Analyzing synchronous and asynchronous parallel distributed genetic algorithms

Enrique Alba*, José M. Troya

Dpto. de Lenguajes y Ciencias de la Computación, Univ. de Málaga, Campus de Teatinos (2.2.A.6), 29071 Málaga, Spain

Abstract

Parallel genetic algorithms (PGAs) have been traditionally used to extend the power of serial genetic algorithms (GAs), since they often can be tailored to provide a larger efficiency on complex search problems. In a PGA several sub-algorithms cooperate in parallel to solve the problem. This high-level definition has led to a considerable number of different implementations that preclude direct comparisons and knowledge exchange. To fill this gap we begin by providing a common framework for studying PGAs. We then analyze the importance of the synchronism in the migration step of various parallel distributed GAs. This implementation issue could affect the evaluation effort as well as could provoke some differences in the search time and speedup. We cover in this study a set of popular evolution schemes relating panmictic (steady-state or generational) and structured-population (cellular) GAs for the islands. We aim at extending existing results to structured-population GAs, and also to new problems. The evaluated PGAs demonstrate linear and even super-linear speedup when run in a cluster of workstations. They also show important numerical benefits if compared with their sequential versions. In addition, we always report lower search times for the asynchronous versions. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Asynchronous parallel GAs; Cellular GAs; Numeric performance; Speedup; Selection pressure

1. Introduction

Genetic algorithms (GAs) are stochastic search methods that have been successfully applied in many search, optimization, and machine learning problems [3,12]. Unlike most other optimization techniques, GAs maintain a population of encoded tentative solutions that are competitively manipulated by applying some variation operators to find a global optimum.

A sequential GA (Fig. 1) proceeds in an iterative manner by generating new populations of strings from the old ones. Every string is the encoded (binary, real,...) version of a tentative solution. An evaluation function associates a fitness value to every string indicating its suitability to the problem. The canonical algorithm applies stochastic operators such as selection, crossover, and mutation on an initially random population in order to compute a whole generation of new strings [12,16].

For non-trivial problems this process might require high computational resources (e.g., large memory and search times), and thus, a variety of algorithmic issues are being studied to design efficient GAs. With this goal in mind, numerous advances are continuously being achieved by designing new operators [10], hybrid algorithms [11], and more. We focus on one of such improvements consisting in using parallel models of GAs (PGAs) [2,6,8].

Several arguments justify our work. First of all, parallel genetic algorithms (PGAs) are naturally prone to parallelism [6] since the operations on the strings can
be easily undertaken in parallel (not so easy for the selection method, since it usually uses the full population). Even the whole population (panmixia) can be geographically structured to localize competitive selection between subsets of the population, thus sustaining a high diversity to reliably converge to optimal solutions [25,28,29]. The evidences of a higher efficiency [4,13], larger diversity maintenance, additional availability of memory/CPU, and their multi-solution capabilities, reinforce the importance of the research advances with PGAs [14].

Using a PGA often leads to superior numerical performance and not only to a faster algorithm. Gordon and Whitley [13] clearly stated this superiority, which has been reported also by Hart et al. [14] and many others [20,24,28]. However, the truly interesting observation is that the use of a structured population either in the form of a set of islands [28] or a diffusion grid [23] is responsible for such numerical benefits. As a consequence, many authors do not use a parallel machine at all to run spatially structured-population models, and still get better results than with traditional serial GAs [13,15].

In many PGA works and papers it is assumed that the search model maps directly onto the parallel hardware, thus making no distinction between the model and its implementation. However, once a structured-population model has been defined, it can be implemented in any monoprocessor or parallel machine (with more or less success). There exist numerous examples of the traditional vision of parallel GAs, such as performing a sequential-like GA by parallelizing the evaluations and the operators (called global parallelization), or running a ring of panmictic GAs, both on a MIMD computer [2,4,28]. More recent works confirm the new vision, e.g. executing a grid of individuals on: SISD [13], MIMD [14], or SIMD computers [25]. Finally, many hybrids have been proposed [8,15,20]. See some examples of these and other models in Fig. 2.

