Massive parallelization of the compact genetic algorithm

Fernando G. Lobo, Cláudio F. Lima, Hugo Mártires
DEEI-FCT, Universidade do Algarve
Campus de Gambelas, 8000-062 Faro, Portugal
E-mail: {flobo,clima}@ualg.pt, hmartires@myrealbox.com

Abstract

This paper presents an architecture which is suitable for a massive parallelization of the compact genetic algorithm. The resulting scheme has three major advantages. First, it has low synchronization costs. Second, it is fault tolerant, and third, it is scalable.

The paper argues that the benefits that can be obtained with the proposed approach is potentially higher than those obtained with traditional parallel genetic algorithms.

1 Introduction

One of the efficiency enhancement techniques that has been investigated in the field of evolutionary computation, both in theory and in practice, is the topic of parallelization [1]. With a traditional parallel genetic algorithm (GA) implementation, population members need to be sent over a computer network, and that imposes a limit on how fast they can be. This paper addresses the parallelization of the compact genetic algorithm (cGA) [2], and take advantage of its compact representation of the population do develop a scheme which significantly reduces the communication overhead.

The paper is organized as follows. The next section presents background material on parallel GAs and section 3 reviews the cGA. Section 4 shows the motivation for parallelizing the cGA and presents an architecture that allows its massive parallelization. In section 5 computer experiments are conducted and its results are discussed. Finally, a number of extensions are outlined, and we finish with a brief summary and the main conclusions of this work.

2 Parallel GAs

An important efficiency question that people are faced with in problem solving is the following: Given a fixed computational time, what is the best way to allocate computer resources in order to have as good a solution as possible. Under such a challenge, the idea of parallelization stands out naturally as a way of improving the efficiency of the problem solving task. By using multiple computers in parallel, there is an opportunity for delivering better solutions in a shorter period of time.

Several researchers have investigated the topic of parallel GAs and the major design issues are in choices such as using one or more populations, and in the case of using multiple populations, decide when, with whom, and how often do individuals communicate with other individuals of other populations.

Although implementing parallel genetic algorithms is relatively simple, the answers to the questions raised above are not straightforward and traditionally have only been answered by means of empirical experimentation. One exception to that has been the work of Cantú-Paz [1] who has built theoretical models that lead to rational decisions for setting the different parameters involved in parallelizing GAs. There are two major ways of implementing parallel GAs: (1) using a single population, and (2) using multiple populations.

In single population parallel GAs, also called Master-Slave parallel GAs, one computer (the master) executes the GA operations and distributes individuals to be evaluated by other computers (the slaves). After evaluating the individuals, the slaves return the results back to the master. There can be significant benefits with such a scheme because the slaves can work in parallel, independently of one another. On the other hand, there is an extra overhead in communication costs that must be paid in order to communicate individuals and fitness values back and forth.

In multiple population parallel GAs, what would be a whole population in a regular non-parallel GA, becomes several smaller populations (usually called demes), each of which is located in a different computer. Each computer executes a regular GA and occasionally, individuals may be exchanged with individuals from other populations. Multiple population parallel GAs are much harder to design because there are more degrees of freedom to explore. Specifically, four main things need to be chosen: (1) the size of each population, (2) the topology of the connection between the populations, (3) the number of individuals that are exchanged, and (4) how often do the individuals exchange.

Cantú-Paz investigated both approaches and con-

cluded that for the case of the Master-Slave architecture, the benefits of parallelization occur mainly on problems with long function evaluation times because it needs constant communication. Multiple population parallel GAs have less communication costs but do not avoid completely the communication scalability problem. In other words, in either approach, communication costs impose a limit on how fast parallel GAs can be.

The next section gives an overview of the cGA, and after that, its parallelization is discussed.

