Statistics and Computing **12**: 91–114, 2002

Improving flexibility and efficiency by adding parallelism to genetic algorithms

ENRIQUE ALBA and JOSÉ M. TROYA

Dpto. de Lenguajes y Ciencias de la Computación, Univ. de Málaga Campus de Teatinos (3.2.12), 29071—Málaga, Spain
eat@lcc.uma.es, troya@lcc.uma.es

Received November 1999 and accepted October 2000

In this paper we develop a study on several types of parallel genetic algorithms (PGAs). Our motivation is to bring some uniformity to the proposal, comparison, and knowledge exchange among the traditionally opposite kinds of serial and parallel GAs. We comparatively analyze the properties of steady-state, generational, and cellular genetic algorithms. Afterwards, this study is extended to consider a distributed model consisting in a ring of GA islands. The analyzed features are the time complexity, selection pressure, schema processing rates, efficacy in finding an optimum, efficiency, speedup, and resistance to scalability. Besides that, we briefly discuss how the migration policy affects the search. Also, some of the search properties of cellular GAs are investigated. The selected benchmark is a representative subset of problems containing real world difficulties. We often conclude that parallel GAs are numerically better and faster than equivalent sequential GAs. Our aim is to shed some light on the advantages and drawbacks of various sequential and parallel GAs to help researchers using them in the very diverse application fields of the evolutionary computation.

Keywords: parallel genetic algorithms, distributed GAs, cellular GAs, PGAs theory, PGA parameters influence, speedup, efficiency, scalability

1. Introduction

Parallel and sequential genetic algorithms (P-GAs) are modern techniques for searching complex problem spaces for an optimum (Bäck, Fogel and Michalewicz 1997, Reeves 1993). In Fig. 1 we locate genetic algorithms in relation to other search techniques. Genetic algorithms are randomized optimization methods that need minimal information on the problem to guide the search. They use a population of multiple structures, each one encoding a tentative solution, to perform a search from many zones of the problem space at the same time. The application of simple stochastic operators moves this population towards better sub-optimal solutions in an iterative manner, until a stopping criterion is fulfilled (Holland 1975, Goldberg 1989a, Bäck, Hammel and Schwefel 1997).

Until recent years, sequential GAs have received the greatest attention from the research community. However, here we focus on parallel GAs since they have many interesting unique features that deserve in-depth analysis (Gordon and Whitley 1993, Baluja 1993). These characteristics include (Hart *et al*. 1997):

(1) the reduction of the time to locate a solution (faster algorithms),
(2) the reduction of the number of function evaluations (cost of the search),
(3) the possibility of having larger populations thanks to the parallel platforms used for running the algorithms, and
(4) the improved quality of the solutions worked out.

In addition, the parallel GA versions are less prone to premature convergence to sub-optimal solutions (Alba and Troya 1999), thus improving the search. PGAs deserve a special attention since they are not just "faster versions" of sequential GAs. Instead, they represent a new algorithmic class providing a different (often better) search mechanism.

We analyze GAs and not EAs (evolutionary algorithms) since GAs are quite popular and receive many applications, and also because many results with GAs are extensible to other kinds of EAs. See (Bäck 1996) for learning how similar the different EA families are. Let us see the well-accepted sub-classes of EAs in Fig. 1, namely genetic algorithms (GA), evolutionary

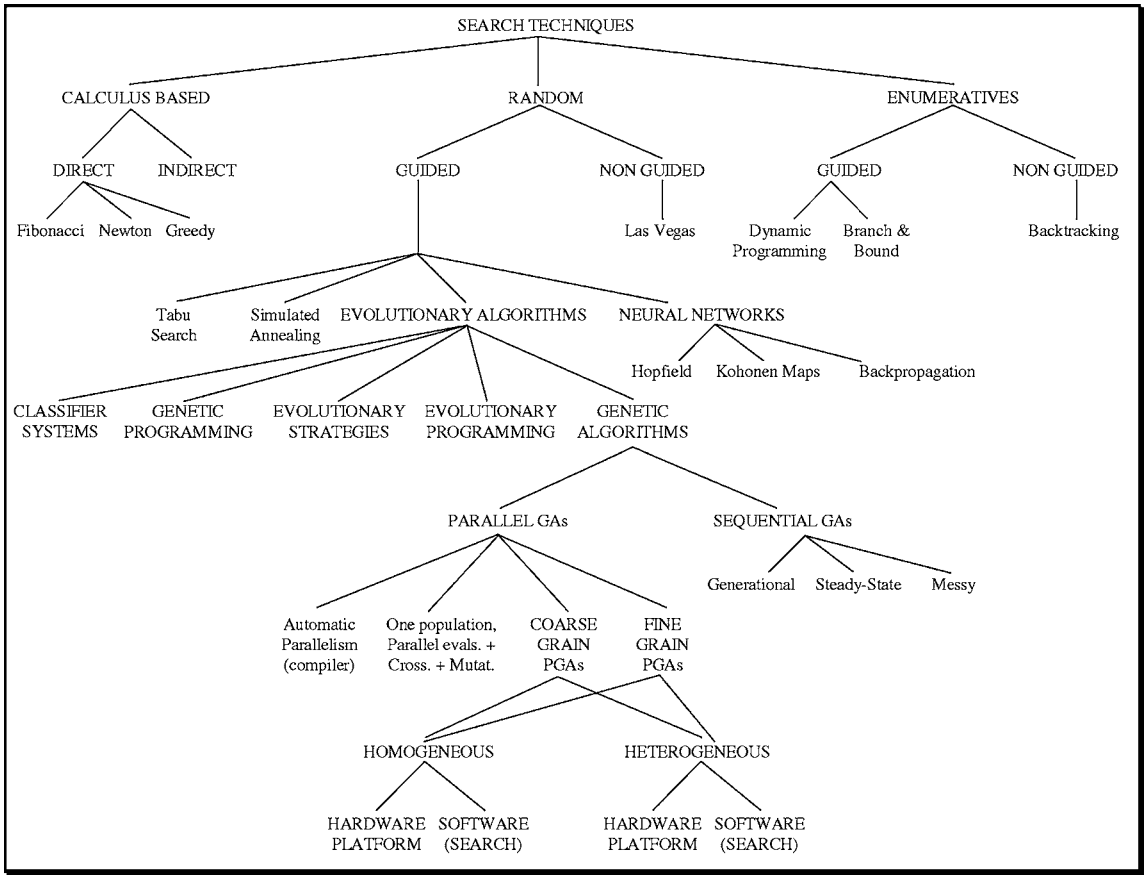


Fig. 1. *Taxonomy of search techniques*

programming (EP), evolution strategies (ES), genetic programming (GP), and classifier systems (CS). In Bäck, Fogel and Michalewicz 1997) the reader can find a great compendium of the state of the art in evolutionary computing (EC). We also have included four types of parallel GAs, a distinction depending of the homogeneity at execution/search levels, and some other popular techniques such as tabu search, simulated annealing and neural networks.

The class of problems solved with GAs is very general, since the optimized function can have an arbitrary complexity e.g., see our results on training neural networks (Alba, Aldana and Troya 1993), load balancing (Alba, Aldana and Troya 1995), designing fuzzy logic controllers (Alba, Cotta and Troya 1996), and validating communication protocols (Alba and Troya 1996). Hence, providing *useful* conclusions on the relative performance of these algorithms by looking to existing works is difficult for many reasons. Firstly, different parameters, hill-climbing, hybridized or specialized operations are considered (Gordon and Whitley 1993, Voigt, Santibáñez-Koref and Born 1992, Mühlenbein, Schomisch and Born 1991). Secondly, in some works only one single problem is studied. Thirdly, existing papers on PGAs usually cover a reduced range of models (Tanese 1989) or analyze only one feature (Munetomo, Takai and Sato 1993). On the contrary, in this paper we present several canonical models, and undertake a broad but detailed presentation. This requires including both theoretical and experimental results, using many non-toy problems, and analyzing important parameters. The selected benchmark contains a compendium of the actual difficulties found in the optimization of real world problems, namely epistasis, multimodality, deception, and large problem instances. All this is very useful for guiding researchers willing to use PGAs.

The main contributions of this paper are (1) the unified description and study of PGAs, and (2) the comparison among the *canonical* models which usually receive parallel implementations. To achieve these goals we make several considerations on their complexity, selection pressure, and relative efficiency (speedup). Also, we include results on their scalability when solving increasingly larger problems. We also discuss the similarities and differences between parallel and speciation GAs (next section). In summary, we revisit some important points in PGAs and offer new results to improve their understanding.

The paper is organized as follows. In Section 2 we present a survey of PGAs to put in context our research and to help non-EA readers. Section 3 analyzes three theoretical issues in PGAs, namely time complexity, selection pressure, and schema processing rates. Section 4 deals with distributed PGAs, while in Section 5 we examine cellular GAs, the underlying model for

fine grain PGAs. Section 6 gives a general PGA performance analysis. The scalability of PGAs is presented in Section 7, and Section 8 summarizes the most important conclusions. Finally, we add an Appendix describing the benchmark and nomenclature to ease the comprehension of the paper.

2. Survey of parallel genetic algorithms

Let us begin by examining the outline of a general PGA (Algorithm 1). As a class of stochastic technique, we can distinguish three major steps in a PGA, namely *initial sampling*, *optimization* and *checking the stopping criterion*. It begins ($t = 0$) by randomly creating a population $P(t = 0)$ of μ structures, each one encoding the p problem variables on some alphabet of symbols. Each structure is usually a vector (string) over $\mathbb{B} = \{0, 1\}$ ($I = \mathbb{B}^{p \cdot l_x}$) or $\mathbb{R}$($I = \mathbb{R}^p$) i.e., most GAs encode each problem variable in l_x bits or in a floating-point number.

An evaluation function Φ is needed each time a new structure is generated in the algorithm. This evaluation is used to associate a real value to the *decoded* version of the structure. This value indicates its quality as a solution to the problem. In textbook GAs decoding means mapping the binary string to a decimal value in some given range.

Afterwards, the GA iteratively applies a set of variation operators to some selected structures from the current population. The goal is to create a new pool of λ tentative solutions and evaluate them to yield $P(t + 1)$ from $P(t)$. This generates a sequence of populations $P(0)$, $P(1)$, $P(2)$, ... with increasingly fitter structures. The stopping criterion ι is to fulfill some condition like reaching a given number of function evaluations, finding an optimum (if known), or detecting stagnation in the algorithm after a given number of generations. See Michalewicz (1992) and Bäck, Hammel and Schwefel (1997) for more in depth background.

The *selection* $s_{\Theta s}$ uses the relationship among the fitness values of the structures to create a mating pool. Some parameters Θ_s might be required depending on the selection type (Baker 1987). Typical variation operators are *crossover* ($\otimes$ binary) and *mutation* (m, unary). Crossover recombines two parents by exchanging string slices to yield two new offspring. Mutation randomly alters the contents of these new offspring. The behavior of these two stochastic operators is governed by a set of parameters such as a probability (rate) of application: $\Theta_c = \{p_c\}$ and $\Theta_m = \{p_m\}$, usually high for crossover ($\otimes$), and low for mutation (m).

Finally, each iteration of the algorithm ends by selecting μ (out of λ) new individuals that will be used as the new population. For this purpose, the temporal pool $P'''(t)$ plus a set Q are considered. Q might be empty ($Q = \emptyset$) or else contain the old population $Q = P(t)$. This step applies a *replacement policy* r. It uses the recently created temporary pool (and optionally the old pool) to compute the new population of μ individuals. It is usual for the best structure in one population to deterministically survive in the next generation (*elitist* evolution).

Many sequential GAs exist, but two of them are especially popular. The first one is the *generational GA* –genGA– where a whole new population ($\lambda = \mu$) is created to replace the old one ($Q = \emptyset$). The second one is the *steady-state GA* –ssGA– in which only a few structures are generated in each iteration ($\lambda = 1$ or 2) and they are inserted in the current population ($Q = P(t)$) (Syswerda 1991). They both are *panmictic* algorithms having an unstructured population.

Algorithm 1: Parallel genetic algorithm

$t := 0$;
initialize: $P(0) := \{\vec{a}_1(0), \ldots, \vec{a}_\mu(0)\} \in I^\mu$;
evaluate: $P(0) : \{\Phi(\vec{a}_1(0)), \ldots, \Phi(\vec{a}_\mu(0))\}$;
while $\iota(P(t)) \neq$ **true do** //*Reproductive plan*
  select: $P'(t) := s_{\Theta_s}(P(t))$;
  recombine: $P''(t) := \otimes_{\Theta_c}(P'(t))$;
  mutate: $P'''(t) := m_{\Theta_m}(P''(t))$;
  evaluate: $P'''(t) : \{\Phi(\vec{a}_1'''(t)), \ldots, \Phi(\vec{a}_\lambda'''(t))\}$;
  replace: $P(t + 1) := r_{\Theta_r}(P'''(t) \cup Q)$;
  <***communication***>
  $t := t + 1$;
end while

Many works such as Levine (1997), Syswerda (1991), and Whitley and Starkweather (1990) show that the one-at-a-time reproduction of ssGA is very useful. However, pure genGAs are still widely accepted, since some of their drawbacks can be solved by using improved operators, that, incidentally, could also improve the canonical ssGA as well. Anyway, further studies are needed to highlight their relative advantages.

