Designing Efficient Master-Slave Parallel Genetic Algorithms

Erick Cantú-Paz

IlliGAL Report No. 97004
May 1997

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

Designing Efficient Master-Slave 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

A simple technique to reduce the execution time of genetic algorithms (GAs) is to divide the task of evaluating the population among several processors. This class of algorithms is called "global" parallel GAs because selection and mating consider the entire population. Global parallel GAs are usually implemented as master-slave programs and require constant interprocessor communication. This will affect their performance, but most investigations of these algorithms ignore the penalty caused by communications. This paper presents an analysis of the execution time of global parallel GAs that includes a simple model of the time used in communications and shows that there is an optimal number of processors that minimizes the execution time. To further reduce the execution time we recommend the use of hybrids that combine global and coarse-grained parallel GAs.

1 Introduction

A simple technique to parallelize genetic algorithms is to divide the task of evaluating the population among several processors. In this technique selection and mating are global (i.e., they consider all the individuals in the population) and the resulting algorithm is called a "global" parallel GA. Global parallel GAs explore the search space in exactly the same manner as a serial GA and are very easy to implement. However, there has been little theoretical analysis to study the advantages that this method offers. The objective of this paper is to quantify the reduction in the execution time and to identify the necessary conditions for an improvement in performance.

The execution time of global parallel GAs has two major components: the time used in computations and the time used to communicate information among processors. In turn, the computation time is largely determined by the size of the population (Goldberg & Deb, 1991) so at first it seems that to improve performance we should reduce the population. However, the population size is also a major factor in the effectiveness of GAs and if the population is reduced then the probability that the GA will find good solutions would decrease (Goldberg, Deb, & Clark, 1992; Harik, Cantú-Paz, Goldberg, & Miller, 1997).

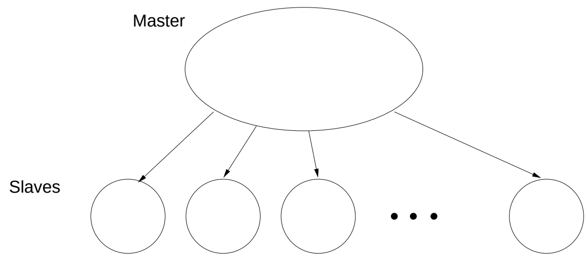


Figure 1: A schematic of a global parallel GA.

Global parallel GAs are usually implemented as master-slave programs. The time used in communications depends directly on the number of slave processors and on the particular hardware used to execute the algorithm. The analysis in this paper assumes a simple but realistic model for the communications time and it shows that there is an optimal number of processors that minimizes the execution time of the global parallel GA.

The next section of the paper presents background material on global parallel GAs. Section 3 gives a theoretical analysis of their performance that results in guidelines for the design of efficient master-slave GAs. Section 4 shows the results from computational experiments that validate the theory. In section 5 the discussion turns to a method that combines global and coarse-grained parallel GAs that promises a greater improvement in performance than any of the two basic parallel methods alone. Finally, we present a summary with the conclusions of this study.

2 Global parallel genetic algorithms

Many parallel programs are based on a divide-and-conquer principle that principle can be applied to GAs in several ways. Indeed, the literature contains numerous examples of different methods to parallelize GAs. This paper concentrates on global parallel genetic algorithms that use one population and divide the task of evaluating the population among several processors. Another popular method of parallelization is coarse-grained parallel GAs which divide the population and execute a conventional GA on each of the subpopulations. Another method that also divides the population is fine-grained parallel GAs, but in this case the subpopulations are much smaller than in the coarse-grained algorithm. It is also possible to combine these methods to produce hybrid parallel GAs and, as we shall see in a later section of this paper, they promise greater improvements in performance. A complete review of the different methods of parallelization can be found elsewhere (Cantú-Paz, 1997).