3 The Compact Genetic Algorithm

Harik et al. [2] noticed that it was possible to mimic the behavior of a simple GA without storing the population explicitly. Such observation came from the fact that during the course of a regular GA run, alleles compete with each other at every gene position. At the beginning, scanning the population column-wise, we should expect to observe that roughly 50% of the alleles have value 0 and 50% of the alleles have value 1. As the search progresses, for each column, either the zeros take over the ones, or vice-versa. Harik et al. built an algorithm that explicitly simulates the random walk that takes place on the allele frequency makeup for every gene position. The resulting algorithm, the cGA, was shown to be operationally equivalent to a simple GA that does not assume any linkage between genes.

Under the cGA, the population is represented by a probability vector. The elements of the vector are the relative frequency counts of the number of 1's for the different gene positions. The cGA manipulates the population in an indirect way through an update step in each allele frequency of the probability vector. Notice that each allele frequency value is a member of a finite set of $N + 1$ possible values, and can be stored with $\log_2(N + 1)$ bits. (N denotes the population size of a regular GA). Instead, a regular GA would require N bits to represent each bit position. Further details about the algorithm can be found in the original source [2].

4 Massive parallelization of the compact GA

The main motivation for parallelizing the cGA comes from the observation that the probability vector is a compact representation of the population, and it is possible to communicate the vector rather than individuals themselves. Communication costs can be reduced this way because the probability vector needs significant less storage than the whole population. This observation has first been made by Harik [3] when the cGA was developed.

Since communication costs can be drastically reduced, it makes sense to clone the probability vector to several computers, and each computer can work independently on solving a problem by running a separate cGA. Then, the different probability vectors would have to be consolidated (or mixed) once in a while.

We have developed an asynchronous parallelization scheme which consists of a manager processor, and an arbitrary number of worker processors (see Figure 1). Initially, the manager starts with a probability vector with 0.5 in all positions, just like in a regular cGA. After that, it sends the vector to all workers who are willing to contribute with CPU time.

Each worker processor runs a cGA on its own based on a local copy of the probability vector. Workers do their job independently and only interrupt the manager once in a while, after a predefined number of m fitness function evaluations have elapsed.

During the interruption period, a worker sends the accumulated results of the last m function evaluations as a vector of probability fluxes with respect to the original probability vector. Subsequently, the manager adds the probability fluxes (values are truncated so that they never exceed 1.0 and never go below 0.0) to its own probability vector, and resends the resulting vector back to the worker. Notice that the manager's probability vector not only incorporates the results of the m function evaluations performed by that particular worker, but it also incorporates the results of the evaluations conducted by the other workers. That is, while a particular worker is working, other workers might be updating the manager's probability vector. Thus, at a given point in time, workers are working with a slightly outdated probability vector. Although this might seem a disadvantage at first sight, the error that is committed by working with a slightly outdated probability vector is likely to be negligible for the overall search because an iteration of the cGA represents only a small step in the action of the GA (this is especially true for large population sizes).

One could think of different ways of parallelizing the cGA but the scheme that we are proposing is particularly attractive because once the manager starts, there can be an arbitrary number of workers, each of which can start and finish at any given point in time making the whole system fault tolerant. When a worker starts, it receives a copy of the manager's probability vector, which already contains the accumulated results of the other cGA workers. On the other hand, when a worker quits, we simply loose a maximum of m function evaluations, which is not a big problem.

The proposed parallelization scheme has several advantages: (1) it has low synchronization costs, (2) it is fault tolerant, and (3) it is scalable.

All the communication that takes place consists of short transactions. Workers do their job independently and only interrupt the manager once in a while. During the interruption period, the manager communicates

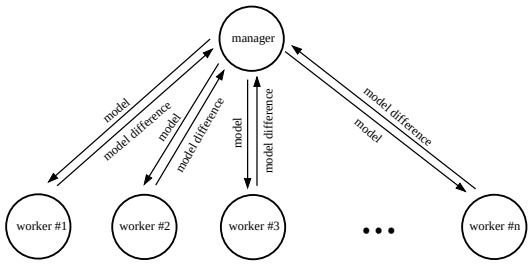

Fig. 1. Manager-worker architecture.

with a single worker, and the other workers can continue working non-stop.

