated annealing drawbacks [11, 21]. Genetic algorithms are executed as a series of steps, called generations. Intuitively, they start with a population with a certain number of individuals (states in the search space). In each generation, the individuals are evaluated and the fittest reproduce and continue in the next generations. The reproduction phase also introduces new individuals with different characteristics. However, these new individuals preserve some of the characteristics of their parents. As this process continues, the population converges to better individuals, which correspond with high probability to the global optimum.

3 DESIGN OF THE DGA

The DGA associates a sub-population with each processing node (in general, the initial population is divided equally among the nodes). In each node, the sub-population goes through the same generational steps that were described in the previous section. However, to simulate true genetic evolution, some individuals must be exchanged among sub-populations. These exchanges are called *migrations*. Migrations are inadequate when the DGA favors some local optimum due to inappropriate perturbation from one sub-population to another. This can happen, for instance, if too many individuals are migrated, or if migrations occur too frequently. Therefore, searching for a global optimum with a DGA requires finding a certain number of parameters.

There are three factors that describe how a given migration works. The first factor specifies the selection criteria for choosing which individuals should be transferred. In our implementation, the most favorable individuals in a sub-population are selected. The second factor is the size and frequency of the migration. This study uses a fixed percentage of the sub-population as size. In each migration the 10% most fittest individuals are exchanged. The frequencies used in the experiments were 10, 100, and 1000 generations. The third factor, the migration pattern, is specific to distributed implementations of genetic algorithms. In parallel implementations, the architecture and interconnection network usually determine the best transmission scheme for migrating individuals. In distributed environments, since machines are connected by a broadcast medium (e.g., bus), the migration pattern is usually guided by other principles. This paper describes the use of a multicast scheme and a sequential ring.

As part of the study, a DGA system was developed based on the serial implementation of GENESIS 5.0 [7] and on PVM 3.3 [3]. The DGA system is called Virtual Machine GENEtic Search Implementation System (VMGENESIS). A description of the algorithm implemented by VMGENESIS is presented in Figure 1. The algorithm consists of three phases: the initialization, the generation and the ending. Participating processes have different attributes in the distinct phases. In the initialization and ending phases, the algorithm uses a master-slave scheme. During the generation phase, all processes perform the same task.

In the initialization phase, processes obtain all configuration information that will be needed during the generation phase. The first process that is created becomes the master. This process reads a set of configuration parameters from a file (e.g., migration interval) and creates a specified number of slave processes using the PVM virtual machine. Then, the master process creates the initial population. The initial population can contain individuals predefined by the user, individuals created randomly, or both. To end this phase, the master divides and distributes the initial population through all the processes. Concurrently with the master, each slave reads the same configuration file. Sub-populations are not created in parallel, because that would complicate the distribution of the user-defined individuals. To reduce the size of the messages and packing overheads, individuals are sent using the packed format of GENESIS.

```
procedure VMGA
   g = 0;   // Generation 0
   id = unique processing node number;
   if id = 0   // Master
      Initialize P(0);     // Create initial population
      Distribute P(0);    // Send sub-populations; including itself
   else        // Slave
      Await to receive P(id, 0);

   forall nodes do     // Concurrently
      Evaluate individuals in P(id, 0);
      while termination condition not satisfied do
         g = g + 1;
         Select P(id, g) from P(id, g-1);       // Create new population
         Recombine individuals in P(id, g);  // Crossover and Mutate
         Evaluate individuals in P(id, g);     // Assign weights
         if g is multiple of migration interval
            Migrate individuals;                  // Send individuals

   Collect results;
```

Figure 1: Pseudocode for VMGENESIS.

During the generation phase, processes execute the basic algorithm described in the previous section. Processes start by evaluating the individuals belonging to the initial sub-populations. Then, they select the best fitted individuals to create new sub-populations. Next, the processes use the crossover and mutation operators to generate a few individuals with different characteristics. Next, they evaluate the new sub-populations. The extra step that is executed is the migration. Migrations are carried out immediately after the evaluation, because at that moment it is possible to determine without extra work which are the most favorable and the worst individuals. Migrations are performed at a fixed number of generations, whenever the generation counter reaches a multiple of the migration interval. The following steps are executed in each migration. First, the process determines the N fittest individuals in its sub-population. The value of N is a percentage of the size of the sub-population, currently 10%. These individuals are then packed in a message and sent. Packing is done field by field to allow the XDR conversion performed by PVM. More efficient implementations could have been done, but support for heterogeneity requires this procedure. Next, the M worst individuals are found. The value of M can be different from N, and it depends on the migration pattern that is being used. Finally, the process waits for the individuals sent by the other processes, to replace the M worst individuals. All communication is done using direct TCP channels, without going through the PVM daemons.