It is important to emphasize that while the global parallelization method does not affect the behavior of the algorithm, the other methods introduce fundamental changes in the way the GA works. For example, in the global method the selection mechanism takes into account the entire population, but in the other methods selection is performed over a subset of individuals (the deme). Also, it is possible to mate with any individual in the global GA, but in the methods that divide the population mating is restricted to the deme.

In global parallel GAs the most common operation that is parallelized is the evaluation of the individuals, because the fitness of an individual is independent from the rest of the population and there is no need to communicate during this phase. The evaluation of individuals is parallelized by assigning a fraction of the population to each of the processors used. Communication occurs only as each processor receives a subset of individuals to evaluate and when the processors return the fitness values.

If the algorithm stops and waits to receive the fitness values for all the population before proceeding into the next generation, then the global parallel GA is called **synchronous** and it has exactly the same properties as a simple GA, with a possibility of better performance being the only difference. However, it is also possible to implement an **asynchronous** global GA where the algorithm does not stop to wait for any slow processors, but it does not work exactly like a simple GA (it resembles a GA with a generation gap). Most global parallel GA implementations are synchronous and in the rest of the paper we assume that global parallel GAs do the exact same search as simple GAs.

The idea of global parallelization is not new. Actually, it has been around for quite some time. For example Bethke (1976) described parallel implementations of a conventional GA and of a GA with a generation gap and showed a very detailed analysis of the efficiency of the use of the processing capacity. His analysis showed that global parallel GAs have an efficiency close to 100%, but his analysis ignores the overhead of communications.

Another early study on global parallel GAs was made by Grefenstette (1981). He proposed four prototypes for parallel GAs. The first prototype is a global parallel GA where there is a "master" processor that does selection and applies crossover and mutation. The individuals are sent to "slave" processors to be evaluated and return to the master at the end of every generation. The second prototype is very similar to the first, but there is no clear division between generations, when any slave processor finishes evaluating an individual it returns it to the master and receives another individual. This scheme is an asynchronous global GA and can maintain a high level of processor utilization, even if the slave processors operate at different speeds. The third prototype is also a global GA, but in this case the population is stored in shared memory which can be accessed by the slaves independently of each other.

Grefenstette's fourth prototype is a coarse-grained parallel GA where the best individuals are broadcast every generation to all the other processors. The complexity of coarse-grained parallel GAs was evident from this early proposal and Grefenstette raised several "interesting questions" about the frequency of migration, the destination of the migrants (topology), and the effect of migration on preventing premature convergence.

Grefenstette also hinted at the possibility of combining the fourth prototype with any of the other three, creating a hybrid parallel GA. In section 5 we explore this possibility in more detail.

The global parallelization method does not require a particular computer architecture, and it can be implemented efficiently on shared- and distributed-memory computers. On a shared-memory multiprocessor, the population can be stored in shared memory and each processor could read a fraction of the population and write back the evaluation results without any conflicts[1]. 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

[1] We mean that there are no conflicts with other processors to access the same memory locations and thus no synchronization is required in this step, but there may be conflicts in the interconnection network that may slow the algorithm.

variable) it may be necessary to balance the computational load among the processors using a dynamic scheduling algorithm (e.g., guided self-scheduling).

On a distributed-memory computer, the population is stored in one processor. This "master" processor is be responsible for sending the individuals to the other processors (the "slaves") for evaluation, collecting the results, and applying the genetic operators to produce the next generation. The difference with a shared-memory implementation is that the master has to send and receive messages explicitly. The time used in communications is similar in both cases and the discussion that follows assumes that the global parallel GA is implemented as a master-slave program on a distributed-memory machine.

An example of global parallelization on a distributed-memory architecture is the work of Fogarty and Huang (1991). Their goal was to evolve a set of rules for a pole balancing application which takes a considerable time to simulate. They used a network of transputers which are microprocessors designed specifically for parallel computations. A transputer can connect directly to only four transputers, and when there is a need to communicate with another node in the network, all the intermediate transputers must receive and retransmit the message. This causes an overhead in communications and to try to minimize it Fogarty and Huang connected the transputers in different topologies, but they concluded that the configuration of the network did not make a significant difference. They obtained reasonable speedups and identified the fast-growing communication overhead as an impediment for further improvements in speed.