In a parallel GA there exist many elementary GAs performing the reproductive plan on their own sub-populations $P^i(t)$. Each sub-algorithm includes an additional phase of *communication* with a set of neighboring sub-algorithms. At first glance, the precedent pseudocode could lead the reader to think that there exist other models of PGAs which differ from the one shown. However, we must realize that each sub-population $P^i(t)$ can contain as much as several hundreds of strings or as few as only one string. The reproductive plan can thus manage millions or only a few operations before the algorithm is engaged with the communication step. Also, we must note that the communication step is not only intended to exchange strings among neighboring sub-populations. On the contrary, any other information judged interesting for the search can be exchanged e.g., statistics or performing string evaluations in other processors. In addition, the number of elementary GAs is another free parameter. This allows us to say that the above pseudocode is a general description of PGAs.

If a population with tens of strings is contained in every sub-algorithm, the PGA is *coarse grained* and interactions with neighbors are sparsely undertaken (Lin, Punch and Goodman 1994) (Fig. 2(b)). If the population is composed of a single individual then we have a *fine grained* or *cellular* GA –cGA– (Fig. 2(c)), and interactions with neighbors are very frequent in order to get the small pool of structures to apply the reproductive plan on (Spiessens and Manderick 1991). It is usual in this *diffusion* model that every sub-algorithm waits (synchronously)

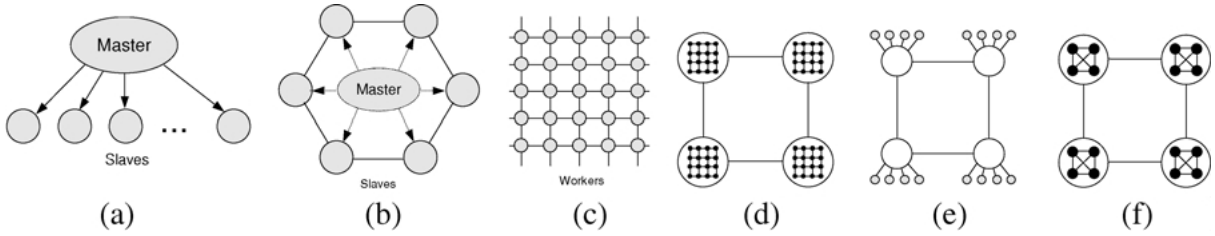


Fig. 2. *Different models of PGA: (a) global parallelization, (b) coarse grain, and (c) fine grain. Many hybrids have been defined by combining PGAs at two levels: (d) coarse and fine grain, (e) coarse grain and global parallelization, and (f) coarse grain plus coarse grain*

for its neighbors, while the coarse grain PGA can either wait or not for incoming strings.

The coarse grain PGA sends and receives strings from neighboring sub-populations, while the fine grain PGA uses an overlapped neighborhood system to provide a smooth diffusion of the strings. Coarse (fine) grain PGAs are easy to parallelize on MIMD (SIMD) computers, respectively (Stender 1993). However, even when run in a monoprocessor, they have some advantages over many sequential panmictic GAs thanks to their structured population.

The research performed on PGAs and on *speciation methods* (Deb and Spears 1997) share some similarities. Speciation GAs are aimed at locating several optima at the same time. To achieve this goal speciating algorithms explicitly maintain different solution species during the optimization process by applying specific techniques. Parallel GAs lead naturally and implicitly to create multiple niches of solution species, just as speciation GAs. However, the latter ones dynamically allocate a different number of individuals and a different fitness value to individuals in every niche. This concentrates effort on more promising peaks, while still maintaining individuals in other areas of the search space. Distributed GAs provide only constant sized niches, and no fitness changes are associated to any standard parallel model. On the other hand, cellular GAs allow speciation e.g., in a grid, but one particular species will eventually take over the whole population if no specific operators are included.

Many sorts of PGAs exist. See again Fig. 2 for an overview of pure and hybrid PGAs. In the taxonomy of Fig. 1 we include four accepted PGA types. The *global parallelization* type evaluates, and possibly crosses and mutates the strings in parallel; selection uses the whole population. Interesting considerations on global parallelization can be found in Cantú-Paz and Goldberg (1997). This model provides lower run time only for very slow objective functions (which is a drawback) and its search mechanism uses a single population.

The type called *automatic parallelization* PGA in Fig. 1 is rarely found since the compiler must "automatically" provide the parallelization of the algorithm. The hybrid models in Fig. 2 combine different PGAs at two levels to enhance the search. Interesting surveys on PGAs can be found in Alba and Troya (1999), Cantú-Paz (1997), and Adamidis (1994).

Previous research has been conducted on these parallel models separately, but much can be gained by studying them under a common viewpoint. They are not opposite sub-classes of algorithms. The traditional distribution of sub-populations with sparse migrations of individuals is considered in this paper as a mechanism to enhance the behavior of any kind of basic GA. This basic GA can be the any of the mentioned GAs (Alba and Troya 2000a).

We provide in Table 1 a quick overview of different PGAs to point out important milestones in parallel computing with GAs. These "implementations" have rarely been studied as "parallel models". Instead, only the implementation itself is usually evaluated.

Some coarse grain algorithms like dGA (Tanese 1989), DGENESIS (Mejía-Olvera and Cantú-Paz 1994), GALOPPS (Goodman 1996), PARAGENESIS (Stender 1993), PGA 2.5 (http://www.aic.nrl.navy.mil/galist/src/pga-2.5.tar.Z) and PGA (Mühlenbein, Schomisch and Born 1991) are relatively close to the general model of migration islands. They often include many own features to improve efficiency. Some other coarse grain models like CoPDEB (Adamidis and Petridis 1996) and GDGA (Herrera and Lozano 2000) have been designed for specific goals such as providing explicit exploration/exploitation by applying different operators on each island.

Some other PGAs execute non-orthodox models of coarse grain evolution. This is the case of GAMAS (Potts, Giddens and Yadav 1994) based on using different alphabets in every island, GENITOR II (Whitley and Starkweather 1990) based on a steady-state reproduction, or iiGA (Lin, Punch and Goodman 1994) that promotes coding and operator heterogeneity.

On the other hand, massively parallel GAs have been strongly associated to the machines on which they run: ASPARAGOS (Gorges-Schleuter 1989) and ECO-GA (Davidor 1991). This is also the case of models of difficult classification (although most of the mentioned ones are of difficult classification!) like PEGAsuS (Ribeiro, Alippi and Treleaven 1993), SP1-GA (Levine 1994) or SGA-Cube (Erickson, Smith and Goldberg 1991). As to the global parallelization PGA, some implementations such as EnGENEer (Robbins 1992) or PGAPack (Levine 1996) are available.

Finally, some efforts to construct general frameworks for PGAs are GAME (Stender 1993), PEGAsuS, and RPL2 (Radcliffe and Surry 1994). They are endowed of "general" programming structures intended to ease the implementation

Table 1. *Details of popular PGAs*

Parallel GA	Kind of parallelism	Topology	Some applications
ASPARAGOS	Fine grain. Applies hill-climbing if no improvement	Ladder	TSP
CoPDEB	Coarse grain. Every sub-pop. applies different operators	Full connected	Func. opt. and ANN's
dGA	Distributed populations. Studies migration rate and freq.	Ring	Function optimization
DGENESIS 1.0	Coarse grain with migrations among sub-populations	Any desired	Function optimization
ECO-GA	Fine grain. One of the first of its class	Grid	Function optimization
EnGENEer	Global parallelization (parallel evaluations)	Master/Slave	Various
GALOPPS 3.1	Coarse grain. A very portable software	Any desired	F. opt. and transport
GAMAS	Coarse grain. Uses 4 species of strings (nodes)	Fixed hierarchy	ANN, func. opt., ...
GAME	Object oriented set of general programming tools	Any desired	TSP, func. opt., ...
GDGA	Coarse grain. Admits explicit exploration/exploitation	Hypercube	Func. opt. (floating p.)
GENITOR II	Coarse grain. Interesting crossover operator	Ring	Func. opt. and ANN's
HSDGA	Hierarchical coarse and fine grain GA. Uses E.S.	Ring, Tree, Star, ...	Function optimization
iiGA	Injection island GA, heterogeneous and asynchronous	Hierarchy of nodes	Function optimization
PARAGENESIS	Coarse grain. Made for the CM-200 (1 ind. $\Leftrightarrow$ 1 CPU)	Multiple	Function optimization
PeGAsuS	Coarse or fine grain. High-level programming on MIMD	Multiple	Teaching and func. opt.
PGA	Sub-populations, migrate the best, local hill-climbing,...	Circular 2-D ladder	Func. opt. and TSP
PGAPack	Global parallelization (parallel evaluations)	Master/Slave	Function optimization
RPL2	Coarse grain. Very flexible to define new GA models	Any desired	Research and Business
SGA-Cube	Coarse grain. Implemented on the nCUBE 2	Hypercube	Function optimization
SP1-GA	128 steady-state islands on an IBM SP1 with 128 nodes	2-D toroidal mesh	Function optimization

of any model of PGA. The user must particularize these general structures to define his/her own algorithm.

All these models and implementations offer different levels of flexibility, ranging from a single PGA to the specification of general PGA models. This list is not complete, of course, but it helps in describing the current "state of the art".

3. Some theoretical aspects of PGAs

In order to offer a more complete comprehension of the possible sources of efficiency and flexibility of PGAs we discuss in this section three main issues relating theoretical aspects. First, we review and derive new values for the algorithmic complexities of panmictic and structured GAs. Structured GAs are the kind of GAs that usually receive a parallel implementation i.e., distributed and cellular GAs. We also include panmictic GAs in the analysis to link our results with these popular GAs. Second, we discuss the selection pressure of these algorithms i.e., the relative growth of the best solution under selection, which represents the exploration/exploitation features of a GA. The third issue we deal with is schema processing rate, a well-known theoretical explanation of the search with GAs.

3.1. *Time complexity of the algorithms*

In this section we derive the basic time complexities of a genGA, ssGA and cGA. Afterwards we discuss the complexity of their distributed versions running a unidirectional ring of islands with sparse migrations of one randomly selected string. This section is brief because we only want to highlight the main points influencing the execution time of these algorithms.

See in Table 2 a summary of their complexities. There, $n = \mu$ is the number of structures in every island, and l is their length. We derive the complexity of one generation in every algorithm i.e., the effort for creating n new individuals from the n old ones. Let us make a separate presentation for every GA class.

In genGAs, a binary tournament has complexity $O(n \cdot n/2)$, because it selects two pairs of random strings and keeps the best one in each pair as a parent. The operations of crossing, mutating, evaluating, and copying a string to the temporary population are $O(2 \cdot l + l + C_{ev} + l)$. The replacement of the old population is $O(n \cdot l)$ and computing statistics is $O(n)$.

In a ssGA, the binary tournament of two parents is $O(n/2 + n/2)$, crossover, mutation and evaluation are $O(2 \cdot l + l + C_{ev})$, ordered insertion is $O(n/2)$, copying the offspring is $O(l)$ and statistics needs only $O(1)$. The basic step in ssGA generates only

Table 2. *A comparison of the time complexity for some algorithms*

Generational	$O(n^2 + n \cdot l)$	$n \cdot \frac{n}{2} + n \cdot (2 \cdot l + l + C_{ev} + l) + n \cdot l + n$
Steady-state	$O(n^2 + n \cdot l)$	$n \cdot [\frac{n}{2} + \frac{n}{2} + 2 \cdot l + l + C_{ev} + \frac{n}{2} + l + 1]$
Cellular	$O(n \cdot l + n \cdot n_{nb})$	$n \cdot (C_{nb} \cdot n_{nb} + n_{nb} + n_{nb} + 2 \cdot l + l + C_{ev} + l) + n \cdot l + n$
Distributed	$O(d \cdot O(< ga >) + d \cdot l)$	$d \cdot O(< ga >) + d \cdot (l + C_{comm} + l) + d \cdot C_{comm} + d$

one child, thus the process has to be repeated n times to have the same computational grain as in the rest of algorithms.

Although we have mentioned binary tournament, fitness proportional selection such as roulette wheel (RW) can be used. In addition, generational GAs can use Stochastic Universal Sampling (SUS). SUS is very efficient for this kind of algorithms (Baker 1987). In fact, SUS can remove the quadratic term from genGAs' complexity to yield $O(n + n \cdot l)$ complexity.