In the last phase, the master collects the results from the

slaves and performs statistical analyses. The collection of the results is executed in two steps. In the first step, each slave process finds its fittest individuals and sends them to the master. This operation is processed in the same manner that was used when individuals were sent in the migrations. In the second step, the master informs the slaves that they can terminate. This step is only carried out after the master has received all results. This two-phase ending scheme was required to avoid early termination of the faster slaves.

4 METHODOLOGY

One of the most common unconstrained optimization problems is the nonlinear least-squares problem. This problem appears frequently during the analysis of experimental data. The nonlinear least-squares problem tries to find a set of parameters X that gives the "best fit" of the model function $f(t, X)$ to the experimental data. The experimental data is a set of data points $y(t_i)$ obtained during a certain period, at discrete points of time t_i. The data-fitting problem is nonlinear if $f(t, X)$ depends on a nonlinear way with any one of its parameters $x_i \in X$. The optimization problem is to find an X that minimizes the following function:

$$\min_X \sum_{i=1}^{m} (y(t_i) - f(t_i, X))^2$$

The model function used in the experiments consisted of a sum of polynomials and exponentials. Models of this sort are common and have been used to model the size of a population, the luminosity of stars and the reliability of computer systems [12, 19]. The particular function used in the experiment has four polynomials and two exponentials, which correspond to eight unknowns, and consequently to an eight-dimension search space. The model function had the form:

$$f(t, X) = x_1 + x_2 t + x_3 t^2 + x_4 t^3 + x_5 e^{x_6 t} + x_7 e^{x_8 t}$$

The experimental data was generated using the model function plus a stochastic factor. All coefficients of the function were floating-point numbers, with values belonging to the range [-10.0, 10.0]. Coefficients were generated using a random number routine. The stochastic factor tries to simulate experimental errors. It was generated with the same random number routine, with values belonging to the interval [0.0, 1.0]. The independent variable, time, moved from 0 to 10 in increments of 0.1; this provided a total of 100 points for the experiment.

The first step in the study was to find a good set of configuration values for the genetic algorithm. They include different crossover and mutation rates, population sizes, and number of generations. All of these experiments were done using the GENESIS 5.0 serial code. The initial population was created randomly with values for each x_i belonging to the interval [-20.0, 20.0]. However, the same seed was used to generate the same initial population for all experiments. The fitness function that was minimized is described by the following equation:

$$fitnessFunct(X) = \sum_{i=1}^{100} (y(t_i) - f(t_i, X))^2$$

The best configuration values that were found are the following: 0.6 for the crossover rate, 0.001 for the mutation rate, 500 individuals for the size of the population, and 4000 for the number of generations (around 2 million function evaluations). This corresponds to 27.7 minutes of execution time for the serial code on a SUN SPARCStation 10/30. The best value found for the fitness function was 14.7. This corresponds roughly to an average error in each data point of 0.38. No attempt was made to get the best parameters for the distributed implementation. Instead, the configuration values of the serial algorithm were applied to the distributed execution. The decision to do this was made in order to avoid introducing extra free variables, and to make the results more comparable.

The nonlinear least-squares problem has been studied extensively by numerical analysts. A number of methods have been developed for this problem, and they usually converge to a solution rapidly. However, unless the initial starting point is close to the solution, they often converge to a local optimum. Two techniques, a minimization function based on a quasi-Newton method with a line search (function UNCMIN from [12]) and an implementation of the Gauss-Newton method, were used to solve the above optimization problem for the purpose of comparison with our work in distributed genetic algorithms. The results obtained with the minimization function were not satisfactory. Even when it was started with the parameter X that was used to generate the experimental data, the function found a worse result (around 35.4) than those obtained with a single-process genetic algorithm. The Gauss-Newton implementation found a better minimum (around 10.2) for an initial X equal to the one used to generate the experimental data. However, when some coefficients of the starting point were changed, it started to converge to local minima (from 100 to 2000). Both of the methods showed poor results when the initial point was far from the solution.