This separate vision of model vs. implementation raises several questions. First, any GA can be run in parallel, although not all of them could show linear speedup. In this sense, our contribution is to extend the existing work on distributed GAs (dGA) from generational island evolution to steady-state [27], and cellular algorithms [30]. These two latter models are worth to study due to their reported higher benefits in many applications [1,25,31].

Second, this suggests the necessity of using a difficult and heterogeneous test suite in order to avoid the misleading conclusions that might be got if a trivial benchmark were used. In this sense, we use a test suite composed of multimodal, deceptive, and epistatic problems. Therefore, we include some of the main difficulties that can be found in complex and real-life problems and analyze them from various complementary points of view.

Third, the experiments should be replicable to help future extensions of our work. This led us to using a readily available parallel hardware such as a cluster of workstations in order to spread out the utilization of parallel distributed GAs. Also, this paper contains parameter and technique descriptions to allow other researchers replicate, study and improve our results (making the paper ‘self-contained’). Besides that, references have been strategically inserted to reinforce our initial problem statements and conclusions, as well as allowing new researchers to get a full picture of parallel GAs.

Finally, some questions are open in relation to the physically parallel execution of the models. Since we focus on distributed GAs with panmictic or structured populations that run in parallel on separate machines, this means that we need to study decisive implementation issues such as the synchronism in their communication steps [14,19]. Other directly related techniques such as the used communication frequency in a parallel distributed GA are also addressed here. Also, the selection pressure and diversity are dealt with in this paper to know better their basic working principles. Our analysis takes into account fundamental metrics for these algorithms such as numerical performance, runtime, and speedup.

The high number of non-standard or machine-dependent PGAs has led to efficient algorithms in many domains. However, these unstructured approaches often hide their canonical behavior, thus
Fig. 2. Different models of PGA: (a) global parallelization, (b) coarse grain, and (c) fine grain. Many hybrids have been defined by combining PGAs at two levels: (d) coarse, and fine grain, (e) coarse grain and global parallelization, and (f) coarse grain plus coarse grain.

making it difficult to forecast further behavior and knowledge exchange. This is the reason for our distinction between structured-population GAs and their parallel implementations. Also, this is why Section 2 begins by presenting a common framework for proposing and studying PGAs. This common framework has many implications in the field of the theoretical study of PGAs, as well as in relation to their implementation (e.g. object oriented) and hybridization, although in this paper we use it only in order to structure the tests and the conclusions.

The paper is organized as follows. In Section 2 we present sequential and parallel GAs as especial cases of a more general meta-heuristic. Also, this is why Section 2 begins by presenting a common framework for proposing and studying PGAs. This common framework has many implications in the field of the theoretical study of PGAs, as well as in relation to their implementation (e.g. object oriented) and hybridization, although in this paper we use it only in order to structure the tests and the conclusions.

The paper is organized as follows. In Section 2 we present sequential and parallel GAs as especial cases of a more general meta-heuristic. In Section 3 we describe the used benchmark. Section 4 presents a study on the search time of synchronous/asynchronous versions for different values of the migration frequency and problems. Section 5 contains an analysis from the point of view of the speedup. In Section 6 we have included some internal details on the expected selection pressure of various models as well as on their decreasing diversity curves. Finally, Section 7 offers a summary and some concluding remarks.

2. Parallel genetic algorithms

In this section we want to show, briefly, how coarse (cgPGA) and fine grain PGAs (fgPGA) are subclasses of the same kind of PGA consisting of a set of communicating sub-algorithms. In addition, we will define the algorithms we are going to use in the analysis.

2.1. A common framework

In order to deal with parallel decentralized GAs we begin by proposing a change in the nomenclature to call them distributed and cellular GAs (dGA and cGA), since the grain is usually intended to refer to their computation/communication ratio, while further differences can also be found in the way in which they both structure their populations (see Fig. 3).

While a distributed GA has large sub-populations (≥1) a cGA has typically only one string in every sub-algorithm. For a dGA the sub-algorithms are loosely connected, while for a cGA they are tightly connected. In addition, in a dGA there exist only a few sub-algorithms, while in a cGA there is a large number of them. This approach to PGAs is directly or explicitly supported in only a few real implementations.
of parallel software for GAs. However, it is very important since it covers the majority of the present variety of parallel software and structured-population models (better than calling them “parallel” models), therefore allowing knowledge exchange. By considering different codings, operators, and communication details, we could explain the full spectrum of parallel implementations.