Abramson and Abela (1992) implemented a GA on a shared-memory computer (an Encore Multimax with 16 processors) to search for efficient timetables for schools. They reported limited speedups, and blamed a few sections of serial code on the critical path of the program for the results. Later, Abramson, Mills, and Perkins (1993) added a distributed-memory machine (a Fujitsu AP1000 with 128 processors) to the experiments, changed the application to train timetables, and modified the code. This time, they reported significant (and almost identical) speedups for up to 16 processors on the two computers, but the speedups degraded significantly as more processors were used, mainly due to the increase in communications.

Another implementation of a global GA was the work by Hauser and Männer (1994). They used three different parallel computers, but only got good speedups on a NERV multiprocessor (speedup of 5 using 6 processors), that has a very low communications overhead. They explained that they did not get good speedups on the other systems they used (a SparcServer and a KSR1) because they did not have complete control over the scheduling of computation threads to processors and the system sometimes made inadequate decisions.

Besides the evaluation of individuals, other aspect of GAs that can be parallelized is the application of the search operators. For example, recombination 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 would offset any 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. Recently, Branke, Andersen, and Schmeck (1997) parallelized different types of global selection on a 2-D grid of processors and showed that their algorithms

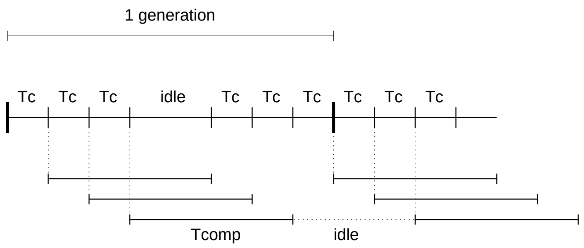


Figure 2: A schematic the execution of a global parallel GA.

are optimal for the topology of the interconnection network of the computer that they used (their algorithms require $O(\sqrt{n})$ time steps on a $\sqrt{n} \times \sqrt{n}$ grid).

In conclusion, global parallel GAs are easy to implement and seem to be a very efficient method of parallelization when the evaluation needs considerable computations. In the next section we formalize this notion and examine how the overhead from the communications affects the performance of this type of algorithms.

3 Analysis

The analysis of the execution time of global parallel GAs centers on the master processor. Figure 2 shows a timeline with the sequence of steps taken by the master. First, on each generation the master sends a fraction of the population to each of the slave processors using time T_c and waits for the results to come back from the slaves. The slaves begin evaluating their portion of the population as soon as they receive it and return it to the master as soon as they finish. When the results are ready the master receives them, using time T_c again. At the end of each cycle the master selects the parents for the next generation and creates the new individuals using crossover and mutation. The time used in these operations is assumed to be small constant and for this reason its contribution to the total execution time is ignored in this section.

The time that the master processor is idle can be computed by taking the difference between the time when the first slave completes its task ($T_c + T_{comp}$) and the time when the master finishes sending the population to the slaves (ST_c). Simplifying terms the idle time for the master is

$$idle = T_{comp} - (S-1)T_c. \tag{1}$$

where T_{comp} is the time used by each slave to evaluate its part of the population and S is the number of slave processors.

Since the population is divided into equally-sized parts among the slaves, the time that they spend in computation is simply

$$T_{comp} = \frac{n\alpha}{S}, \tag{2}$$

where α is the time required to evaluate one individual and n is the size of the population.