5 EXPERIMENTAL RESULTS

This paper reports experiments with two different migration patterns. Each migration pattern is compared against the best execution of the GENESIS 5.0 serial code running on a single processing node. The first pattern, a sequential migration ring, requires a small amount of communication. The second pattern uses a multicast migration scheme, which involves more communication.

Two performance metrics were used in the experiments: speedups and quality of search. Speedups are used to determine the execution time improvements that are achieved with the distributed implementation. Consequently, speedups also give an estimate of the importance of the overheads introduced by running the genetic algorithm in a cluster of workstations. Quality of search is used to determine if the distributed implementation is able to find an solution "as good" as the serial implementation. This measure is used, for instance, to study whether the optimization quality of the genetic algorithm is affected by using small sub-populations instead of a large population.

The distributed environment consisted of sixteen SUN SPARCstation 10/30 workstations, with 64 Mbytes of RAM and

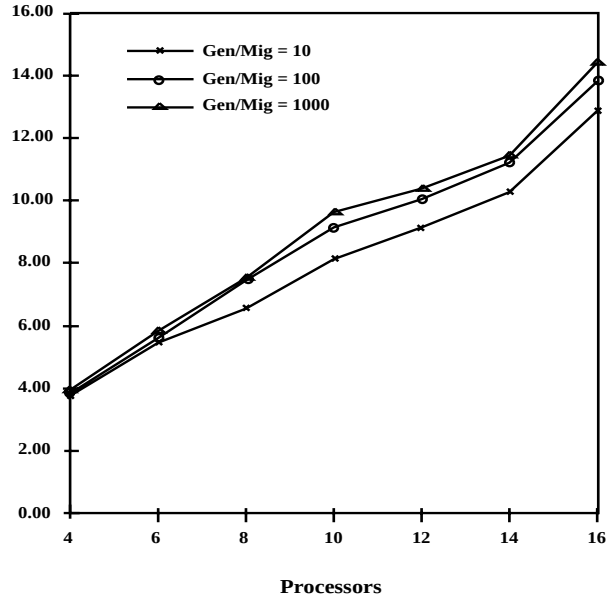

Figure 2: Speedups on a ring configuration.

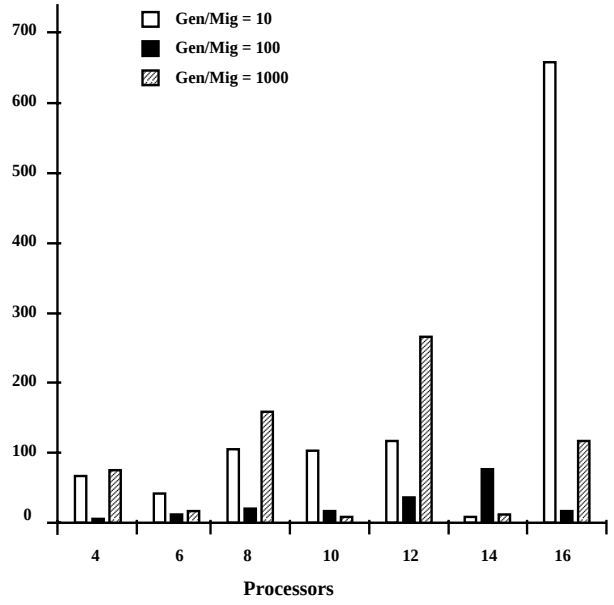

Figure 3: Quality of search on a ring configuration.

running the Solaris 4.1.3. The interconnection network was a 10 Mbits/sec Ethernet. Both GENESIS 5.0 and VMGENESIS were compiled using the gcc 2.6.0 compiler with the optimization level O4. All experiments were run on machines with light load. This environment is a homogeneous network in the sense that all the machines composing the virtual machine are of the same brand and model. This choice was based on the need to isolate performance factors independent of computer specifics. Although it is likely that execution times will vary with different computers, it is assumed that the proportion among different performance factors will remain similar.

5.1 RING PATTERN

In the ring migration scheme, nodes were organized in a logical ring. This scheme migrates individuals from one processing node to the next based on a unique identifier assigned to each node. In each migration stage, a node sends its best individuals to the next node in the ring and receives individuals from the previous node. This communication pattern is implemented at a logical level, since the underlying interconnection network is a bus. As a consequence, our results do not take any advantage of the nearest-neighbor communication pattern in terms of reducing the contention for the interconnection network.