In fact, this is a natural vision of PGAs that is inspired in the initial work of Holland [16], and more recently of Daida et al. [11], in which a granulated adaptive system is a set of grains with shared structures. Every grain has its own reproductive plan and internal/external representations for its structures. This scheme has been very useful for providing a general framework of PGAs [9], and even for a hybrid cooperation of sub-algorithms. It has many implications in relation to computability, and other advances in evolutionary algorithms (EAs). In this work we only mention it to justify the interest for using a distributed GA with panmictic or spatially structured population.

In the rest of this section we define more formally sequential and parallel GAs by following this unification principle. They both can be thought of as being useful to find an admissible vector of free parameters for which an arbitrary quality criterion is optimized:

\[
f(x) \rightarrow \max : \text{find } x^* \text{ such that } \forall x \in M : f(x) \leq f(x^*) = f^*.
\]  

(1)

The outline of a general PGA is presented in Algorithm 1. It begins by randomly creating a population \( P(t=0) \) of \( \mu \) structures ‘strings’, each one encoding the \( p \) problem variables, usually as a vector over \( \mathbb{B} = \{0,1\}^p \) or \( \mathbb{R} = \mathbb{R}^p \). An evaluation function \( \Phi \) is needed to associate a quality real value to every structure.

![Algorithm 1 (Parallel genetic algorithm).](image)

\( t = 0; \)
\begin{align*}
\text{initialize:} & \quad P(0) = \{x_0(0), \ldots, x_\mu(0)\} \in \mathbb{B}^\mu; \\
\text{evaluate:} & \quad P(0) : \Phi(x_0(0), \ldots, x_\mu(0)); \\
\text{while} & \quad t(P(t)) \neq \text{true} \quad \# \text{Reproductive Loop}\; \\
& \quad \text{select:} \quad x'(t) = s_{\Theta}(P(t)); \\
& \quad \text{recombine:} \quad x'(t) = \Theta_{\Theta}(P(t)); \\
& \quad \text{mutate:} \quad x'(t) = m_{\Theta}(P(t)); \\
& \quad \text{evaluate:} \quad x'(t) : \Phi(x_0(0), \ldots, x_\mu(0)); \\
& \quad \text{replace:} \quad P(t+1) = r_{\Theta}(P(t) \cup Q); \\
\end{align*}

In a parallel GA there exist many elementary GAs (grains) working on separate sub-populations \( P_i(t) \).
Each sub-algorithm includes an additional phase of periodic communication with a set of neighboring sub-algorithms located on some topology.

This communication usually consists in exchanging a set of individuals, although nothing prevents the sub-algorithms of exchanging another kind of information such as population statistics. However the sub-algorithms are all performing the same reproductive plan. Otherwise the PGA is heterogeneous [2,15]. Even the representation could differ among the sub-algorithms, posing new challenges to the exchange of individuals.

2.2. Algorithms and parameterizations

In connection with the previous description, our steady-state panmictic algorithm (Fig. 5a) generates one single individual in every iteration. It is inserted back in the population only if it is better (larger fitness) than the worst existing individual.

In all the cGAs we use (Fig. 5b), a NEWS neighborhood is defined (North–East–West–South in a toroidal grid [23]) in which overlapping demes of 5 strings (4 + 1) execute the same reproductive plan. In every neighborhood (deme) the new string computed after
selection, crossover, and mutation replaces the current one only if it is better (binary tournament). This process is repeated for all the neighborhoods in the grid of the cGA in every generation (there are as many neighborhoods as strings).

In parallel dGAs (Fig. 5c shows an MIMD implementation) the loosely coupled sub-populations have a considerable size ($\mu \geq 1$). Throughout this paper we deal with homogeneous dGAs in which migrants are selected randomly, and the target island replaces its worst string with the incoming one only if it is better. The sub-algorithms are disposed in a unidirectional ring since it has an easy implementation and other advantages [7]. Every island is run on a separate Ultra-SPARC 1 processor in a cluster linked by an ATM communication network.