The cellular GA selects a set of neighbors for every one of the n strings. The complexity of *identifying* neighbors is $O(C_{nb} \cdot n_{nb})$ and selecting two parents is $O(n_{nb} + n_{nb})$, depending both on the neighborhood size n_{nb} and a constant C_{nb} for arbitrary neighborhood shapes. The crossover, mutation, evaluation and insertion in a temporary population is $O(2 \cdot l + l + C_{ev} + l)$. The replacement of the old pool is $O(n \cdot l)$ and computing statistics is $O(n)$.

The distributed ring is composed of d islands and every distributed step accounts for the complexity of the basic islands. The migration, transference (system dependent constant C_{comm}) and replacement of the worst string with an immigrant are of complexity $O(l + C_{comm} + l)$. Finally, computing statistics needs $O(d \cdot C_{comm})$ for the requests to the islands and $O(d)$ for making the global statistics for the distributed algorithm. The communication time can be modeled by a power law $T_{comm} = C_{comm} \cdot d^{\gamma}$, that yields a constant value when a ring is used ($\gamma = 0$). See Cantú-Paz and Goldberg (1997) for more details on this last matter.

In summary, genGA and ssGA have the same complexity when using tournament selection. If RW is used then a genGA will have a slight advantage over ssGA since each pair of selections work out two new children in genGA, but only one in a standard ssGA. When using SUS the complexity of genGA is smaller than that of ssGA.

Cellular GAs have a complexity that depends on the technique for managing the neighborhood. Their complexity is quite good if a MIMD machine is used, while it is similar to panmictic GAs in a monoprocessor implementation. In fact, since neighborhoods are very small (5 strings) any selection operator will provide faster executions than panmictic GAs in a monoprocessor. Finally, distributed GAs have a complexity depending on the complexity of its islands, with a clear overhead due to the communication steps needed for migrating strings and computing global statistics. Of course, when cGA/dGA runs on SIMD/MIMD computers their complexity is shared among parallel processors, which greatly reduces the total run time.

3.2. *Selection pressure in centralized and decentralized models*

In order to understand the basic expected performance of these algorithms we derive their theoretical models of proportional selection pressure. Selection is one of the primary operations in a GA because it controls the exploration/exploitation ratio. We summarize the work in Syswerda (1991) comparing genGAs vs. ssGAs and extend it to consider also cGAs by using the formalization in Sarma and De Jong (1996). Our contribution is the unified description of both panmictic and structured models, and the extension for distributed algorithms. Other statistical explanations successfully predict the *takeover regime* (Rogers and Prügel-Bennett 1999) i.e., the expected time one solution needs to occupy the full population under selection.

In Table 3 we show the expected number of instances (copies) allocated to a given string of fitness f in the next step $n_{t+1}(f)$. In a genGA, every string gets an increment in its expected number of copies proportional to the ratio between its fitness and the average fitness of the population. As regards ssGA, when random replacement is used, the average behavior after creating $n = \mu$ strings is similar to a that of genGA. On the other hand, if we replace the least-fit string in the ssGA then the number of copies in the next step grows fitness-proportionally and decreases only for the least-fit string.

The model for the cGA (Sarma and De Jong 1996) uses a parameter a that controls the selection pressure depending on the neighborhood, grid, and selection operator; $n^*(f)$ is the expected proportion of strings having fitness f at convergence. The *proportion of the best string* is $P_{b,t} = n_t(f_{\max})$, and at convergence $n^*(f_{\max}) = 1$. The parameter a is a function of the population structure and of the kind of selection being applied.

Finally, for the distributed model, the number of instances is highly related to the sum of expected instances of each GA in the ring, only perturbed by migrations. When migration occurs, each string increases its number of instances uniformly due to the random selection of migrants in each island. The reduction only affects to the least-fit string, since the immigrant always replace it in the target island (see Section 4.1 for more details on the migration policy).

The genGA and ssGA(random) show the same selection pressure—we call it SP1. The ssGA(least_fit) has a faster convergence, both in theory–SP2–and practice (as we will see). The cGA–SP3–is easily tunable by changing the parameter a. We will notice these theoretical differences in practice when solving the problems in the forthcoming sections.

Table 3. *Models of selection pressure for every algorithm*

Generational	$n_{t+1}(f) = n_t(f) \cdot \frac{f}{\bar{f}}$
Steady-state (random)	$n_{t+1}(f) = n_t(f) + n_t(f) \cdot (\frac{f}{\sum_{i=1}^{n} f_i} - \frac{1}{n})$
Steady-state (least-fit)	$n_{t+1}(f) = n_t(f) + n_t(f) \cdot (\frac{f}{\sum_{i=1}^{n} f_i}) - (1 - \lceil \frac{f - f_{\min}}{f} \rceil)$
Cellular	$n_{t+1}(f) = \frac{n^*(f)}{1 + (\frac{n^*(f)}{n_0(f)} - 1) \cdot e^{-at}}$
Distributed	$n_{t+1}(f) = \sum_{i=1}^{d} n^i_{t+1}(f) \mid n^i_{t+1}(f) = n^i_t(f) + n^i_t(f) \cdot (\frac{1}{n}) - (1 - \lceil \frac{f - f_{\min}}{f} \rceil)$

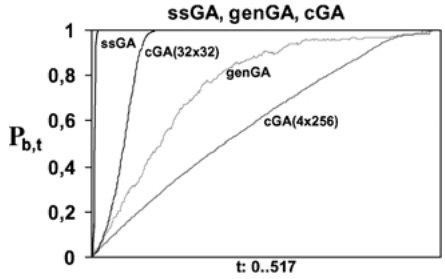


Fig. 3. *Proportion of the best class in steady-state (least-fit), generational, and cellular evolution modes (population size is 1024 individuals)*

The advantage of a cGA is that we can tune the selection pressure by changing the grid shape, keeping constant the rest of parameters (including the neighborhood). See Fig. 3 to confirm this fact: the cGA with 32×32 strings provides a larger selection pressure and a faster convergence than with 4×256. This represents an additional degree of flexibility in that the user can shift the selection pressure from high to low easily at will (Alba and Troya 2000b).

In Fig. 3 it is clear that there exists a quick convergence to the best string of the ssGA(least-fit), and the moderate growth in the genGA. The cGA can be shifted from high to low selection pressure simply by changing the shape of the grid from 32×32 to 4×256. We will revisit this subject later in Section 5.

Since our focus is set on PGAs, we also plot in Fig. 4 the relative growth of the best string until convergence for three distributed algorithms with 8 islands: dssGA, dgenGA and dcGA. In all the tests one random string is being exchanged (least fit replacement). The results for dssGA (left) show a slight decrease in the curve slope with respect ssGA. This should provide good results due to the combined effect of a better diversity and a lower run time. In dgenGA and in the rest of distributed models a higher selection pressure can be obtained by making migrations more frequently (**f** label in Fig. 4; see also Section 4.1). A 0 means separate (idle) evolution, 1 means tight coupling, and 32 represents long isolation.

The selection pressure is much more flexible for dcGA since the elementary cGA islands allow for a flexible tuning (Fig. 4, right). In fact, increasing the selection pressure in a dcGA is possible in two ways: by making more frequent migrations or/and by using squared grids. A third way is to change the selection operator, but this can be made in any sort of GA.

In conclusion, highly coupled islands provide a larger selection pressure and allow a centralized-like evolution in a lower run time with respect panmictic algorithms. In dcGA there exists an additional parameter to control the selection pressure: the shape (or neighborhood) of the grid population.

The selection pressure is only one of the features of a GA. When variation operators are included, the convergence velocity is usually modified. However, selection pressure is a good indicator of the expected behavior of the algorithm. We have just offered a comparative overview on this matter for many models of panmictic and structured GAs. The goal of the following section will be to study some fundamental search properties of these algorithms, and also to confirm the selection pressure just analyzed.

3.3. *Schema processing rates in panmictic and structured GAs*

Our aim in this section is twofold. First, we want to introduce the reader into a popular theoretical explanation of the working principles of genetic algorithms: *the schema theorem*. Second, we want to check the relative selection pressure of both panmictic and structured GAs when analyzing these *schemata*.

The *schema theorem* is one plausible explanation for the behavior of genetic algorithms (Holland 1975, Goldberg 1989a). It is based on the concept of a *schema*. A *schema* is a string over the alphabet $V = \{0, 1, *\}$ having defined (0 or 1) and undefined ($*$) positions. The GA implicitly processes $O(n^3)$ schemata (hyperplanes) while it is really working with n strings (Holland 1975). Two important features of a schema are its *order* and its *defining length*. The order of a schema is the number of defined (non *) positions, while its defining length is the difference between the location of its two outermost defined positions.

The mathematical form of the schema theorem (equation (1)) provides a lower bound for the expected number of instances of a schema H in next generation $m(H, t + 1)$. This estimate is based on the importance of its fitness $\Phi(H)$ and on its survival probability under crossover and mutation. The parameter p_c and p_m are the probabilities of crossover and mutation, respectively. $\delta(H)$ is the defining length, and $o(H)$ is the order of the schema. The algorithm is supposed to implicitly recombine low order

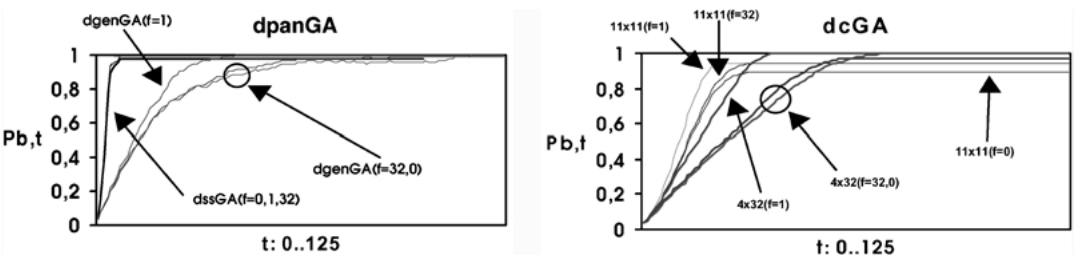


Fig. 4. *Proportion of the best class in distributed versions of ssGA/genGA (left) and cGA (right). Global population size is 1024 individuals*

Table 4. *Parameters, algorithms and the problem being studied*

Parameters	Values	Algorithms	Strings and solution schema
Problem size	3	xxGA(1,80,_,_,ss,ifb)	Solution1:
String length	36	xxGA(1,80,_,_,sus,e)	110000000000 110000000000 000000000011
Pop size	80	xxGA(1,2 × 40,_,_,cel,ifb)	Solution2:
p_c	1.0	xxGA(4,20,r,640,ss,ifb)	110000000000 000000000011 000000000011
p_m	0.01	xxGA(4,20,r,32sus,e)	Schema: [order = 32, defining length = 35]
Max. allowed evals.	10400	xxGA(4,2 × 10,r,32,cel,ifb)	110000000000 **00000000** 000000000011

schemata of short defining length to work out better solutions. See Goldberg (1989b) for extensions on these concepts in parallel GAs.

$$m(H, t+1) \geq m(H, t) \cdot \frac{\Phi(H)}{\bar{\Phi}} \cdot \left\{1 - \left(p_c \cdot \frac{\delta(H)}{l-1}\right) - o(H) \cdot p_m\right\} \quad (1)$$

The schema theorem can be translated into an equality by including string gains and losses (Whitley 1993). However, even if an equality is used to predict the exact number of schema instances, the schema theorem fails to capture the full complexity of a GA. Many works have been devoted to solve these deficiencies. Some shortcomings come from an unnecessarily restricted interpretation of schemata under binary alphabets (Antonisse 1989). Also, building blocks are difficult to identify in many applications (Whitley 1993), and there is not direct information about the fitness function in the given estimate (Bäck *et al*. 1997).

Nevertheless, the theorem depicts the exponential growth of expected number of instances as a desirable characteristic for a GA, and this still holds whenever it can be attained. Besides this, the schema theorem gives useful information on the GA when "used to predict the number of instances of a given schema only for the *next* generation" (Whitley 1993).

The new interpretation of schemata for non-binary alphabets overcomes the drawbacks of low expressiveness that the initial interpretation assigned to alphabets of higher cardinality (for example integers $V = Z$ or reals $V = I\!R$) (Antonisse 1989). In addition, new extended and alternative explanations to the schema theorem are being proposed and used; see Bäck *et al*. (1997) for a summary of theories on genetic algorithms. Besides, the reader can find in Menke (1997) a correct estimate, and in Poli (1999) a deeper understanding in relation to GA operators.

In this section, we study the schema processing abilities of the models. We proceed in two steps. The first step will be to experimentally confirm the theoretical selection pressures derived in the precedent section in the schema space. The second step is intended to confirm the expected exponentially growth of the number of instances belonging to a solution schema.