Figure 2 shows the observed speedups for three migration frequencies. The figure plots three curves, each corresponding to a different level of migration. Every migration is initiated whenever the generation counter reaches a multiple of the migration interval. The curve "Gen/Mig = 10" corresponds to executions in which migrations were performed every 10 generations, giving a total of 400 migrations. The migration interval for the other two curves was 100 and 1000, giving a total number of migrations of 40 and 4 respectively. As expected, the executions corresponding to the migration interval of 1000 gave the best speedups. The speedup for the execution with sixteen machines was 14.4. The speedups for the other two migration intervals were somewhat smaller, with values of 13.9 and 12.8 for sixteen machines. However, it is worth noticing that the differences are small for migration intervals that are 10 times and 100 times apart.

Figure 3 shows the minimums that were obtained with various executions. Each minimum represents the best optimum that was found in all sub-populations. It can be observed that values change both for different numbers of processors and for different migration rates. The best average results were obtained with the migration interval of 100. Most of its optima were close to the value obtained with one processor. The best overall quality of search was 7.9 with two processors, followed by 10.6 with sixteen processors. The results obtained with a migration interval of 10 were reasonable (except for sixteen processors). The less desirable results are due to the high frequency of migrations. In each migration, the 10% worst individuals in one sub-population are replaced by the best individuals of another sub-population. If the migration frequency is too high, the diversity present in each sub-population tends to disappear, and all sub-populations start to converge to a small number of well-fitted individuals. However, there is no guarantee that these individuals are converging to the global optimum; they might be converging to a local optimum. Two extreme cases of this behavior can be observed with fourteen and sixteen processors. With fourteen processors, the population converged to the small minimum of 8.7, and with sixteen it converged to 656.4. With the migration interval of 1000, only four migrations were performed. On average, the quality of search was not acceptable. This phenomenon occurs because each sub-population is converging almost independently to its optimum. This often results in poor minimums, because each processor is working with a smaller number of individuals than

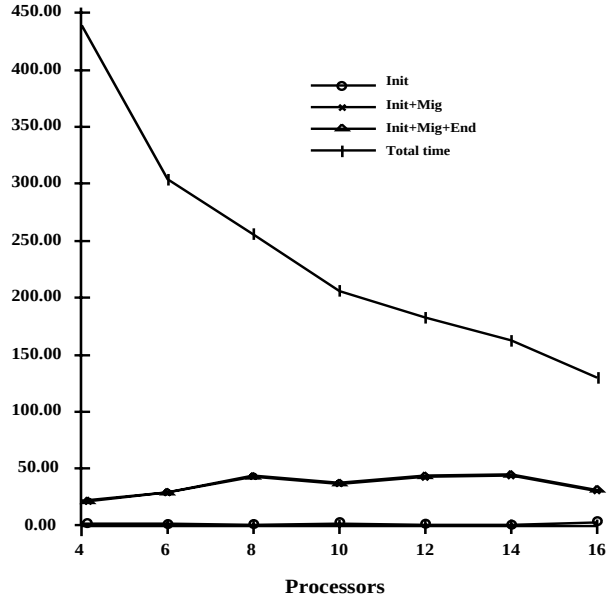

Figure 4: Execution time divided by the different computation phases (100 Gen/Mig).

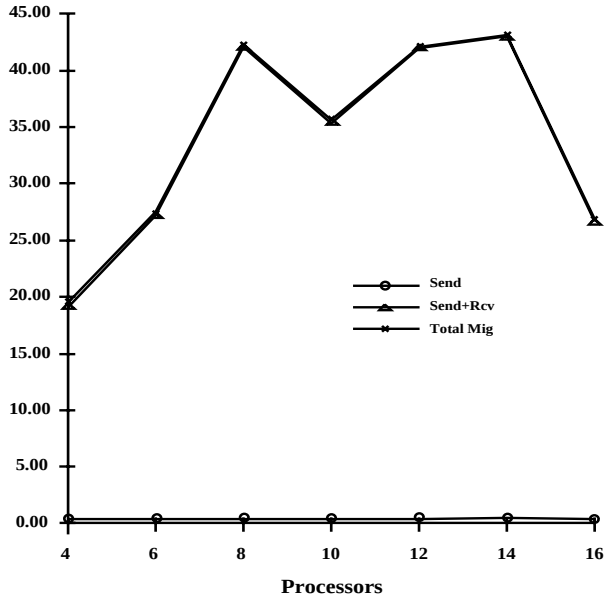

Figure 5: Migration time (100 Gen/Mig).

the optimal size of population (which was the one that was used with one processor).