We define the migration frequency in terms of the number of evaluations made in the island, and not in terms of the number of generations, since we are comparing models of different basic steps. A distributed GA with $d$ steady-state islands must take $\mu/d$ steps to complete one generation, while a distributed cGA would compute a full new generation of $\mu$ individuals in every parallel iteration. Anyway, in all the cases we will study the frequency of migrations in multiples of the global population size: $1\mu$, $2\mu$, $4\mu$, \ldots. The graphs in this paper show these multiples: 1, 2, 4, \ldots. A value of 0 means an idle (partitioned) distributed evolution.

All the experiments compare synchronous/asynchronous versions of parallel dGAs (labeled with $s$ and $a$, respectively, in the graphs). The experiments use equivalent parameterizations for all the algorithms solving the same problem (512 individuals and the same probabilities). In addition, standard operators are used: proportional selection, double point crossover, and bit flip mutation for the first two problems. For the two instances of the third problem we use proportional plus random parent selection (to enhance diversity and hyper-plane recombination), uniform crossover [26], and real additive mutation for the float genes. See the explanation of the characteristics of these three problem domains in Section 3.

3. The test suite

The problems we use to study the sequential and parallel models of GAs are described in this section.

All the problems maximize a fitness function. We use in this work the generalized sphere problem, the subset sum problem, and a problem consisting in training a neural network (NN) by using a genetic algorithm.

The sphere problem will help us in understanding the PGA models by defining a baseline for comparisons. The subset sum problem is NP-complete and difficult for the algorithms. Finally, training an NN needs an efficient exploration of the search space, final local tuning of the parameters in the strings, and show high epistasis (weight correlation), and multimodality.

3.1. The generalized sphere problem (SPH16–32)

The first problem is an instance of the well-known sphere function (Eq. (2)). We use 16 variables each one encoded in 32 bits (string length $l = 512$ bits), and call it SPH16–32. This problem is not especially difficult for a GA, but it helps us in developing comparisons.

$$f(x) = \sum_{i=1}^{n} x_i^2, \quad x_i \in [-5.12, 5.12].$$ (2)

3.2. The subset sum problem (SSS128)

The second optimization task we address is the subset sum problem [17]. It consists in finding a subset of values $V \subseteq W$ from a set of integers $W = \{\omega_1, \omega_2, \ldots, \omega_n\}$, such that the sum of this subset approaches, without exceeding, a constant $C$.

We use an instance of $n = 128$ integers generated as explained in [17] and call it the SSS128 problem. The range for the integers $\omega_i$ is [0,10^4] (and not [0,10^3], as other authors use) to increase the difficulty.

In order to formulate it as a maximization problem we first compute the sum $P(x) = \sum_{i=1}^{n} x_i \cdot \omega_i$ for a tentative solution $x$, and then use the fitness function:

$$f(x) = a \cdot P(x) + (1 - a) \max[C - 0.1P(x), 0],$$ (3)

where $a=1$ when $x$ is admissible, i.e., when $C - P(x) \geq 0$, and $a=0$ otherwise. Solutions exceeding the constant $C$ are penalized.

3.3. Training multilayer perceptrons (“traffic” and “parity4”)

We finally use PGAs for learning the set of weights [1,21,31,32] that make a multilayer perceptron (a
subclass of neural networks [22]) to classify a set of patterns correctly (supervised learning). We test the algorithms on two NN problems. The first NN we consider computes the parity of four binary inputs and uses three layers of 4–4–1 binary neurons encoded as real values in a string for the GA (Fig. 6). Every string thus contains 1 = 4·4 + 4·1 + 4 + 1 = 25 variables (weights plus biases).

The fitness function computes the error between the expected and the actual output values over the 16 possible training patterns. Then, this error is subtracted from the maximum possible error to have a maximization function.

The second multilayer perceptron is trained to predict the level of urban traffic in a road (one single output) from three inputs: the number of vehicles per hour, the average height of the buildings at both sides of the road, and the road width [5].