Now we can compute the the total execution time of the master processor as

$$T_{tot} = 2ST_c + idle = \frac{n\alpha}{S} + (S+1)T_c. \tag{3}$$

From this equation for the total time, it is evident that as more slaves are used the computation time decreases, but at the same time, the communications time increases. This tradeoff entails the existence of an optimal number of slaves that minimizes the execution time. Making $\frac{\partial T_{tot}}{\partial S} = 0$ and solving for S results in

$$S^* = \sqrt{\frac{n\alpha}{T_c}}. \tag{4}$$

However, this last equation assumes that T_c is constant with respect to the number of slaves. More often the cost of exchanging information between two processors depends linearly on the amount of information, and in our case the amount of information depends on the number of slaves. Therefore a better expression for T_c is

$$T_c = A\text{size} + C, \tag{5}$$

where A and C are hardware-dependent constants. C is a fixed overhead cost associated with any communication and in our case size $= nl/S$ is the size of the n/S individuals sent to each slave when each individual is represented with l bytes.

Using this linear model for the communications time also results in a tradeoff and the optimal number of slaves can be computed in the same manner as before resulting in

$$S^* = \sqrt{\frac{(A+\alpha)n}{C}}. \tag{6}$$

An important consideration in implementing global parallel GAs is that the frequent communication may offset any gains in computation time. The time that a simple GA uses in one generation is $T_s = n\alpha$ and to ensure that the parallel implementation has a better performance than a simple GA the following relationship must hold:

$$\frac{T_s}{T_{tot}} = \frac{n\alpha}{\frac{n\alpha}{S} + (S+1)T_c} > 1. \tag{7}$$

This ratio is the parallel speedup of the global parallel GA. Solving for α results in a necessary condition for better performance in the parallel case:

$$\alpha > \frac{S+1}{S-1}\frac{S}{n}T_c \approx \frac{S}{n}T_c. \tag{8}$$

This condition is easy to check without implementing a global parallel GA by measuring α and T_c. Since it is likely that T_c depends linearly on the size of the information to be transmitted, we can substitute equation 5 in the previous inequality to obtain a more specialized condition

$$\alpha > Al + \frac{S}{n}C.$$

This condition implies that for simple problems (with a short evaluation time) global parallel GAs are not an option to reduce the execution time. But for problems with long execution times global parallel GAs show a great improvement in performance. The next section presents experiments using functions with different evaluation times and shows that the theory presented here predicts the performance gains accurately.

4 Experiments

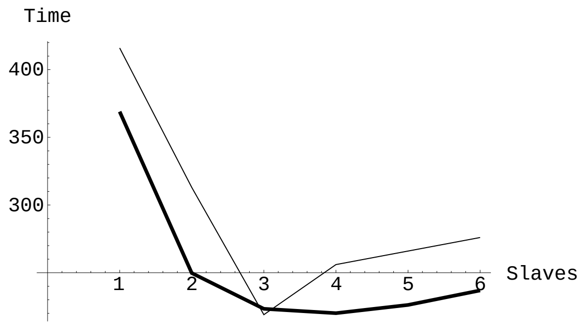
This section describes a particular master-slave implementation of global parallel GA on a network of IBM RS6000 workstations. The workstations are connected with a 10 Mbits/sec Ethernet and all communications are implemented using PVM 3.3. This is a rather slow communications environment, so we do not expect any performance improvements when simple test functions are used. For this reason we used first an artificial function that could be altered easily to change its evaluation time (α). We also experimented with a complex neural network application that takes a very long time to evaluate.

The first problem is a dummy function that consists in a simple loop with a single addition that can be repeated an arbitrary number of times. The length of the individuals was set to 80 bytes and the population size to 120 individuals. The global parallel GA was executed for 10 generations and the results reported are the average of 30 runs of the elapsed time. We determined empirically that the communication time in our system could be modeled as $T_c = 20 + 0.00526x$ (in milliseconds) where x is the size in bytes of the message.