The optimization problem used here consists in finding one out of two solution strings of 36 bits. The fitness evaluation (assuming a maximization problem) subtracts from 72 the sum of hamming distances of the evaluated string to each one of these strings (subtracting 0 when one of the two solutions is evaluated). The optimum is thus 72, and the problem is bimodal. In Table 4 we give the strings, the studied schema, the compared algorithms, and the parameters of the problem (see the nomenclature in the Appendix).

The analyzed schema contains the two solution strings as well as another 14 "undesired" strings that are not a solution to the problem. In fact, it is not a very difficult problem, but we use it because it is easy to track the schema proportion in the population. We have selected a schema of high order and long defining length because it is a "bad" schema; to confirm this end, see again the definition of the schema theorem in equation (1).

The results of tracking this schema in the population are very indicative of the internal behavior of the algorithms (Fig. 5). Generational GAs, distributed or not, have problems with introducing this schema in their populations, as predicted by the schema theorem. However, the four other algorithms (d-ssGA and d-cGA) do show an exponentially increased number of instances for this schema. Thus, the first conclusion is that genGA is appropriately described with this theorem, while d-ssGA and d-cGA are not.

In addition, the two generational GAs are very slow. The ssGA is the quicker in exploiting the genotypic material, while its distributed counterpart (dssGA) is better in preserving diversity and smoothing the curve of ssGA, but far from the slow progress of dgenGA. The results for cGA and dcGA show an intermediate pressure level, as expected. Since the population is small

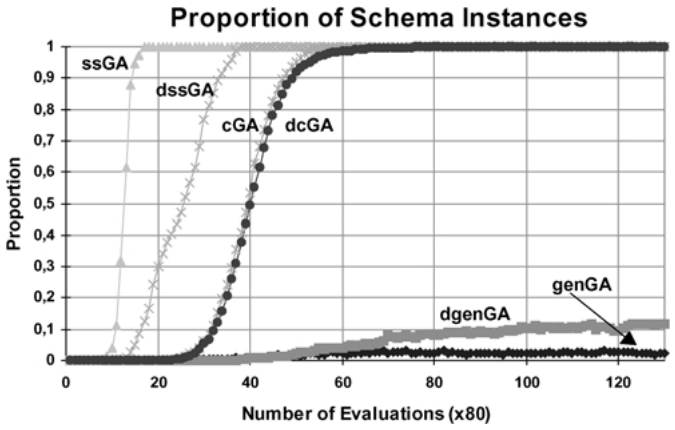


Fig. 5. *Proportion of schema instances for the algorithms (average of 20 runs)*

(80 individuals), cGA and dcGA are not very influenced by the grid shape. This is also the reason of the similarities between the schema processing rates of cGA and dcGA.

Larger differences can be observed either for larger populations and for more difficult problems (with high epistasis, multimodality, ...), in which a distributed algorithm provides a more noticeable impact on the performance. As stated before, our second concern is selection pressure in schema processing. Selection pressure can be also noticed when studying schemata. We can confirm that clear differences in the growth of the better strings appear, even when crossover and mutation are used.

In short, the proportion of instances of the solution schema grows very slowly in generational GAs, even for such a simple problem. The deviation from a desirable behavior is clearly noticeable for genGA and dgenGA. The d-ssGA, and d-cGA do present the theoretically expected "exponential growth" of instances that traditionally has been credited to good GAs, even for this "undesired" schema.

Here, we end with the theoretical side on the internal behavior of panmictic and structured GAs. Until now, we have been offering some results that have suggested some relationship with the migration policy in a parallel dGA, and with the grid shape in a cellular GA. In the next two sections we will get deeper on these matters.

4. Parallel distributed GAs

Parallel distributed GAs have a few parameters that are very important for guiding the search and that distinguish them from other GAs. This section discusses four main issues in parallel distributed GAs, namely their migration policy, effort, synchronous versus asynchronous execution, and speedup. Every subsection will deal with such matters. In addition, further extensions to new problems are analyzed along this section.

4.1. *Migration policy in a distributed GA*

In relation to the migration policy, many authors have reported the benefits of running "loosely coupled" islands (Belding 1995, East and McFarlane 1993, Tanese 1989), both in terms of lower execution time and numerical effort. However, no clear conclusions can be drawn since most of these results utilize non-standard algorithms or include new operators, local search, etc. To save this gap, here we link the migration frequency to the size of the global population and study the effects on canonical versions of dssGA and dcGA.

The migration policy determines the connectivity of the islands in the parallel distributed GA. We define the migration policy as a tuple of five values (Alba and Troya 2000a):

Definition (Migration Policy). A *migration policy* in a parallel distributed GA can be defined by five items:

$$M = (m, \zeta, \omega_S, \omega_R, s) \tag{2}$$

where:

- m: is the *number of individuals* undergoing migration, $m \in \{0, 1, \ldots, \infty\}$ (*migration rate*).
- ζ: is *the frequency of migration* (we express it in number of evaluations), $\zeta \in \{0, 1, \ldots, \infty\}$.
- ω_S: is the policy for *selecting migrants*; usually a copy of an individual is sent to the neighboring island. Alternatively, the individual itself could be sent.
- ω_R: is the *migration replacement policy* used for integrating an incoming individual in the population of the receptor island.
- s: is the *kind of communication*, either synchronous or asynchronous exchange.

We define the migration gap (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 steady-state model must take μ steps to complete one generation (if we follow the traditional nomenclature) while the atomic step of genGA and cGA is to complete a full new generation of μ individuals. Anyway, in all the algorithms a generation needs μ evaluations of the objective function to be completed.

We will study the frequency of migrations as a function of the global population size. Specifically, integer factors such as $1 \cdot \mu, 2 \cdot \mu, 4 \cdot \mu, 8 \cdot \mu$, and fractional factors like $0.25 \cdot \mu$ or $0.5 \cdot \mu$ will be used when the population size (μ) is large enough. The value 0 stands for disconnected distributed evolution.

In this way, the migration ω_M is used in the communication phase (see Algorithm 1 in Section 2) of a PGA Δ_{par}. It is an operator indicating how the structures of one sub-algorithm Δ_i are shared by another one Δ_j. The selection operator ω_S determines which of the structures of the source sub-algorithm Δ_i will be inserted in the target sub-algorithm Δ_j, i.e. the migrated strings; $\omega_S(\Delta_i, \Delta_j)$ is the set of shared structures between neighboring sub-algorithms. If $\omega_S(\Delta_i, \Delta_j) = \emptyset$ then the sub-algorithms i and j are not neighbors:

$$\omega_{M\Theta_M}(\Delta_j) = \omega_R \circ \omega_S(\Delta_i, \Delta_j) \,|\, \forall \Delta_i, \Delta_j \in \Delta_{par} \tag{3}$$

The number of migrated individuals is the number of shared structures in every island:

$$m = |\omega_S(\Delta_i, \Delta_j)| \tag{4}$$

and the probability of application is

$$p_M = \frac{1}{\zeta} \tag{5}$$

Some works strongly suggest to use asynchronous communications (Alba and Troya 2001, Hart *et al.* 1997, Maruyama, Hirose and Konagaya 1993). This can be achieved by inserting an individual whenever it arrives, thus eliminating the necessity of blocking the algorithm every ζ steps (i.e., sending and receiving strings are managed in separate portions of the code).

The set of parameters controlling the migration is defined by the migration policy:

$$\Theta_M = \{M\} \tag{6}$$

The basic reproductive cycle of the dGA is then a composition of the island reproductive cycle and the migration operator:

$$\omega_d = \omega_M \circ \omega_{island} \tag{7}$$

If ω_{island} works on a small pool of spatially close neighbors, then we have a dcGA.

To analyze the influence of the migration policy, we have performed several tests with MMDP15 and RAS20 (see the Appendix). We study the influence of the migration frequency ζ, and the migration selection ω_S ("better" or "random") in a ring of islands. We plot the number of hits on dssGA and dcGA in Fig. 6 (optimum found out of 50 runs). Eight different migration frequencies are analyzed for dseqGA = dssGA and dcGA.

The results indicate that it is better to migrate a random string than the best one, since migration of the best string often generate super-individuals in the target population and diversity is quickly lost. A second conclusion is that loosely coupled islands provide a better efficacy; 16 or 32 are the factors that worked out the best results. A high isolated search time allows quick integration/rejection of incoming migrants, thus provoking a clear non-panmictic evolution in the dGA. The dcGA yields better results than the dseqGA = dssGA in all the tests (with the exception of 16 for RAS20). This tendency is also found in the next subsections.

The forthcoming subsections are devoted to compare the parallel distributed models dssGA and dcGA from additional points of view. We study the numeric effort (4.2), the effects of synchronization (4.3), and the speedup (4.4) of distributed versions of ssGA and cGA. We end in Section 4.5 by summarizing and extending these results.

The parallel dGAs with 1 to 8 processors have been run in a LAN of similar computers (SUN UltraSparc 1, 143 Mhz and 64 Mb RAM) using an ATM communication network (155 Mbps). Since the migration frequency and the synchronization of the distributed algorithms will influence the search we plot different cases using three isolation time values throughout. These values range from a highly coupled algorithm ($\zeta = 1$) to an almost idle set of islands ($\zeta = 4$), with an intermediate value of $\zeta = 2$. Besides that, we compare the effects of using synchronous (**s** label) and asynchronous (**a** label) versions.

We first study a large instance of the generalized Sphere problem with 16 variables and 32 bits per variable ($l = 512$ bits) (problem SPH16-32, see the Appendix for more details). For all the results we use a global population of 512 individuals, two point crossover (TPX) $p_c = 1.0$, and bit-flip mutation $p_m = 1/l$.

4.2. *Numeric effort*

All the numeric results when solving SPH16-32 share some clear characteristics (see Fig. 7). Since this problem is not especially difficult for a PGA, tight migration ($\zeta = 1$) are faster in finding a solution. This is an exception to the "general rule" by which isolated islands are thought to perform better. However, for SPH16-32, a high interaction among the sub-algorithms provides the same effect than using panmixis (one single population) with the added advantage of a much lower execution time.

The dssGA (Fig. 7 left) is more sensible to the number of processors, while the dcGA (Fig. 7 right) is less influenced. This confirms the trends found in Section 4.1 in that the search of a cGA is *numerically* better only for difficult problems (even when distributed).

4.3. *Synchronous vs. asynchronous*

Synchronous and asynchronous versions perform both the same algorithm. Sync and async lines of Fig. 7 are quite similar since the island GAs and the computers are homogeneous. They could

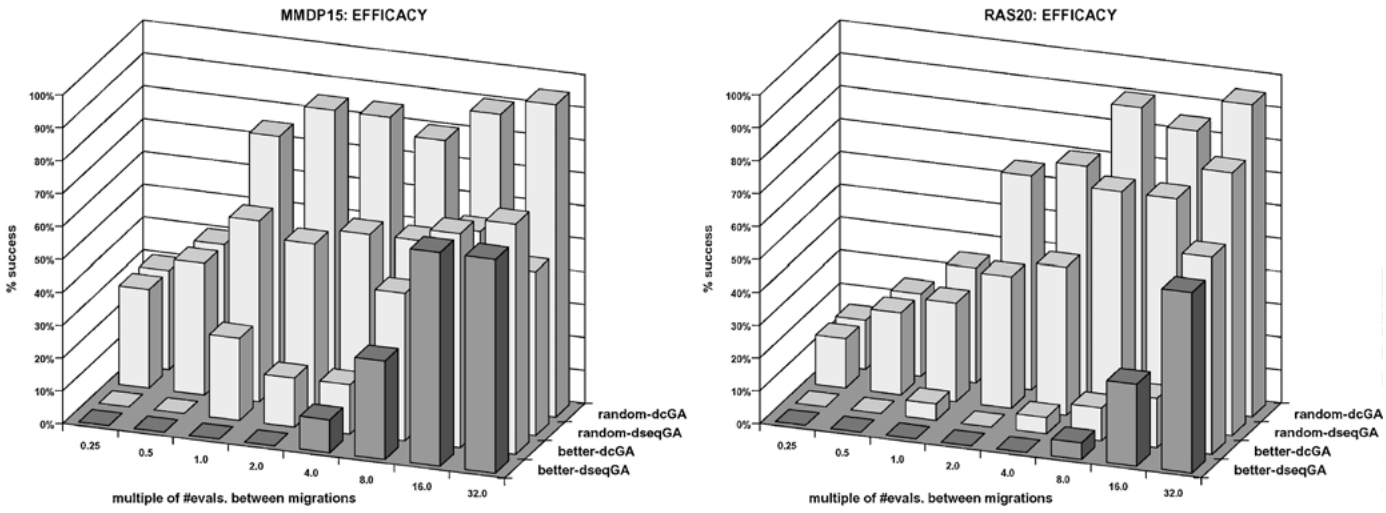


Fig. 6. *Efficacy of several migration policies on distributed GAs for MMDP15 (left) and RAS20 (right)*