All these values are floating point numbers in the range [0,1]. The perceptron has a structure made of three layers of 3–30–1 sigmoid neurons. This yields strings of 1 = 151 real values. For this problem we also present results when encoding every weight with 8 bits (strings of 1 = 1208 bits length) to experiment with very long strings. Every function evaluation needs to compute the error along a set of 41 patterns (time consuming).

4. Real time analysis of the impact of the synchronism

Let us begin first by taking a look to the differences and similarities between synchronous and asynchronous parallel distributed GAs. Both synchronous and asynchronous parallel dGAs perform both the same algorithm. However, synchronous islands wait for every incoming string they must accept, while asynchronous ones do not: they insert one migrant whenever it arrives. In general, if all the machines are of the same type (our case) no differences should appear in the evaluation effort or algorithmic behavior (we check this hypothesis in this work). This is not necessarily true if processors of different types are being used.

Fig. 6. Encoding the NN weights into a string of values. \((x - y)\) represents the weight for the link between the output of the neuron \(x\) and the input of neuron \(y\). Bias are not showed although they are used.

Fig. 7. Time in solving SPH16–32 with (a) synchronous and (b) asynchronous versions.
In Fig. 7 we show the expected execution time (50 independent runs) for different number of processors, island evolutions, and migration frequencies to solve SPH16–32. The asynchronous models (Fig. 7b) consistently lessen the search time with respect to their equivalent synchronous algorithms (Fig. 7a) for any of the tested migration frequencies (1,2,4), or island evolutions (steady-state or cellular). For more than 4 processors differences are negligible, since any version quickly found a solution.

With respect to the migration frequency notice that a tight coupling (value of 1) is the better in solving this problem. The reason can be found in the absence of difficulties in the search space of the SPH16–32 function, amenable to a panmictic GA. In fact, a distributed GA with high coupling resembles the search performed by such a panmictic GA (but faster).

The effects of the synchronization on the subset sum clearly demonstrate the drawbacks of a tight coupling on the search time when problems have a high complexity (Fig. 8b). The contrary holds for the above SPH16–32 since a panmictic-like evolution is enough to solve it. In Fig. 8a we plot the numeric results for many different isolation times, island evolution and
1, 2 or 8 processors (100 independent runs for every set of these three parameters). The synchronous and asynchronous models present a similar numeric performance, thus confirming our initial hypothesis on the similarities when run in a homogeneous parallel hardware platform.

All of them found a solution with a similar effort, except when migration frequency is 1, since in many cases the parallel distributed execution was unable to locate a solution (see the success percentage on the top of its associated bars). Values of 16 and 32 provided always a better numeric efficiency and 100% success rate in all the experiments.

Also, a better resistance of dcGA to bad migration frequencies is detected, i.e. for migration frequency 1. For example, for 2 processors, dssGA had a 57% of success while dcGA had 100%, and needed a smaller number of evaluations. However, when using eight islands, dcGA has very small grids in every sub-algorithm, thus losing its search properties (success rate drops from 100% to a 39–47%). On the contrary, dssGA performs better when using eight islands (74%) than when using two (54%) in terms of its success rate, due to the higher success probabilities encountered in the larger independent sampling provided by every one of its smaller panmictic islands.

Although the number of evaluations is very similar for synchronous and asynchronous versions, in all the cases (except for migration frequency 1) the time to get a solution is consistently reduced by the parallel distributed versions if compared with the non-distributed algorithm. As an exception, we can see a scenario for SSS128 in which a parallel dGA is worse than its sequential counterpart: excessive interaction among the islands (see Fig. 8 again).

Many of the previously mentioned conclusions also hold for the experiments when training the “traffic” NN (see the results in Fig. 9). Only a parallel version (8 processors) is showed since no sequential version with ssGA nor with cGA of equivalent parameters was able to solve it even after 15 h with the same final error. For this network the migration interval has a minor impact on the number of visited points (Fig. 9a).

The main difficulty here is in making the final tuning of the floating-point real-values, and in the useless effort wasted in computing bad strings created after two good parents due to the loss of functionality in the resulting children [1]. After the first half part of their execution (approximately) the algorithms have reached the neighborhood of a viable solution, and then they spent a lot of time in refining the gene values. This smoothens the overall differences in the effort for locating the region of one solution, thus providing very similar total runtimes.