For the first experiment the evaluation time of the test function was set to 1.9 milliseconds. Using the analysis presented in the previous section we can calculate the optimal number of slaves as $S^* = \sqrt{\frac{(A+\alpha)n}{C}} = \sqrt{\frac{(1.90526)120}{20}} = 3.38$. Figure 3 shows the elapsed time per generation of the global parallel GA along with the theoretical prediction (using equation 3). The first row of the table is the elapsed time that a serial GA takes to evaluate the population. Note that the master-slave algorithm was faster than the serial GA only when three slaves are used.

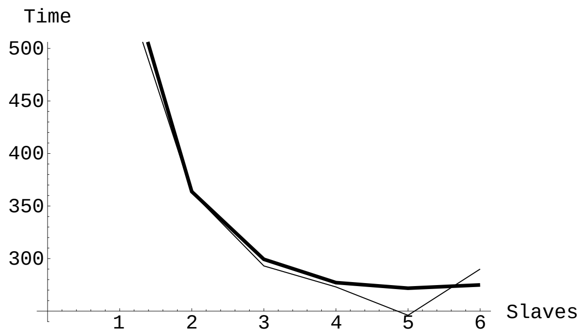
In the second experiment we changed the evaluation time of the test function to 3.8 milliseconds and the results for this experiment are shown in figure 4. For this experiment the optimal number of slaves was $S^* = \sqrt{\frac{(3.80526)120}{20}} = 4.78$. We doubled the evaluation time again for the third experiment ($\alpha = 7.6$ ms) and the results are shown in figure 5. In this case $S^* = 6.75$ and therefore the optimal point does not appear in the experiments. From these results we see that the theoretical time matches the experiments quite well and that the optimum number of slaves is predicted accurately by equation 6.

The last experiment used a more complex evaluation function. It was an application where the GA searched for the weights of the connections of a neural network with 13 inputs, 30 units in the hidden layer, and 5 outputs. The objective was to classify a set of 738 vectors that represent sleep patterns into 5 classes. The evaluation function decoded the weights from a string of 5150 bytes, tested each of the patterns, and calculated the percentage of the classifications made correctly. On our computers each evaluation takes 3.85 seconds to complete, which makes this problem an ideal candidate for the global parallelization method. As we can see in figure 6 the total elapsed time decreased linearly with the number of slaves because the time used in communications was only a small fraction of the total time. In



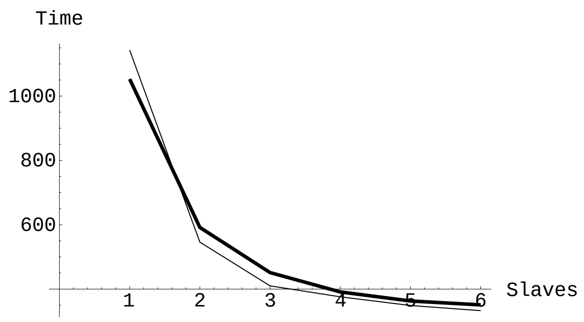
Slaves	Theory	Exp.
0	228	246
1	368	416
2	249	313
3	223	**219**
4	**220**	256
5	226	266
6	237	276

Figure 3: Elapsed time (ms) per generation for the 1.9 milliseconds problem. The thick line on the plot is the theoretical predictions and the thin line is the experimental results. On the table, the numbers with the bold typeface are the minimum execution times.



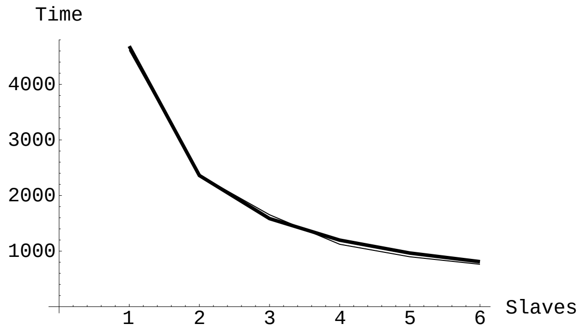
Slaves	Theory	Exp.
0	456	459
1	596	573
2	363	363
3	299	293
4	277	273
5	**271**	**246**
6	275	290

Figure 4: Elapsed time (ms) per generation for the 3.8 milliseconds problem.