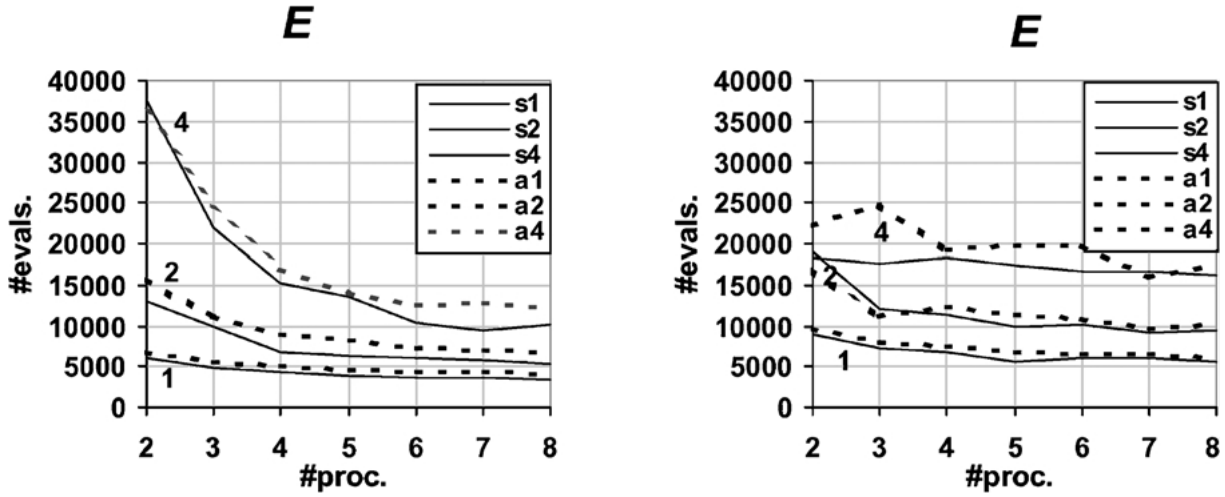


Fig. 7. *Number of evaluations for the dssGA (left) and dcGA (right) for SPH16-32. Three migration frequencies (1, 2, 4) are shown for synchronous (s) and asynchronous (a) versions*

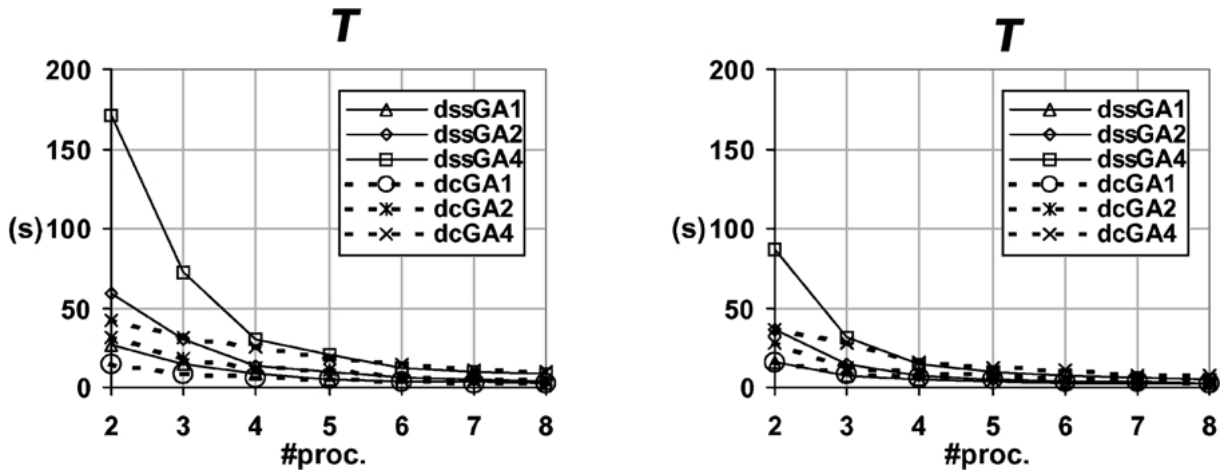


Fig. 8. *Absolute time for solving SPH16-32 with dssGA and dcGA. We plot two separate comparisons: for the synchronous algorithms (left) and for the asynchronous ones (right)*

have shown great differences if variable string lengths or different parameters were being used (heterogeneous search, see again the taxonomy in Fig. 1).

However, the expected execution time (Fig. 8) for different number of processors is consistently lessen by any asynchronous model with respect to its equivalent synchronous algorithm. This is easily explainable since the synchronous dssGA must wait in every iteration, while the asynchronous one must not. Since the number of evaluations to solve the same problem are quite similar for each version, asynchronous models are expected to show a better speedup (next section).

An interesting finding is that, although the dssGA outperforms dcGA numerically, the user response time is smaller for dcGA than for dssGA. Since communications are sought after every step, the higher computational grain of dcGA is advantageous (see the left and right parts of Fig. 8). The dssGA can be improved by ignoring any communication task until having performed a given number of evaluations ($\gg 1$). However, by doing this, the asynchronous model becomes meaningless. Nevertheless, although this is not a good practice for fair comparisons, it could be a good idea for speeding up applications using PGAs.

4.4. *Speedup*

Speedup is a widely used measure allowing to assess the efficiency of a parallel algorithm. It has been being used for many years to analyze deterministic algorithms. It relates the times of running the same algorithm in 1 and n_{proc} processors (equation (8)).

$$S(n_{proc}) = \frac{T_1}{T_{n_{proc}}} \tag{8}$$

However, before discussing speedup in parallel GAs we need to make some especial considerations. When dealing with PGAs

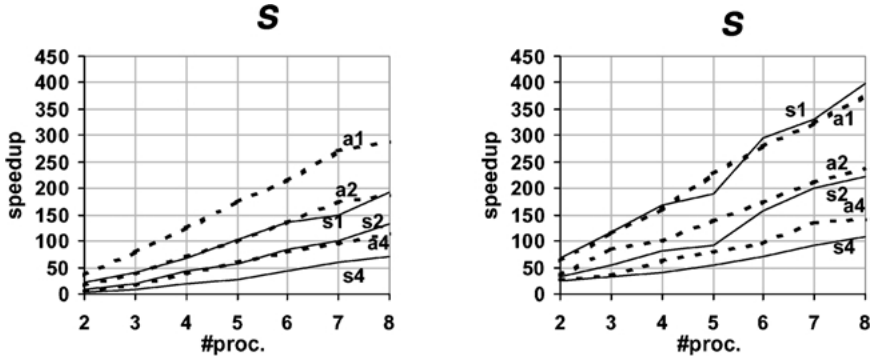


Fig. 9. *Speedup for the dssGA (left) and dcGA (right)*

we must use average times and not absolute times, since PGAs are non-deterministic algorithms (equation (9)).

$$S(n_{proc}) = \frac{\bar{T}_1}{\bar{T}_{n_{proc}}} \tag{9}$$

In addition, as many works have established, sequential and parallel GAs must be compared by running them until a solution of the same quality has been found (Cantú-Paz and Goldberg 1997, Hart *et al*. 1997). Completing the same number of steps *must not* be used as the termination criterion when measuring speedup.

Besides, although speedup is upper bounded by the number of processors n_{proc}, for a PGA it might not, since the PGA reduces *both* the number of necessary steps and the execution time in relation to the sequential one. See superlinear speedups in Belding (1995). The theoretical explanation is that a PGA can find a solution in any time $T > 0$ (Shonkwiler 1993). Also, the separate chunks of data structures can fit into the cachés of processors (while they do not fit in a monoprocessor), thus providing an additional source for superlinear speedup. Therefore, superlinear speedup is possible from a theoretical point of view whatever operators or parameters are being used. But there are still some ambiguities. We can choose to measure the speedup of a distributed GA on n_{proc} processors versus (1) a sequential panmictic GA or (2) versus the same distributed GA on a single processor. These two possibilities are discussed in the next two subsections.

4.4.1. Distribution versus panmixia

In this subsection we compare the time to locate a solution needed by a panmictic GA versus a distributed GA with the same global population and parameters. This is the usual way in which many authors have measured speedup in the past. Hence, we do the same in this section only to establish a bridge from these works to a "more correct" or "fairer" way of measuring speedup that we will introduce in Subsection 4.4.2. The goal of the present section is twofold. First, we want to show that super-linear speedup is possible in practice, wherever the GA is. Second, we want to point out that the EC community must use in the future a more correct and widely accepted way of measuring speedup.

In Fig. 9 we show the speedup of dssGA and dcGA for solving SPH16-32. Two conclusions can be drawn. On the one hand, the speedup is clearly superlinear since e.g., an eight island dssGA (dcGA) is better than the sequential ssGA (cGA) in a factor larger than 8. This is due to its higher diversity and parallel exploration from many different zones of the problem space. The second outcome is the excessively large magnitude of such a speedup. The reason is that a dGA and a panmictic GAs are not (definitely) the same algorithm.

4.4.2. Orthodox speedup

Superlinear speedup is possible from both, theoretical and practical sides. However, the reader could think that other parameters such as the complexity of the selection algorithm is allowing this gain in speed when changing from panmictic to distributed GAs. Splitting the population can of course reduce the execution time because resulting sub-populations are smaller. But, if this were the source of superlinear speedup, we could not get a speedup larger than n_{proc} when comparing the *same distributed* algorithm on 1 and n_{proc} processors.

In this section we show that exactly the same multi-population algorithm can show superlinear speedup (Alba and Troya 2001).

Incidentally, in deterministic algorithms, superlinear speedup can arise since the data structures are split into smaller pieces that fit into the cachés of the processors, while the whole data structure did not fit in because of its larger size. This source of superlinear speedup can also appear when using non-deterministic algorithms such as PGAs.

For these tests, we use a total of 512 individuals separated into the number of islands. The algorithm is an homogeneous parallel dssGA in which all the islands perform proportional selection, two point crossover with probability $p_c = 1.0$, and bit mutation with probability $p_m = 1/\ell$, being ℓ the string length. We show results with the Sphere problem having 16 variables each one encoded in $l_x = 32$ bits ($\ell = 512$), and with an instance of the MMDP deceptive problem with 40 unitation $l_x = 6$ bit segments ($\ell = 240$)—see the Appendix.

In the following graphs we call "weak1" to a speedup measure in which the base case is the panmictic GA and the

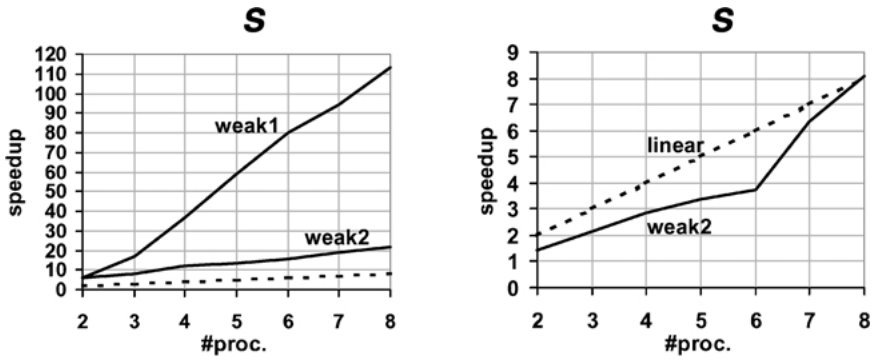


Fig. 10. *Weak definition type 1 versus type 2 for SPH16-32 (left) and MMDP40 (right)*

distributed GA has the same number of islands as processors. We call "weak2" the measure in which the distributed GA has always a constant number of islands, and they are allocated to the available processors. The size of the population is always constant and equivalent between the compared algorithms. In both cases the measure is called "weak" because a "strong" speedup can only be defined when comparing with the worst time of the fastest sequential algorithm, as the traditional definition of speedup requires (Akl 1992). We use the weak measure because we are only considering one type of GA with no warranties that it is the best algorithm to solve the problem.

Figure 10 left shows that superlinear speedup is possible when comparing a distributed algorithm on n_{proc} processors with a panmictic one (*weak1*), and also when comparing it to itself on one processor (*weak2*). This is especially clear for the SPH16-32 problem. For MMDP40, the weak2 measure is slightly sublinear, but the trend is to increase toward linear and moderate superlinear speedup when more processors are used. Of course, one cannot expect to achieve superlinear values for any algorithm and problem, and MMDP40 is an example of such a case we include to asses the correctness of our measures.

In short, superlinear speedup is possible, both theoretically and practically, in PGAs.

4.5. *Quick summary and extensions of the results*

To summarize the mentioned conclusions, dcGA worked out the overall best results measured in terms of real time. Besides that, the results of precedent sections and the scaling properties presented in this section give dcGA some advantages. The dcGA is more robust, and increasing the number of processors is more beneficial for it than for dssGA. The time of dcGA for easy GA-problems like the Sphere becomes very similar to dssGA for 8 processors, and dcGA shows many other advantages when faced to difficult problems.