The architecture is fault tolerant because workers can go up or down at any given point in time. This makes it suitable for massive parallelization using the Internet. It is scalable because potentially there is no limit on the number of workers.

5 Computer simulations

This section presents computer simulations that were done to validate the proposed approach. In order to simplify both the implementation and the interpretation of the results, we decided to do a serial implementation of the parallel cGA architecture. The serial implementation simulates that there are a number of P worker processors and 1 manager processor. The P worker processors start running at the same time and they all execute at the same speed. In addition, it is assumed that the communication cost associated with a manager-worker transaction takes a constant time which is proportional to the probability vector's size. Such a scheme can be implemented by having a collection of P regular compact GAs, each one with its own probability vector, and iterating through all of them, doing a small step of the cGA main loop, one at a time. After a particular cGA worker completes m fitness function evaluations, the worker-manager communication is simulated as described in section 4.

We present experiments on a bounded deceptive function consisting of the concatenation of 10 copies of a 3-bit trap function with deceptive-to-optimal ratio of 0.7 [4]. This same function has been used in the original cGA work. We simulate a selection rate of $s = 8$ and did tests with a population size of $N = 100000$ individuals (each worker processor runs a cGA that simulates a 100000 population size). We chose this population size because we wanted to use a size large enough to solve all the building blocks correctly. We use $s = 8$ following the recommendation given by Harik et al. in the original cGA paper for this type of problem. Finally, we chose this problem as a test function because, even

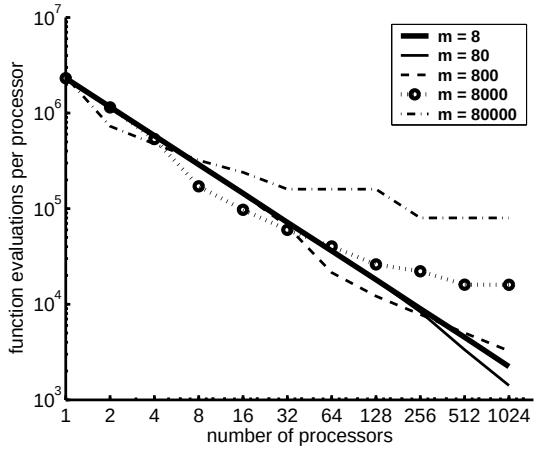

Fig. 2. Function evaluations per processor.

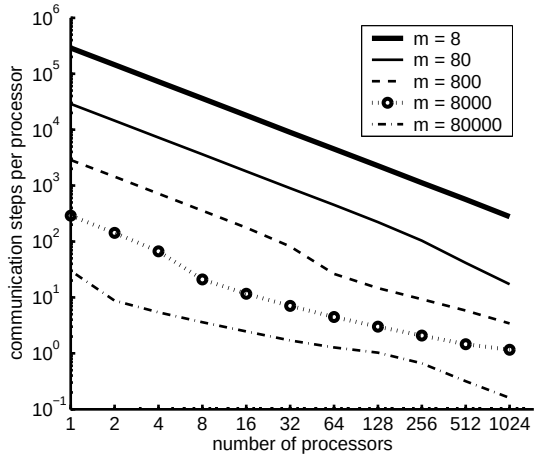

Fig. 3. Communication steps per processor.

though the cGA is a poor algorithm in solving the problem, we wanted to use a function that requires a large population size because those are the situations where the benefits from parallelization are more pronounced.

Having fixed both the population size and the selection rate, we decided to systematically vary the number of worker processors P, as well as the m parameter which has an effect on the rate of communication that occurs between the manager and a worker. We did experiments for P in $\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$, and for a particular P, we varied the parameter m in $\{8, 80, 800, 8000, 80000\}$. This totalled 55 different configurations, each of which was run 30 independent times.

The m parameter is important because it is the one that affects communication costs. Smaller m values imply an

increase in communication costs. On the other hand, for very large m values, performance degrades because the cGA workers start sampling individuals from outdated probability vectors.