Figure 4 illustrates the elapsed times of each phase in the execution, as seen by the master. The values shown correspond to the migration interval of 100. As expected, most of the time was spent on computation (the area between the curves "Total time" and "Init+Mig+End"). The time spent on both the initialization and ending phases was negligible when compared to the migration phase. Even with sixteen machines, the cost of initialization and ending phases does not seem to compromise the speedups. The migration overhead starts to grow, and then it stabilizes and decreases a little, as the number of nodes grows. One explanation for this behavior could be the size of the messages. As the number of nodes increases, each sub-population gets smaller, and fewer individuals are migrated (10% of the subpopulation is migrated). However, according to Figure 5, this is not an important factor. The figure represents the migration time in three categories: the time spent calling the pvm_send routine ("Send"), the time waiting in the pvm_rcv routine (area between "Send" and "Send+Rcv") and the time used to determine the migrating individuals and to pack and unpack messages (area between "Total time" and "Send+Rcv"). It can be easily observed that most of the time is spent waiting for the messages. This behavior is caused by small load imbalances in the computing nodes. The irregular pattern of the waiting time also points in this direction. To solve this problem, a dynamic load-balancing mechanism was developed. Its description is given later in the paper.

5.2 MULTICAST PATTERN

In the multicast migration scheme, individuals are migrated from one sub-population to all others. As discussed in the previous section, a migration is initiated whenever the generation counter reaches a multiple of the migration interval. Then, the best 10% individuals are selected and sent to the other nodes. This type of migration scheme is well adapted to the interconnection network being used. Ideally, one message would be sufficient to distribute the migrating individuals. However, PVM does not take advantage of this. With the current technology most of the software that is available in the commodity workstations does not support communication by multicast. As a result, PVM has to send one message to each node that needs to receive the migrating individuals.

Figure 6 gives the speedups for this migration pattern. The curves correspond to the same migration intervals that were used for the ring migration scheme. As before, the least frequent migration rates give better speedups, since most of the overhead is due to communication. However, in this case the speedups vary widely from one migration interval to another. The speedups for sixteen processors, from the least to the most frequent migrations, are 14.1, 11.5 and 6.7 respectively. For the migration interval of 10, the speedups start to decrease when the number of nodes increases from fourteen to sixteen. One reason for the performance decrease with the multicast scheme is that more messages are being sent and many of them are sent at the same time. However, the most important factor is the load imbalances. Curves similar to those in Figure 5 show that the time spent in the pvm_rcv routine is much higher than the other overheads. Instead of having to wait for a single process, with this migration scheme, a node has to wait for all processes before it can continue with the computation. This means that at each migration interval, all nodes perform something similar to a global syn-

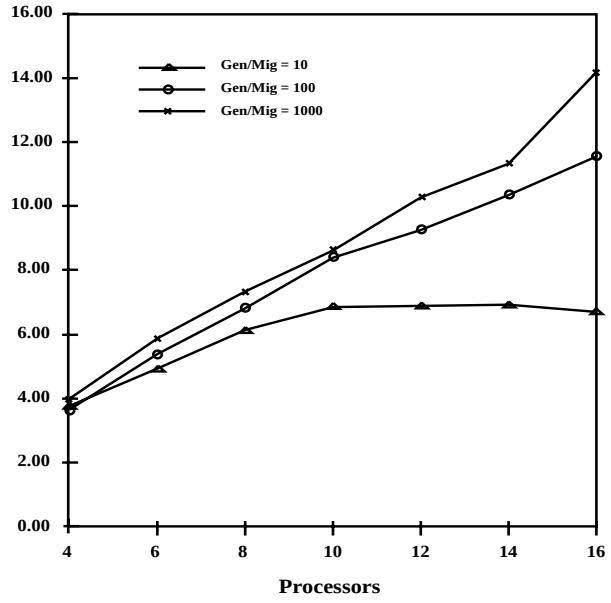

Figure 6: Speedups on a multicast configuration.