The search time of dssGA (Fig. 9b) is very good due to its high exploitation. For dcGA, the loosely connected and the partitioned variants are the better ones. This is so since the cGA islands are capable of a timely sustained exploitation without important diversity losses. However, dssGA seems to be more appropriate for training neural networks since the high exploitation and parallelism favor quick execution, and, at the same time, the relative isolation provided by this model preserves exploration and diversity.
Notice (only results for the “traffic” NN in this paper) that the search time is also influenced by the encoding (Fig. 9b). A binary representation is slightly slower than a real encoding for dssGA, as suggested with other models and problems in [18] due to the larger strings the algorithm must manage. However, the number of evaluations is somewhat smaller since binary strings induce a pseudo-Boolean search in a discrete space.

To summarize, it can be seen that the number of evaluations needed for SPH16–32, SSS128, and for the “traffic” NN (part (a) of Figs. 8 and 9) are almost not affected by the synchronization. On the contrary, the search time is considerably lessened by all the asynchronous algorithms.

5. Speedup

Before discussing the results on the speedup of the analyzed algorithms, we need to make some previous considerations. As many works have established [7,14], sequential and parallel GAs must be compared by running them until a solution of the same quality has been found, and not until the same number of steps has been completed. In other algorithms the speedup (Eq. (4)) is upper bounded by the number of processors \( n_{\text{proc}} \), but for a PGA it might not, since it reduces both the number of necessary steps and the expected execution time \( T_{n_{\text{proc}}} \) in relation to the sequential one \( T_1 \). This means that super-linear speedups are possible (see some examples in [4,24]).

\[
s(n_{\text{proc}}) = \frac{T_1}{T_{n_{\text{proc}}}}. \tag{4}
\]

In fact, super-linear speedups are possible for many other stochastic search methods since there exists a non-zero probability of needing \( T \) time units, for any \( T > 0 \), in reaching one solution. As stated in [24], there exists an exponential relationship between the speedup and the number of processors. See this relationship in Eq. (5), where \( s \) is the acceleration factor (super-linear speedup when \( s > 1 \)).

\[
s(n_{\text{proc}}) = n_{\text{proc}} s_{n_{\text{proc}}-1}. \tag{5}
\]

In Fig. 10 we show the speedup of dssGA and dcGA in solving SPH16–32 (50 independent runs). Two conclusions can be drawn. On the one hand, the speedup is clearly super-linear, since e.g. an eight island dssGA (dcGA) is better than the sequential panmictic ssGA (cGA) in a factor larger than 8. This is due to the higher diversity and parallel exploration from many different zones of this GA-easy problem.

On the other hand, since in this case the problem is not too difficult, highly coupled islands (e.g. migration frequency 1) are better — like using a single population but faster. This does not hold for other problems. In fact, a large isolation time is recommended in
many works \[4,6\], as also shown in our results of the previous section. Besides that, the asynchronous versions show definitely better speedup values than the synchronous ones for dssGA (Fig. 10a), and slightly better for dcGA (Fig. 10b) for equivalent migration frequencies \((a1 > s1, a2 > s2, \ldots)\).

The speedup for the subset sum problem (Fig. 11) confirms the benefits of a sparse migration (100 independent runs): the speedup is better for larger isolation times (16 and 32). Super-linear speedups for dssGA, and almost linear speedup for dcGAs with migration frequency of 32 have been obtained. More reasonable speedup values are in general observed due to the difficulty of this problem. As expected, a tight coupling sometimes prevents the occurrence of linear speedups.

We finally plot in Fig. 12 the results when training the “parity4” neural network. Again, the asynchronous models had a larger speedup and lower absolute execution times. Also, a long isolated evolution is the best parameterization for any of the tested models.