Figures 2 and 3 show the results. In terms of fitness function evaluations per processor, we observe a linear speedup for low m values. For instance, for $m = 8$ we observe a straight line on the log-log plot. Using the data directly, we calculated the slope of the line and obtained an approximate value of -0.3. In order to take into account the different logarithm bases, we need to multiply it by $\log_2 10$ (y-axis is $\log_{10}$, x-axis is $\log_2$) yielding a slope of approximately -1. This means that the number of function evaluations per processor decreases linearly with a growing number of processors. That is, whenever we double the number of processors, the average number of fitness function evaluations per processor gets cut by a half. Likewise, in terms of communication costs, as we raise the parameter m, the average number of communication steps between manager and worker decreases in the same proportion as expected. For instance, for $m = 80$, communication costs are reduced 10 times when compared with $m = 8$. Notice that there is a degradation in terms of speedup for the larger m values. For instance, for $m = 8000$ and $m = 80000$ (which is about the same order of the population size), the speedup obtained goes away from the idealized case. This can be explained by the fact that in this case (and especially with a large number of processors), the average number of communication steps per processor approaches zero. That means that a large fraction of processors were actually doing some work but never communicated their results back to the manager because the problem was solved before they had a chance to do so.

6 Extensions

It would be interesting to do a mathematical analysis of the proposed parallel cGA. A number of questions come to mind. For instance, what is the effect of the m parameter? What about the number of workers P? Should m be adjusted automatically as a function of P and N? Our experiments suggest that there is an "optimal" m that depends on the number of cGA workers P, and most likely depends on the population size N as well.

Another extension that could be done is to compare the proposed parallel architecture with those used more often in traditional parallel GAs, either master-slave and multiple deme GAs. Again, our experiments suggest that the parallel cGA is likely to be on top of regular parallel GAs due to lower communication costs.

The model structure of the cGA never changes, every gene is always treated independently. There are other probabilistic model building genetic algorithms (PMBGAs) [5] which are able to learn a more complex structure dynamically as the search progresses. One could think of using some of the ideas presented here for parallelizing these more complex PMBGAs.

Finally, it would be interesting to have a parallel cGA implementation based on the Internet infrastructure, where computers around the world could contribute with some processing power when they are idle. Similar schemes have been done with other projects, one of the most well known is the SETI@home project [6].

7 Summary and conclusions

This paper reviewed the compact GA and presented an architecture that allows its massive parallelization. The motivation for doing so has been discussed and a serial implementation of the parallel architecture was simulated. Computer experiments were done under idealized conditions and we have verified an almost linear speedup with a growing number of processors.

The paper presented a novel way of parallelizing GAs. This was possible due to the different operational mechanisms of the cGA when compared with a more traditional GA. By taking advantage of the compact representation of the population, it becomes possible do distribute its representation to different computers without the associated cost of sending it individual by individual.

Acknowledgements

This work was sponsored by FCT/MCES under grant POSI/SRI/42065/2001.

References

[1] Cantú-Paz, E. (2000) Efficient and accurate parallel genetic algorithms. Kluwer Academic Publishers, Boston, MA.

[2] Harik, G.R., Lobo, F.G., Goldberg, D.E. (1999) The compact genetic algorithm. IEEE Transactions on Evolutionary Computation **3**, pp. 287–297

[3] Harik, G.R. (1997) Personal communication.

[4] Deb, K., Goldberg, D.E. (1993) Analyzing deception in trap functions. In Whitley, L.D., ed.: Foundations of Genetic Algorithms 2, San Mateo, CA, Morgan Kaufmann, pp. 93–108

[5] Pelikan, M., Goldberg, D.E., Lobo, F. (2002) A survey of optimization by building and using probabilistic models. Computational Optimization and Applications **21**, pp. 5–20

[6] Korpela, E., Werthimer, D., Anderson, D., Cobb, J., Lebofsky, M. (2001) SETI@home - massively distributed computing for SETI. Computing in Science and Engineering **3** pp. 79