Let us mention that the speedup is not always that large in all the applications of a PGA, as it might be suggested from graphs in Fig. 9 and 10 (left). More difficult problems do not allow for such values. However, the superiority of asynchronous PGAs over synchronous ones has been also reported frequently in these years (Hart *et al*. 1997), (Maruyama, Hirose and Konagaya 1993).

A dssGA with frequent migrations could consume an unnecessarily large communication time. This is where a distributed algorithm with a larger grain of computation such as dcGA could show its advantages over dssGA.

Although the parallel characteristics of dcGA seem to be superior to dssGA we cannot recommend it as "the best" distributed PGA for global optimization (the No Free Lunch theorem prevents on this matter (Wolpert and Macready 1997)). In fact, complex applications in which final parameter tuning is very important are often better solved by using dssGA. Training a neural network is an example of such a complex domain. It has high epistasis (genotype linkage), multimodality and nonlinearities. Some dssGA-like algorithms such as GENITOR II (Whitley and Starkweather 1990) have proved to be useful in the past for this task.

In Fig. 11 we show the influence of the synchronization and migration frequency for both, a dssGA and dcGA used to train the neural network described in the Appendix. We confirm two conclusions: asynchronous algorithms need lower (often much lower) execution times, and they are less influenced by the migration frequency.

These results prove that asynchronous algorithms are better in easy and in complex domains, and that applications

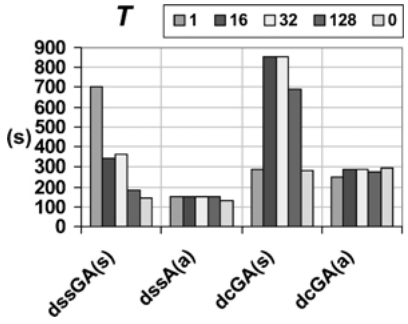


Fig. 11. *Time to train a neural network with dssGA and dcGA using 8 processors. We plot synchronous (s) and asynchronous (a) versions in the graph. Migration frequencies 1, 16, 32, and 128, and total isolation –0– are evaluated*

of distributed PGAs should prefer them over synchronous implementations (Alba and Troya 2001).

A somewhat surprising result is that the evolution of non-connected islands performs quite well in this problem. The reason can be found in the characteristics of training multilayer perceptrons with EAs in general. Since the strings represent sets of weights, and since each set of weights is a given functional mapping of the training patterns, crossing two strings is most times useless. The resulting offspring often lacks of the functional modules that are present in its parents. In general, crossover is not very extended in training NNs with EAs, we use it here for avoiding long explanations on when and how it is useful. This effect promotes algorithms that only merge two parent strings when they have very similar contents. This kind of intra-species crossover is natural and implicitly provided by a multi-population PGA, thus explaining the good behavior of disconnected executions, where isolated species evolve.

In this application, the dcGA is a slower mechanism for final tuning of the solution in relation to the dssGA. This is why many authors directly propose parallel models having a hill-climbing technique that works inside the neighborhoods of a cGA (Mühlenbein, Schomisch and Born 1991).

5. Cellular GAs

In this section we discuss the importance of the neighborhood and topology in the cellular GA. The influence of these two parameters on the performance of a cGA has already been suggested but it is only in recent works that common numerical models are gaining acceptance (Sarma and De Jong 1997). Since cGAs are not widely known optimization algorithms we present a detailed study on them.

Concretely, we first characterize the shape/neighborhood relationship with a numerical ratio value in Subsection 5.1. Then, we analyze the influence on the ratio in the effort for obtaining a solution (Subsection 5.2), in the efficacy of the algorithm (Subsection 5.3), in the scalability of the algorithm (Subsection 5.4), and, finally, we address a problem-dependent ratio performance study (Subsection 5.5).

5.1. *Ratio between the neighborhood and topology radii in a cGA*

Many parameters distinguish a cellular GA from other kinds of algorithms. In particular, the topology in which individuals are placed, and the neighborhood inside which the reproductive plan is applied are very important. This is common sense since the population is not a pool like in panmictic GAs. In cGAs the population is structured, usually in some kind of grid in which one string is located in every node. Toroidal meshes are very usual, with the grid folded to a torus. A geographical distribution like this one needs also some rules to decide which strings belong to the same neighborhood, for every given string in the population.

Different neighborhoods can be defined on a toroidal mesh of individuals. See some possible neighborhoods in Fig. 12.

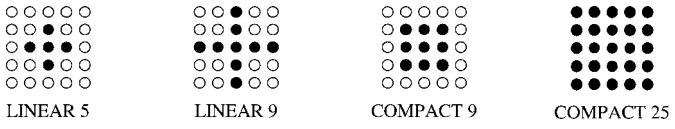


Fig. 12. *Types of neighborhoods: Linear and compact*

We fill in black the strings belonging to the same neighborhood hosted by the central string. All these neighborhoods share some common characteristics (Baluja 1993). Usually, they all are composed by a small number of strings in order to reduce the communication overhead in a SIMD machine, or the execution time if a monoprocessor implementation is being used to hold the grid. Many authors use the NEWS (North-East-West-South) neighborhood since it is very simple (also called linear5).

Such a large diversity of topologies and neighborhoods needs a common quantification. The results in De Jong and Sarma (1995), and Sarma and De Jong (1996) proved that the relationship between neighborhood and topology determines the selection pressure of a cGA. Here, we provide an independent and different confirmation of their theory by using a quantity to characterize the topology, called "the radius". We consider the grid to have a radius representing the dispersion of n^* points in a circle centered in $(\bar{x}, \bar{y})$ (Fig. 13).

This definition can quantify the type of neighborhood as well, and thus it helps in distinguishing different neighborhoods that might be defined on the same toroidal grid.

For a constant number of individuals ($n = n^*$) and neighborhood the radius will increase as the grid gets thinner (Fig. 14(a)). Therefore, the thinner is the grid the smaller is the overall ratio. In Sarma and De Jong (1996) it is shown that different grids and neighborhoods having the same ratio show also the same selection pressure. This and other works (Baluja 1993) change the type of neighborhood to get different selection pressures on the same grid. Our contribution consists in keeping a constant neighborhood (NEWS) and changing the grid shape. Therefore, we can tune the selection intensity by changing a single parameter:

$$rad = \sqrt{\frac{\sum (x_i - \bar{x})^2 + \sum (y_i - \bar{y})^2}{n^*}}$$

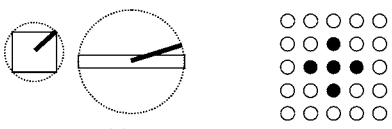
$$\bar{x} = \frac{\sum_{i=1}^{n^*} x_i}{n^*} \qquad \bar{y} = \frac{\sum_{i=1}^{n^*} y_i}{n^*}$$

(a)

$$ratio_{cGA} = \frac{rad_{neighborhood}}{rad_{topology}}$$

(b)

Fig. 13. *(a) Radius, and (b) the ratio between the topology and neighborhood radii*



$$rad_{linear5} = \sqrt{\frac{2+2}{3+2}} = 0.8944$$

(c)

Fig. 14. *Two radii (a), NEWS neighborhood (also known as linear5) (b), and its radius (c)*

Table 5. *Parameters of the tests*

Parameters	ras10 (16 bits)	ras20 (8 bits)	ros10 (16 bits)	ros20 (8 bits)	sph3 (32 bits)	sph6 (32 bits)	ummdp6 (6 bits)	mmdp16 (6 bits)
Problem size	10	20	10	20	3	6	6	16
String length	160	160	160	160	96	192	36	96
p_c	1.0	1.0	0.8	0.8	1.0	1.0	1.0	1.0
p_m	0.01	0.01	0.03	0.03	0.01	0.01	0.05	0.05

the ratio (Alba and Troya 2000b). This is much easier to implement, use, and study (see also Alba, Cotta and Troya 2000).

5.2. *Ratio and effort*

Reducing the ratio in some way means reducing the global selection pressure in the cellular GA. This is expected to allow a high diversity and promote exploration.

To show the influence of the ratio in the search we applied different cGAs to solve a set of problems and then compared the resulting number of steps. Parameters are in Table 5 and results in Fig. 15. We have used canonical versions and have not carried out any special parameter tuning to avoid biasing the results.

In Fig. 15 we can appreciate the computational effort for solving every problem in terms of the number of evaluations (average of 50 independent runs). Clearly, ros20-8 and ras20-8 are the more difficult problem instances among all, and mmdp16-6 is the easier one. However, our goal is not to compare different problems, but different grid shapes on the same problem. For a given problem, the number of evaluations to get a solution is essentially the same for any grid except for the 2×50 and 1×100 shapes (very small ratios). For these thin grids the algorithm needs a larger number of evaluations to get a solution. We will validate this outcome also on much larger populations later in this section. Thinner grids provide a higher diversity and exploration (advantage), but this usually requires a larger number of evaluations (drawback). An intermediate ratio might be the best trade-off decision.

5.3. *Ratio and efficacy*

In this section we show that there exists a relationship between the number of hits in finding a solution to difficult problems and thinner grids. See Fig. 16 to verify this point. The smaller problem instances are successfully solved whatever the disposition of individuals is (right half of graph in Fig. 16). But for the two instances of Rosenbrock and Rastrigin functions (10 and 20 variables) thinner grids are progressively better.

Therefore, we can conclude that thinner grids provide better efficacy on difficult problems (with bounded and constant computational resources). The increment in the number of evaluations shown by small ratios is a practical drawback for "easy" problems, but an advantage when analyzing the resulting efficacy. There exist some alternatives that could alleviate this

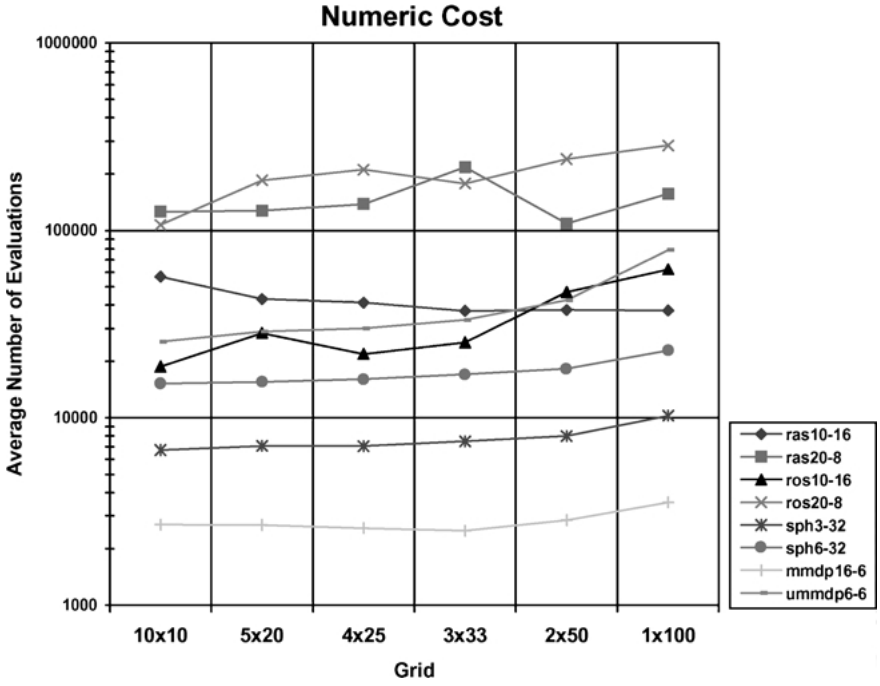


Fig. 15. *Efficiency in terms of the average #evaluations to solve the problems*

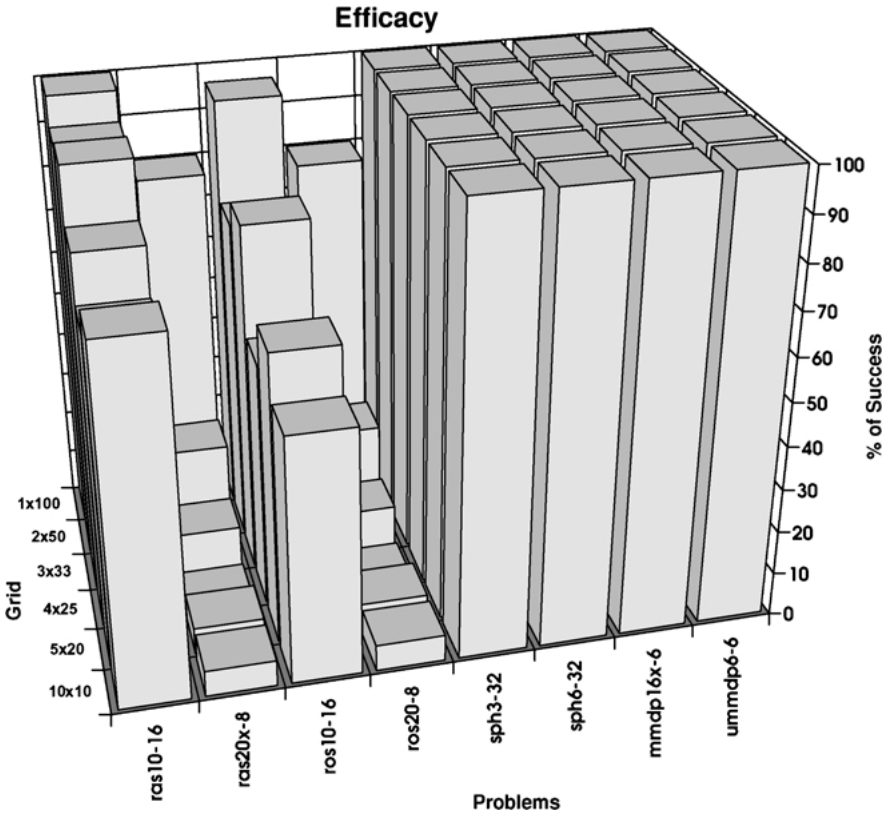


Fig. 16. *Percentage of successful runs (out of 50) in solving multiple problems with different shapes (i.e., different ratios)*

higher number of evaluations. For example, instead of using a cGA with a large grid, we could run several connected islands of smaller cGAs in parallel; this algorithm, which we have called dcGA, would profit from the physical and numerical benefits of the migration models, like shown in Alba and Troya (2001).