At this point, after discussing so many configurations, algorithms, migration frequencies, and synchronous/asynchronous versions, what is really clear is the superiority of the tested distributed models over the sequential ones for a set of different and non-trivial problems. Also, the advantages of using the asynchronous models can be definitely stated, thus requiring some justification for further works dealing with synchronous (and not with asynchronous) communications. We also have shown that in many cases, it is better (faster, robust against wrong parameter set-

6. Further understanding of the basic models

In this section our aim is to get deeper in the understanding of the working principles of the basic selection models guiding the search of the previously analyzed algorithms. With this goal in mind we are first undertaking a general study on the selection pressure either in steady-state and cellular GAs, and also
6.1. Basic selection pressure

In order to further understand the basic expected performance of these algorithms we will study their selection pressure, since selection is one of the primary operations controlling the exploration/exploitation ratio. We extend the work in [27] comparing genGAs vs. ssGAs to consider also cGAs, and all their distributed counterparts. Besides that, the reader can find more details on the selection pressure of cGAs in [23].

In Fig. 13 we plot the normalized expected best class \( P_{b,t} \) (best string) growing rate using only proportional selection on a 1024 string population with uniform random fitness values in the range \([0, \ldots, 255]\). The quick convergence of the population to the best string for ssGA and the moderate growth of genGA is clear. The cGA can be shifted from high to low selection pressure simply by changing the shape of the grid (making it thinner: from \(32 \times 32\) to \(4 \times 256\)). This is an easier alternative than changing the kind of neighborhood [23] for obtaining a new selection pressure in the grid. Of course, all these three GAs use a given selection operator that can also be tuned to provide a better exploitation/exploration balance.

Since we focus on PGAs, Fig. 14 plots the relative growth of the best string throughout many generations for the three distributed algorithms: dssGA, dgenGA and dcGA. The results for dssGA with eight islands (Fig. 14a) show a slight decrease in the curve slope with respect to its panmictic version. This would be beneficial since dssGA shows a better diversity and at the same time a lower runtime. In dgenGA and in the rest of distributed models, a higher selection pressure can be got by increasing the migration frequency (we plot frequencies of 0, 1, and 32). See this effect, e.g. on the results for two grids (squared \(11 \times 11 = 121 \sim 128\) and rectangular \(4 \times 32\)) in an eight island dcGA in Fig. 14b.

For the same kind of grid the selection pressure is larger as the migration is more frequent (i.e. 1). Partitioned algorithms (frequency 0) show the slowest growing curves for dcGA and also for dssGA and dgenGA.

The selection pressure is more flexible for dcGA since every basic cGA island allows for a flexible tuning. In fact, increasing the selection pressure is

![Fig. 13. Proportion of the best class versus generations in steady-state, generational and cellular evolution modes. The population size is 1024 individuals.](image)

![Fig. 14. Proportion of the best class versus generations. We plot the distributed versions of (a) the panmictic models shown in Fig. 13: steady-state, generational, and (b) for a structured cellular GA with two different grid shapes. Global population size is also 1024 individuals.](image)
possible in two ways: by making migration more frequently or/and by using squared grids. A third way is to change the selection operator, but this can be made on any of these three GAs.

In conclusion, highly coupled islands provide a larger selection pressure (dssGA has been almost not affected since by default it is high) and allows a centralized-like evolution in a lower runtime with respect to a serial execution. In dcGA there exists an additional parameter to control the selection pressure: the shape (or neighborhood) of the grid population.

### 6.2. Diversity

In this section we are going to show that the synchronism does not alter the fundamental numerical behavior of the distributed PGAs when using a homogeneous pool of processors. In previous sections we have seen how the numerical effort is almost independent of the synchronism with some small drifts when the number of processors is medium–high.

Also, we are going to see that, from a diversity point of view, the synchronism does not numerically affect the algorithm. To achieve this conclusion we will track the genotypic information present in the population as evolution proceeds.

In order to measure the diversity level we use the mean bit entropy of the population as a natural way of monitoring the information quantity in every GA step. The mean entropy $H$ of a population $P(t)$ at time $t$ is defined in Eq. (6), in which $P_i^0$ is the proportion of 0’s at string position $i$: $1, \ldots, l$, and $P_i^1$ is the proportion of 1’s:

$$H[P(t)] = -\frac{1}{l} \sum_{i=1}^{l} \left( P_i^0 \times \log_2 P_i^0 + P_i^1 \times \log_2 P_i^1 \right).$$  \hspace{1cm} (6)

For the SSS128 problem, when we distribute a 512 string population over $n_{\text{proc}} = 8$ processors with a synchronous migration step every 32 generations (Fig. 15a), we can see how the diversity is effectively lessened for dgenGA and dssGA, but not for dcGA. This explains the good results in terms of success rate of dcGA.