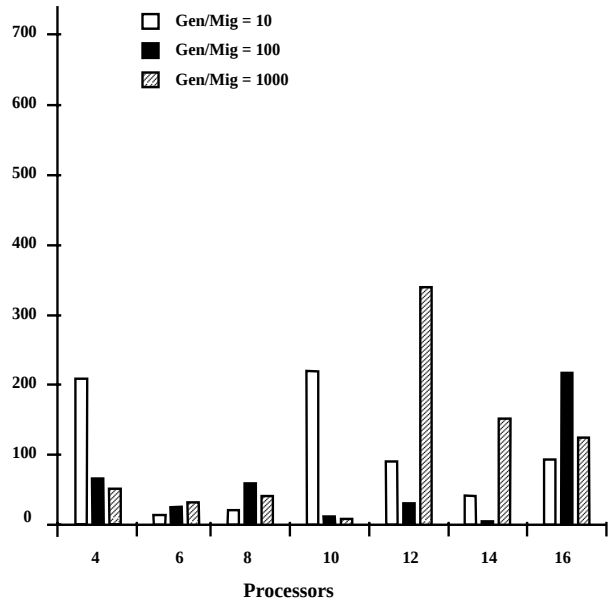

Figure 7: Quality of search on a multicast configuration.

chronization before they continue. As a result there is a much higher dependence on load imbalances.

Measurements of the minimums in the multicast scheme are shown in Figure 7. Intuitively, it should seem that the multicast migration scheme should behave in a manner relatively similar to the single population configuration (or at least more so than the ring configuration would), since there is a fair amount of communication. This can be observed for up to ten processors. However, for a higher number of nodes, this configuration suffers from the problem that too many individuals are exchanged in each migration. A node replaces all its individuals in each migration, since it receives individuals from all the other nodes. This problem is over and beyond the problem of too much communication mentioned in the discussion of the ring configuration. The consequence of these two problems is that nodes might start to converge to a local optimum, as explained previously. To solve the problem of too many exchanges of individuals, one can decrease the percentage of migrants or increase the population size.

5.3 LOAD BALANCING

A cluster of workstations shared by many users creates several load balancing problems for any distributed application that tries to explore the parallel nature of these systems. In fact, most of the previous overheads in the genetic algorithm implementation were due to load imbalances. To solve this problem, a load-balancing mechanism with certain properties has to be used. First, the mechanism should be able to adapt dynamically to different load weights. It also should perform its function without interfering with the basic genetic algorithm. Second, the load-balancing mechanism should avoid migration-based schemes. In these schemes part or all the work being executed in one node is moved to another node. However, it is usually difficult to determine whether it is worth doing the migration, since the new node might become overloaded immediately after the migration, or the previous node might become free. Last, the load-balancing mechanism should be very simple and efficient.

The load-balancing mechanism that was implemented worked in three phases. Each phase was designed to deal with a distinct kind of load. The purpose of the first phase is to tolerate small load differences. In the migration step, a process does not block while waiting for a message to arrive. It simply checks whether there is any message available. If a message has arrived, the process reads the data and continues as before. Otherwise, until it receives a message, the process continues with the computation and tries to determine at the end of each generation whether a message has arrived.

The second phase is used to solve moderate load problems. When the first phase starts, a waiting counter is set to a value N (50 in the current implementation). This counter is decremented whenever the process is unable to read a message because it has not arrived. The second phase of the load-balancing mechanism starts when the counter reaches zero. Therefore, this phase is initiated only when the sender is delayed by N generations. In this phase, the only thing that the process does is send a message to the sender process telling it to skip N generations. The key idea behind the second phase is that the sender is already suffering from a large unrecoverable delay. Therefore, the best solution is to skip some generations and hope that the load will decrease in the future.

For the normal cases of load imbalances, phases one and two should be sufficient. The third phase is used as last resort to solve the remaining cases of overloading. This phase is started whenever the second phase has been executed M times (5 in the current implementation). In this phase, the sender process is removed from the computation. The removing procedure is exe-