5.4. *Ratio and scalability*

Now we turn to consider the way in which the *same* cGA can solve problems of increasing size. This will give us an idea of how the cGA exploits and profits from the genotypic information. The better the cGA scales the better it is supposed to perform when faced with unseen problems. Table 6 contains the parameters used to get the results shown in Fig. 17 (only non-distributed algorithms). We use small population sizes because our interest in using bounded computational resources (wide applicability). As in the previous tests, 2×64 means we are using a grid of 128 individuals geographically distributed in 2 rows of 64 strings each. The same holds for the rest of population size values.

For all the instances of a given family the cGA is exactly the same. Let us show in Fig. 17(left) how the algorithm is able to manage the instances of easy problems like the SPH16-32 without important increments in the numeric effort. Problems like ROS and RAS need a clear larger number of evaluations.

The graph in Fig. 17 left suggests the relative difficulty that the cGA face to solve these two problems, much more evident for the ROS family due to its epistasis. In addition, deceptive problems (MMDP) provoke large jumps on the efficacy for very large instances.

It is important to keep in mind the relationship between cGA and other algorithms such as ssGA and genGA. We compare in Fig. 17 right their percentage of hits for several ROS instances. When analyzing results that only include cGAs one could be prone to think that some grid shapes are not efficient.

Table 6. *Parameters for solving SPHxx, RASxx, ROSxx, and MMDPxx with a cGA*

Parameters	SPH {1, 2, 3, 4, 5, 6}	RAS {10, 12, 14, 16, 18, 20}	ROS {10, 12, 14, 16, 18, 20}	MMDP {1, 2, 4, 8, 16}
Pop. shape & size	$2 \times 64 = 128$	$2 \times 50 = 100$	$2 \times 95 = 190$	$2 \times 24 = 48$
p_c	1.0	1.0	0.8	1.0
p_m	0.01	0.01	0.03	0.05
Max. funct. eval.	300000	300000	450000	300000

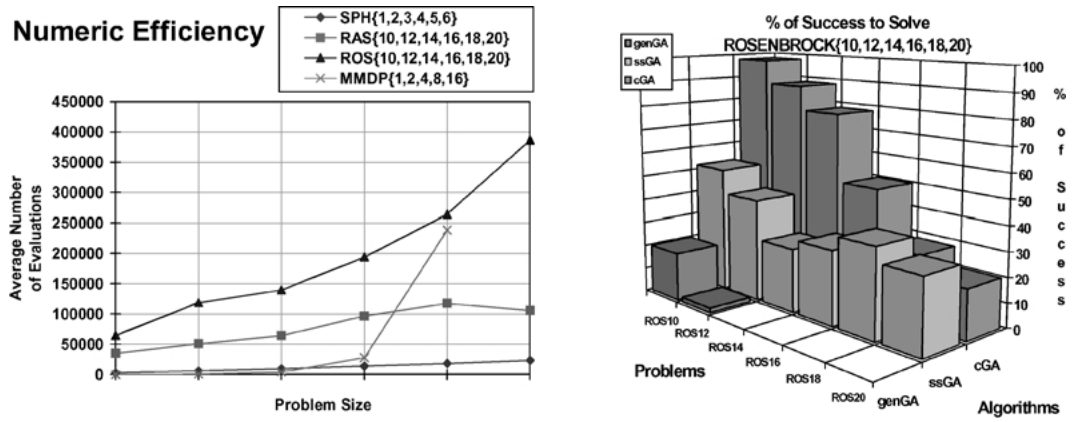


Fig. 17. *(left) Scaling properties of a cGA and (right) percentage of success for Rosenbrock of a cGA versus ssGA and genGA (parameters in Table 5, 50 executions)*

This conclusion might be wrong since even in these cases the cGA could outperform panmictic GAs. This is the reason to include Fig. 17 right at this point. Since ssGA and genGA have not a structured population, convergence to sub-optimal solutions is often encountered. In particular, the genGA is very slow.

Although we have been able to solve many problems with small populations, one can wonder if the scaling of a cGA still holds for much larger populations and difficult problems. We have used a total population of 1024 individuals and solve MMDP of 15, 20, 25, 30, 35 and 40 sub-functions. The graphs in Fig. 18 reveal the computational effort (left) and the scalability (right) of different shaped grids.

The numerical effort grows similarly for grids 32×32, 16×64 and 8×128. These are good news since, in addition, the thinner grids maintain a good diversity and a larger number of hits than square grids in other problems like Rastrigin or Rosenbrock. The very thin grid 4×256 needs a larger computational effort, as expected after the previous results. The growth in size is directly followed by the growth in the effort to solve the problem, with a slight advantage for the square grid (high ratio). Even a very small ratio such as 4×256 scales well.

As a conclusion, we see that thinner grids perform well in difficult problems and that they scale adequately. A square grid scales slightly better than thinner grids, has similar computational effort on MMDPxx, and worst results in the previous problems. The exception is for extremely thin grids in which the larger computational effort is only admissible when a complex problem is being solved.

5.5. *Ratio and problem dependent behavior*

We now proceed to present and analyze the results of our cGA when solving two different problems with various ratios. These problems are the multimodal deceptive MMDP40 function, and a highly epistatic instance of a problem generator known as the P-PEAKS problem (De Jong, Potters and Spears 1997). See the

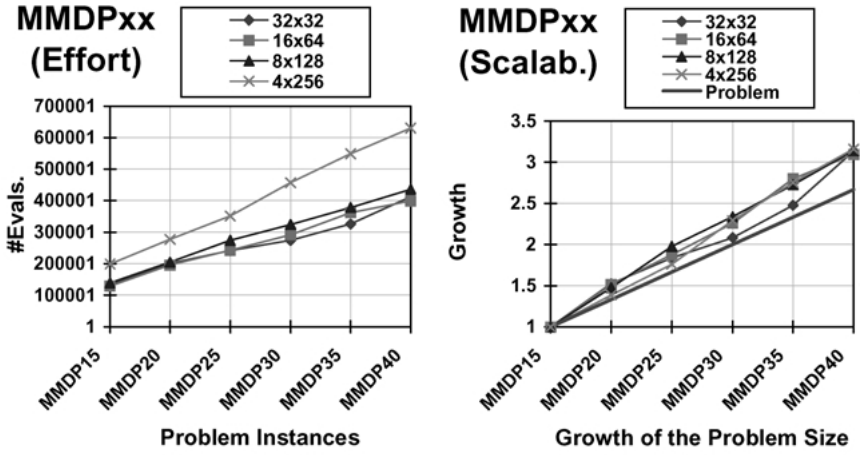


Fig. 18. *Numeric cost (#evals.) to get a solution in a cGA (left) and its scalability (right) for different shaped cGAs and increasingly larger instances of MMDP*

Table 7. *Parameters for solving MMDP40 and P-PEAKS*

Total popsize	$\Rightarrow$ 400 individuals
String length (ℓ)	$\Rightarrow$ 240 for MMDP40, and 100 bits for P-PEAKS
Parent's selection	$\Rightarrow$ Roulette wheel + Roulette wheel
Crossover	$\Rightarrow$ Two point, $p_c = 1.0$
Bit mutation	$\Rightarrow$ $p_m = 1/\ell = 0.0042$, $p_m = 1/\ell = 0.01$
Replacement	$\Rightarrow$ Rep_If_Better

Appendix for more details on these optimization tasks. For these two problems we have used the parameterization summarized in Table 7. Our goal is to prove that there exists no globally better value of ratio, but instead, every problem is more suited for a given value of ratio.

The selected techniques and parameter values are very usual in EA's in order to get results widely useful for the research community. We use 400 individuals (of bit length ℓ), proportional selection of two parents within the neighborhood of 5 strings, two point crossover with probability 1.0, and bit-flip mutation with probability $1/\ell$.

For every point in the grid, we replace it only if the new string is better than the existing one. The stopping condition for all the algorithms and problems is to find a solution. We analyze the cost of solving a problem by measuring the number of function evaluations. For every problem, the same cGA is applied in which we only change the ratio between neighborhood and topology. Three clearly different ratios are used corresponding to the grid shapes 20×20, 10×40, and 4×100 individuals. The computed ratio values with NEWS is 0.110, 0.075 and 0.031, respectively (see Table 8).

The results are shown in Table 9. We summarize the average results of 30 independent runs for solving the two problems. We can see that a small ratio (0.031) is more efficient than a high ratio in P-PEAKS. For this epistatic problem the reduced selection pressure of a small ratio improves exploration. However, for MMDP40, the higher pressure of the square grid (0.110) is more efficient. We underline the best results in Table 9.

These values confirm the previous results in which a high ratio is more efficient in the absence of epistatis, and also for a medium degree of multimodality (MMDP40). A small ratio is desirable when epistasis or multimodality are high (P-PEAKS), because it speeds up the search and quickly gets out of local optima. In these cases, the driving force of the search resides in that different grid regions are mapped to different search regions, sought in parallel.

Table 8. *Three different grid shapes, their radius, and their ratio with NEWS*

Shape	$\text{rad}_{topology}$	rad_{NEWS}	Ratio
20×20	8.15	0.8944	$0.10968 \approx 0.110$
10×40	11.9	0.8944	$0.07519 \approx 0.075$
4×100	28.9	0.8944	$0.03096 \approx 0.031$

Table 9. *Mean number of evaluations*

	0.110	0.075	0.031
MMDP40	<u>170637.9</u>	237003.4	331719.6
P-PEAKS	50458.1	50364.6	<u>48653.6</u>

Table 10. *MMDP40 and P-PEAKS: t-test (p-values)*

	MMDP40			P-PEAKS		
ratio	0.110	0.075	0.031	0.110	0.075	0.031
0.110	–	0.0177	5.12e-06	–	0.9409	0.0829
0.075	0.0177	–	0.0181	0.9409	–	0.1531
0.031	5.12e-06	0.0181	–	0.0829	0.1531	–

Since we want to be sure that these results are statistically significant, Table 10 contains the results of performing t-tests on the average results of Table 9. If we consider the standard $p\text{-}value = 0.05$ level of significance, all the results with MMDP40 are meaningful, thus indicating that a high ratio is more efficient. The significance with P-PEAKS is above 0.05 (around $\sim$0.1), which only allows us to say that there is "a trend" toward smaller ratios. However, the tests have had a great regularity and the smaller ratios yielded the faster algorithms for P-PEAKS.

6. Numeric efficiency

After considering such a large number of behaviors and algorithms we now turn to provide an overview on their search features. Therefore, this section contains an empirical vision of the theoretical behavior of the algorithms. We will track the progress of the observed best string in the population of serial and parallel GAs. Besides the mentioned algorithms, non-elitist versions (**ne** labels) are analyzed to confirm the popular thought that at least the best string must survive between consecutive generations (*elitist selection*, label **e**). Non-elitism is rarely found in algorithms used to solve (real) problems with large computational requirements.

The global population size is kept the same for all the results in this section (180 individuals), and one random string is migrated every $32 \cdot TotalPopsize$ evaluations (loosely coupled) in distributed GAs, as Section 4.1 suggests.

We only plot RAS20 in Fig. 19 as a representative example, although we carried out similar tests for all the problems (average results of 50 independent runs). We conclude (Fig. 19) that the basic ssGA –**ssa**– and cGA(ifb) –**ci**– algorithms are the best among all of them in reaching an optimum with a fast convergence. The algorithms having selection pressure SP2 and SP3 (see Section 3.2) perform with the highest efficiency. The distributed execution maintains or enlarges the numerical efficiency (e.g., in genGAs), and improves algorithms having SP1 (see the rank in Table 11) although they still converge slowly.