Slaves	Theory	Exp.
0	912	921
1	1053	1143
2	591	546
3	451	410
4	391	376
5	362	349
6	350	333

Figure 5: Elapsed time (ms) per generation for the 7.6 milliseconds problem.



Slaves	Theory	Exp.
1	4620	4615
2	2310	2366
3	1540	1654
4	1156	1123
5	925	898
6	771	763

Figure 6: Elapsed times (sec) per generation for the NN problem.

fact, the communications overhead was so small that for this problem $S^* = 152$ processors.

5 Hybrid parallel algorithms

The previous sections show that adding more slaves to a global parallel GA may improve the performance greatly. However, the improvement is not infinite and after a certain optimal number of slaves the performance will decrease as more processors are used. The question that we face now is how should we employ other processors that might be available in a constructive way?

This section explores the possibility of combining global GAs with coarse-grained parallel GAs to achieve better performance than with any of the two methods alone. Coarse-grained GAs are a different class of parallel GAs that have separate demes (or subpopulations) that exchange individuals occasionally. It is possible to combine global GAs with coarse-grained GAs to form a kind of hybrid-parallel GA (Cantú-Paz, 1997). In this class of hybrid the two methods of parallelizing GAs form a hierarchy with the coarse-grained algorithm at the upper level (see figure 7).

The behavior of coarse-grained GAs is controlled by many variables and for this reason their analysis presents many difficulties. However, the size of the demes (or subpopulations) plays a major role as it is the principal factor in determining the quality of the final solution and the time that the GA needs to find it. Recently, Cantú-Paz and Goldberg (1997a) developed a theory to determine the size of the demes that is needed to reach a solution of a desired quality for two bounding cases of coarse-grained parallel GAs. The two bounding cases are a set of isolated demes and a set of fully connected demes. In the case of the connected demes the migration rate is set to the highest value possible.

Combining the deme-sizing models with a general model for the communications time Cantú-Paz and Goldberg (1997b) predicted the expected parallel speedups for the two bounding cases. They reached two major conclusions from their analysis of coarse-grained GAs. First, the speedup that is expected in the case where the demes execute in complete isolation is not very significant. Second, in the case where the demes communicate, there is an optimal number of demes (and an associated deme size) that maximizes the speedup (see figure 8).

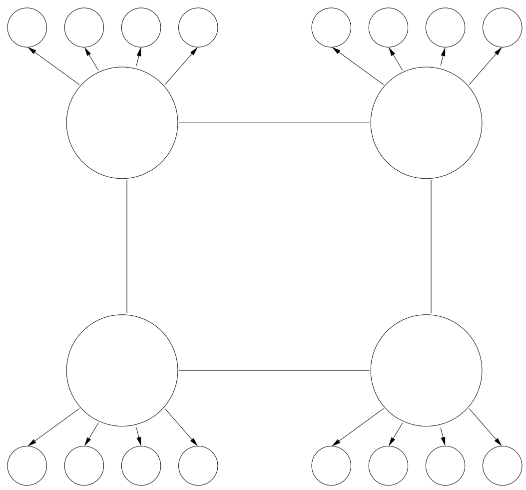


Figure 7: A schematic of a hybrid parallel GA. At the higher level this hybrid is a coarse-grained parallel GA where each node is a global parallel GA.

The important lesson is that both coarse-grained and global parallel GAs have an optimal number of processors after which the performance will be reduced. By using a hybrid it is possible to use additional processors and reduce the execution time beyond the limits of any of the two methods. Bianchini and Brown (1993) present an example of this hybrid method and show experimentally that it can find solutions of the same quality of a global parallel GA or a conventional coarse-grained GA in less time.