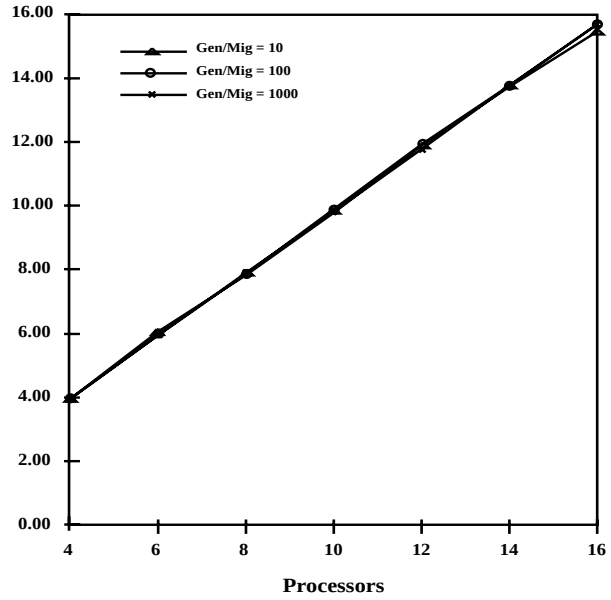

Figure 8: Speedups on a ring configuration with load balancing.

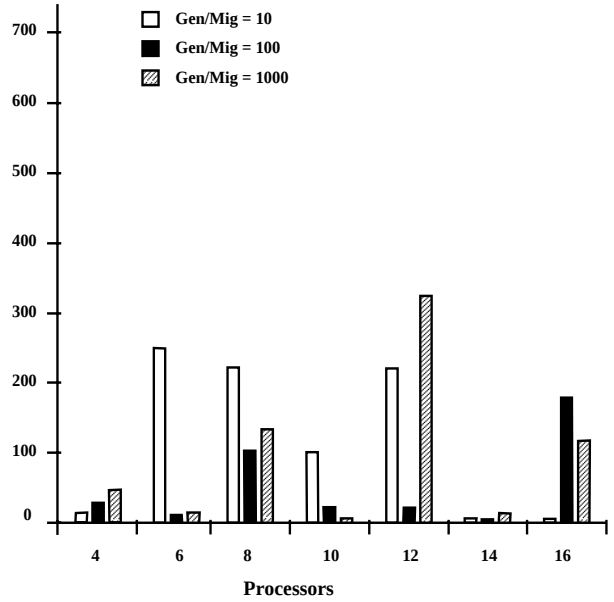

Figure 9: Quality of search on a ring configuration with load balancing.

cuted in two steps. In the first step, the receiving process sends a message to the delayed process telling it to exit. In the second step, the delayed process informs all the other processes about its exit, distributes its individuals through them, and then leaves the computation. The third phase should be applied with a certain care because it ends up migrating work, which is something that this mechanism tried to avoid.

Figures 8 and 9 show the measurements of a ring migration pattern with the load-balancing mechanism. It is possible to observe that most of the overheads are removed, and that speedups scale almost linearly. The cost of the load-balancing mechanism is that migrations start to happen at irregular generational times. However, on average, this did not seem to affect the quality of search. In certain cases, even better minimums were obtained.

5.4 RELATED WORK

Genetic algorithms, proposed by Holland [8], are inherently parallel algorithms for searching and optimization problems. In the 1980s, Grefenstette [5] developed four different design versions of parallel genetic algorithms (PGA). One of the versions is the network model, also known as coarse-grained or island model. This model subdivides the population among participating processes, each of which applies the entire genetic algorithm to its sub-population. In each iteration, processes broadcast their fittest individuals to the other processes. Our experiments use the island model by distributing sub-populations to different workstations connected by a local area network. Migration factors such as interval, connectivity, and topology can be specified by input parameters. In particular, we use the synchronous island model by migrating individuals at the end of the evaluation phase. Since sub-populations are processed on different workstations with different speeds and loads, the workload can be uneven, forcing some processes to wait for others to migrate individuals. To address the load imbalance issue, we introduce optimizations that allow a sub-population to continue to the next phase even when individuals from other sub-populations have not arrived.

The topology of the connections of nodes and the degree of connectivity influence the performance of island-model PGA. Many implementations of island-model PGA are tightly coupled with the underlying network topology and system architecture. For instance, the network topology can be a 2-D mesh [16] or a hypercube [22], and the communication paradigm can be a shared-memory multiprocessor or a message-based multicomputer. Such implementations are not portable across different architectures and topologies. Our implementation uses PVM for portability and creates a logical communication topology based on an input description file.