In general, we have got these same results for many other problems because of the high (medium) exploitation characteristics of a steady-state (generational) island-based dGA. Whenever a considerable population size is used in a cellular GA (64, 128 individuals and more) its properties of very high diversity, isolation by distance and exploration are preserved; on the contrary, for very small populations, structuring the population does not show evident advantages.

If compared with the asynchronous versions of equal migration frequency (Fig. 15b), we will see that no differences exist in terms of diversity between these two implementation alternatives. Therefore, we can confirm the expected similitude between the two versions (synchronous/asynchronous) of a given distributed homogeneous GA from an algorithmic
point of view (effort and diversity), given that similar hardware is being used for both of them.

7. Concluding remarks

In this paper we have stated the importance of distributed parallel genetic algorithms as well as the advantages of a common analysis of different models. The canonical revision of their behavior is very interesting for novel-GA and non-GA researchers willing to apply these techniques in their own domains.

Parallel distributed GAs almost always outperformed sequential GAs. We have distributed sequential steady-state, generational, and cellular GAs with the goal of extending the existing studies to cover panmictic and structured-population algorithms. The studied parallel distributed models are targeted to a wide spectrum of research institutions because of their easy implementation on MIMD platforms.

In all the experiments asynchronous algorithms outperformed their equivalent synchronous counterparts in real time. This confirms other existing results with different PGAs and problems, clearly stating the advantages of the asynchronous communications [14]. Our results have taken into account the kind of performance analysis proposed in [14], where the expected execution time to get a solution of the same quality is used to compare the algorithms. Besides that, the number of function evaluations is used as a reasonable metric of the numerical performance of PGAs. The stochastic behavior of these algorithms forced us to use mean or expected values for the runtime and number of evaluations, since a single trial of any of them is not particularly meaningful. We have performed 50 and 100 independent runs on every point in every graph to offer representative values.

After having fulfilled all those requirements (as requested in the conclusions of [14]), we can confirm that parallel distributed genetic algorithms are sometimes capable of providing super-linear speedup, as shown in previous works as [4,24] (which did not meet all these requirements in their studies).

As a future work it would be interesting to study the influence of the migration policy, as well as the importance of using different parameters and operators in every island (heterogeneity). Besides that, the outlined invariance of the mean entropy of the population for both, the synchronous and the asynchronous versions, is a research line we are already exploring. Preliminary results confirm that these two kinds of implementations perform the same search when similar processors are used.

In addition, the importance of the selected island model for a given fitness landscape is an open research line. In particular, for cGA islands, the grid shape has a potentially important role in guiding the selective pressure. In the cGA islands of this paper the height was always set to four strings (narrow grids), and the width was set to the corresponding value, depending of the sub-population size needed to ensure an equivalent global population. Since the size and shape of the grids in the islands of a dcGA can be tuned, an open question is how this could help in allowing still more efficient and customizable algorithms.

References


Enrique Alba received his Ph.D. in Computer Science in 1999 from the University of Málaga (Spain). Since 1991 he has worked in evolutionary algorithms in the department of “Lenguajes y Ciencias de la Computación” of the University of Málaga. At present he works as an Associate Professor in this Department, and his current interests are in evolutionary algorithms, parallel models and their applications. He has received national and international awards for his research activities and belongs to the EvoNet European network of excellence, and also he participates in the program committees of the main conferences on evolutionary algorithms.

José M. Troya received his Ph.D. in Physics in 1980 from the University Complutense of Madrid (Spain). From 1980 to 1988 he was an Associate Professor in that university, and since then he has been a Professor in the Department of “Lenguajes y Ciencias de la Computación” at the University of Málaga. His research interests are in parallel algorithms and software architectures, and in the development of languages and tools for programming and modeling distributed systems. He has directed many national and international research projects and numerous Ph.D. students in the mentioned fields.