6 Summary and Conclusions

As GAs are applied to larger and more difficult search problems it becomes necessary to design faster algorithms that retain the capability of finding acceptable solutions. Global parallel GAs search the space in the same manner as a conventional GA and in consequence retain all their advantages. Also, global GAs are easy to implement, but require a constant exchange of individuals between the master and the slave processors. This communication overhead reduces the performance gained by partitioning the computation load among several processors. The analysis contained in this report shows that in realistic situations there is an optimal number of slave processors that minimizes the execution time of the global parallel GA. The analysis results in simple expressions that predict accurately the execution time and the optimal number of slaves.

The analysis also shows a necessary condition for an increased performance in the parallel case. The importance of this condition is that it is easy to evaluate without implementing a global parallel GA. Simple tests to determine the cost of evaluating one individual and the constants for the communications model are sufficient.

The existence of an optimal number of slaves limits the number of processors that can be used to decrease the execution time. To overcome this limit, we propose to use a hybrid GA with a coarse-grained parallel GA at the upper level and global GAs at the deme level.

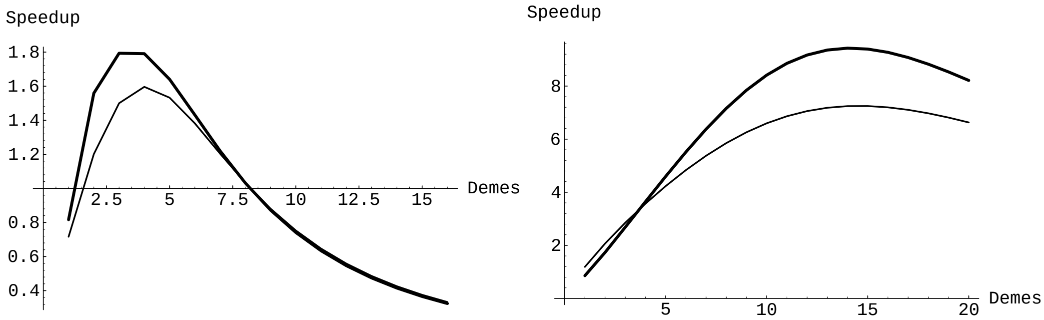


Figure 8: Projected and experimental speedups for test functions with 20 copies of a 4-bit trap (left) and 8-bit trap (right) functions using from 1 to 16 *fully connected* demes. The quality requirement was to find at least 16 copies correct. The thick lines are the theoretical predictions and the thin lines are the experimental results.

Acknowledgments

I wish to thank David E. Goldberg for his useful comments on the material presented here.

This study was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number 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.

Erick Cantú-Paz was supported by a Fulbright-García Robles-CONACYT Fellowship.

References

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

Bethke, A. D. (1976). *Comparison of genetic algorithms and gradient-based optimizers on parallel processors: Efficiency of use of processing capacity* (Tech. Rep. No. 197). Ann Arbor, MI: University of Michigan, Logic of Computers Group.

Bianchini, R., & Brown, C. M. (1993). Parallel genetic algorithms on distributed-memory architectures. In Atkins, S., & Wagner, A. S. (Eds.), *Transputer Research and Applications 6* (pp. 67–82). Amsterdam: IOS Press.

Branke, J., Andersen, H. C., & Schmeck, H. (1997, January). *Parallelising global selection in evolutionary algorithms.* Submitted to Journal of Parallel and Distributed Computing.

Cantú-Paz, E. (1997). *A survey of parallel genetic algorithms* (IlliGAL Report No. 97003). Urbana, IL: University of Illinois at Urbana-Champaign.

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). Modeling speedups of idealized bounding cases of parallel genetic algorithms. In Bäck, T. (Ed.), *Proceedings of the Seventh International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann Publishers.

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

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.

Harik, G., Cantú-Paz, E., Goldberg, D., & Miller, B. (1997). The gambler's ruin problem, genetic algorithms, and the sizing of populations. In Bäck, T. (Ed.), *Proceedings of the Fourth International Conference on Evolutionary Computation* (pp. 7–12). New York: IEEE Press.

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.