One major problem with genetic algorithms is that the population can prematurely converge to a suboptimal solution. To solve this problem, global parameters such as population size, crossover rate, and mutation rate must be carefully chosen [6]. While tuning the parameters has been the recommended technique to control premature convergence, there is no general method for choosing ideal parameters, which are usually application-dependent. Hybrid algorithms have been proposed to address this problem by combining the genetic algorithm with another method, to incorporate strengths and overcome shortcomings of both methods. Muhlenbein et al. [17] introduced hillclimbing to the PGA. The PGA attempts to obtain a local minimum in each sub-population. If a sub-population fails to progress after a number of generations, hillclimbing is applied. Mahfoud and Goldberg [15] incorporated a genetic algorithm

with simulated annealing to take advantage of the large population of possible solutions of the genetic algorithms while retaining the convergence properties of simulated annealing. Since hybrid algorithms can add another level of complexity to the implementation, this study did not consider this approach. Instead, our effort was focused on building a DGA capable of achieving near-linear speedups on a cluster of workstations.

6 CONCLUSIONS

This paper describes an implementation of a distributed genetic algorithm for a cluster of workstations. Two migration patterns were proposed: a logical ring and a multicast scheme. Three migration frequencies were applied to each migration pattern. The best results, in terms both of speedup and quality of search, were obtained with the ring configuration. In the ring configuration, speedups were close to 14 for the medium frequency of migration in a cluster with sixteen machines. The majority of the overheads came from the load imbalances during the computation. To remedy this problem, a dynamic load-balancing mechanism was developed. Results with a ring configuration with the load-balancing mechanism show that linear speedups can be achieved for clusters with 16 workstations.

The various experiments showed that the quality of search depends on the pattern of migration, frequency of migration, and number of processors. The experiments show that the best results were obtained for the medium frequency of migration. Higher frequencies of migration result in the replacement of too many individuals, reducing the diversity of the population. As a consequence, the genetic algorithm starts to converge to a local optimum. Smaller frequencies of migration lead to almost local convergence, which can also cause the algorithm to converge to a local optimum.

ACKNOWLEDGMENTS

We would like to thank Professor Daniel A. Reed for his comments and guidance on the initial draft of this paper. We also wish to thank the referees for their suggestions. We thank Professor W. Kent Fuchs for making available to us the machines that were used to test VMGENESIS, and Pedro Trancoso for his comments.

REFERENCES

[1] E. Felten, S. Karlin, and S. W. Otto. The traveling salesman problem on a hypercube, MIMD computer. In *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

[2] A. G. Ferreira and J. Zverovnik. Bounding the probability of success of stochastic methods for global optimization. *Computers Mathematical Applications*, 25(10/11), 1993.

[3] A. Geist et al. PVM 3 user's guide and reference manual. Technical Report 12187, Oak Ridge National Laboratory, 1994.

[4] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, 1989.

[5] J. J. Grefenstette. Parallel adaptive algorithms for function optimization. Technical Report CS-81-19, Vanderbilt University, 1981.

[6] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1), 1986.

[7] J. J. Grefenstette. A user's guide to GENESIS version 5.0. Technical report, Naval Research Laboratory, 1990.

[8] J. H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[9] C. Janikow and D. St. Clair. Genetic algorithms: Simulating nature's methods of evolving the best design solution. *IEEE Potentials*, 1995.

[10] P. Jog and D. Van Gucht. Parallelisation of probabilistic sequential search algorithms. In *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987.

[11] V. S. Johnston and M. Franklin. Is beauty in the eye of the beholder? *Ethnology and Sociobiology*, 14, 1993.

[12] D. Kahaner, C. Moler, and S. Nash. *Numerical methods and software*. Prentice Hall Series in Computational Mathematics, 1989.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598), 1983.

[14] S.-C. Lin et al. Coarse grain parallel genetic algorithms: Categorization and new approach. *Parallel & Distributed Processing*, 1994.

[15] S. W. Mahfoud and D. E. Goldberg. Parallel recombinative simulated annealing: a genetic algorithm. *Parallel Computing*, 21, 1995.

[16] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[17] H. Muhlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17, 1991.

[18] C. B. Pettey, M. R. Leuze, and J. J. Grefenstette. A parallel genetic algorithm. In *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987.

[19] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems: design and evaluation*. Digital Press, 1992.

[20] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 1994.

[21] B. Stuckman, G. Evans, and M. Mollaghasemi. Comparison of global search methods for design optimization using simulation. In *Proceedings of the 1991 Winter Simulation Conference*, 1991.

[22] R. Tanese. Parallel genetic algorithm for a hypercube. In *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987.