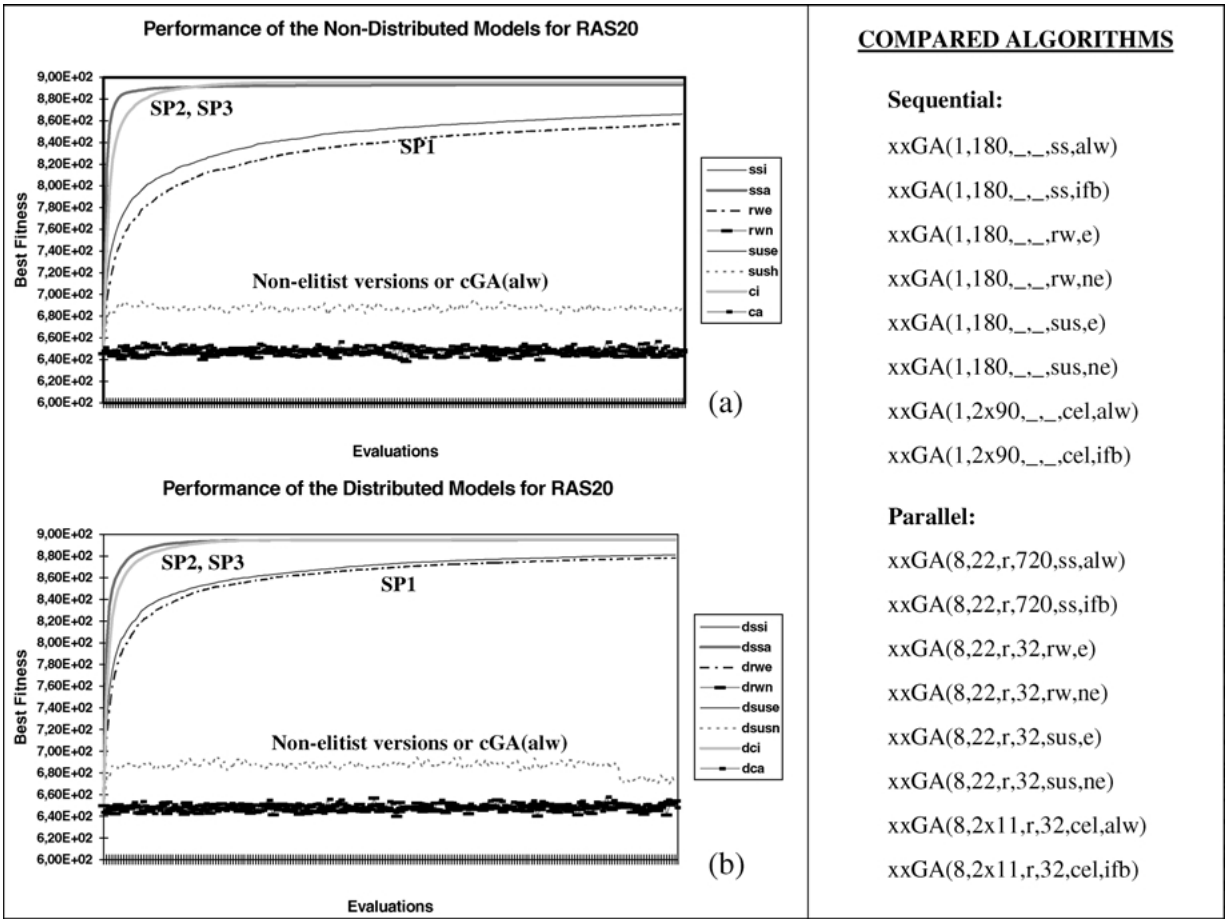


Fig. 19. *Performance on RAS20 ($8 \times 20 = 160$ bits, 180 individuals, $p_c = 1.0$, $p_m = 0.01$, plotted during 540000 evaluations). (a) Non-distributed and (b) distributed*

Clearly, the non-elitist generational GAs and the cGA(always)—*always* replacing every considered string with the newly computed one—are the worst algorithms. This confirms the importance of using elitism (binary tournament) in the replacement of a cGA, and the low convergence velocity of genGAs. In addition, it is clear that roulette wheel selection (**rw**) shows a larger sampling error than stochastic universal sampling (**sus**) (Baker 1987). Our contribution is to conclude that this result holds not only for serial GAs, but also for distributed parallel GAs.

In Fig. 19(b) the distributed versions (8 islands) confirm the ranking in Table 11. Distribution seems to speed up the slow algorithms, and tends to unify the behavior of similar non-distributed selection pressures.

Table 11. *Ranking*

1. dssi, dci, dssa
2. ssi, ci, ssa
3. dsuse, drwe
4. suse, rwe
5. dsusn,susn
6. ca, dca
7. drwn, rwn

7. Resistance to scalability

In this section our goal is to find out the average number of evaluations needed to reach a solution and how it scales with the increasing difficulty of the problems. The analysis include the algorithms xxGA(. . . ,ss,ifb), xxGA(. . . ,cel,ifb), and xxGA(. . . ,sus,e), either distributed (**d** label) and not. Here, we want to extend and unify the results obtained only with cellular GAs (Section 5.4) to include all the algorithms.

We allow a maximum of $3 \cdot 10^5$ evaluations ($4.5 \cdot 10^5$ for Rosenbrock) and make a *constant* and single parameterization *for all the instances* of a given problem. Figure 20 plots the results. The parameters have been provided in Table 6.

In Fig. 20 we can see some general trends. First, the number of evaluations in generational GAs, distributed or not, grows very quickly with the problem size for all the problems, with a lower number of steps if distributed. This is an undesired behavior.

Second, the ssGA is slightly faster than the cGA and their distributed versions for RASxx and ROSxx.

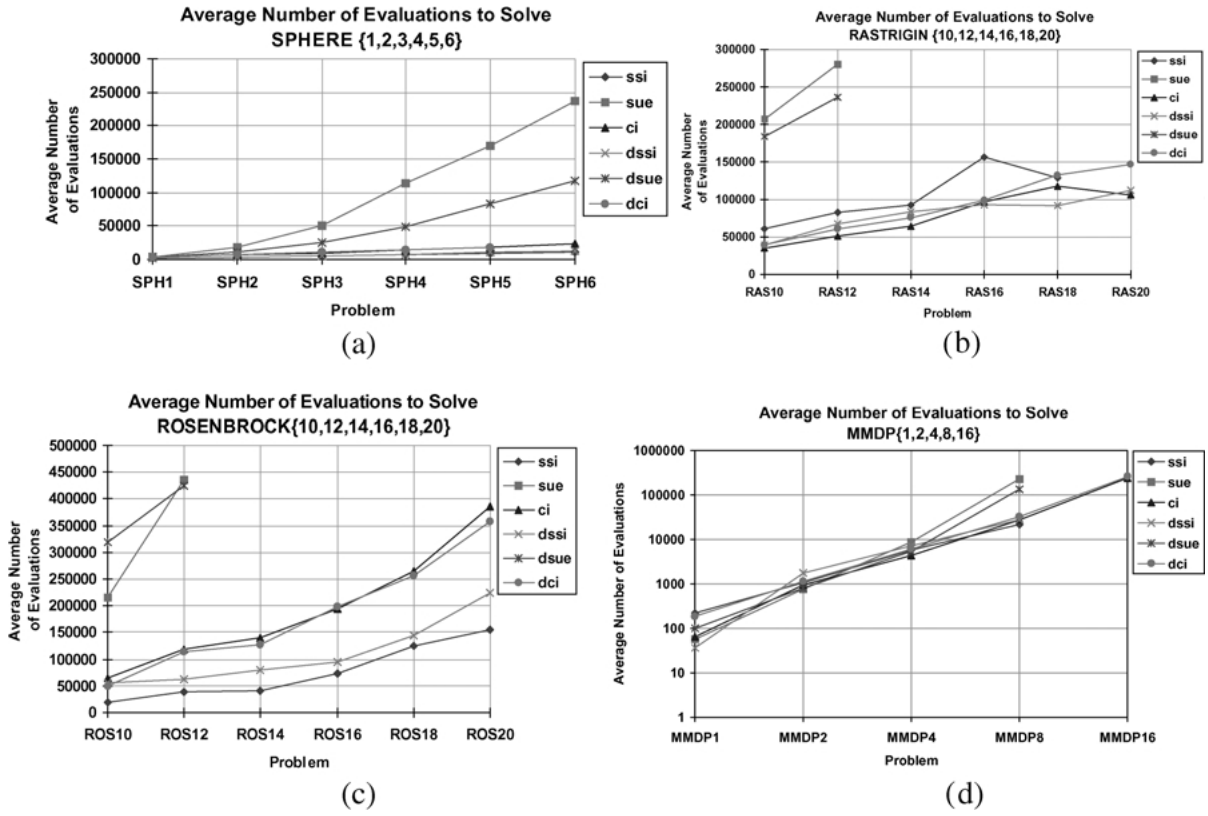


Fig. 20. *Growth of the computational cost for different instances of (a) SPHxx, (b) RASxx, (c) ROSxx and (d) MMDPxx functional families. Mean of 50 independent runs*

Third, the distributed cGAs would have needed larger subpopulations for showing clear numerical benefits. A small population prevents elementary cGAs to show their intrinsic properties since the selection pressure is quite large.

The results for other difficult problems like MMDP16 are more conclusive. With the same resources, the cGA and dcGA are the only ones efficiently solving the largest instances or even solving the problem at all. The steady-state even fails to solve MMDP16.

From an additional point of view, the cGA and dcGA (Fig. 21) give the best efficacy on these problems. There are some exceptions for ROS18 and ROS20 in which the ssGA is slightly better than the cGA.

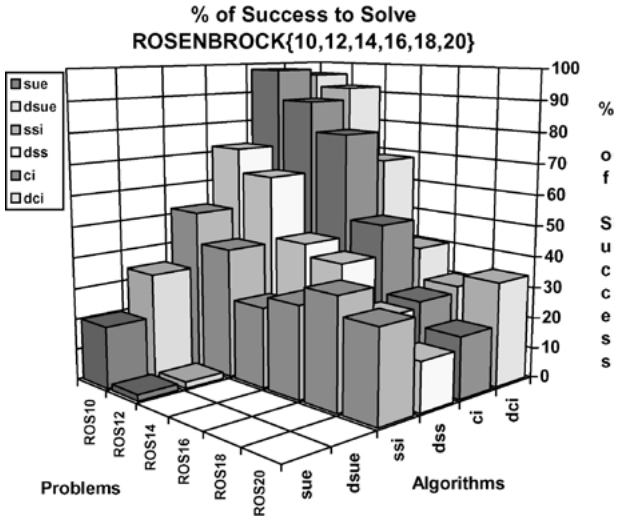


Fig. 21. *Plot of the % of success for ROSxx*

8. Conclusions and future work

In this paper we have presented a unified point of view for the class of sequential and parallel GAs. This has allowed to compare a large number of algorithms fairly. Since there are many versions in the literature for any class of GA and PGA we use popular canonical expressions of them, by stressing the importance of the underlying distributed, cellular, or mixed model.

The cellular GA is slower than the steady-state in many cases, but better in efficacy for most tested problems. Local selection procedures and other refinements could greatly improve the search with cGAs. We have confirmed the importance of the ratio between the radius of the topology and the neighborhood, in cGAs. This is a tradeoff parameter which has to be considered for obtaining good solutions efficiently.

The distributed and non-distributed ssGA and cGA algorithms yield a fast schema growth in the studied problem.

Table A.1. *Symbols used for referring to the algorithms*

Symbols	Meaning
*xx*GA(d, n, m, g, e, re)	$d \in \{1, \ldots, \infty\}$ number of sub-populations running in parallel $n \in \{1, \ldots, \infty\}$ size of one population $m \in \{$r, better$\}$ controls whether we migrate a random or the best individual $g \in \{1, \ldots, \infty\}$ migration gap (steps) between two successive migrations $e \in \{$ss, rw, sus, cel$\}$ evolution mode: steady-state, generational or cellular $re \in \{$alw, ifb, e, ne$\}$ replac. policy: always, if better, elitist or non elitist
ssi ≡ *steady-state if_better*	ssa ≡ *steady_state always replace least-fit*
rwe ≡ *roulette-wheel with elitism*	rwn ≡ *roulette-wheel without elitism*
suse ≡ *stochastic universal sampling with elitism*	susn ≡ *stochast. universal sampl. without elitism*
ci ≡ *cellular GA with if_better replacement*	ca ≡ *cellular GA with always replacement*
dssi, dssa, drwe, drwn, dsuse, dsusn, dci, dca ≡ *distributed versions of the algorithms*	

However, the predicted and actual number of instances are quite different in our tests for these algorithms, indicating that the basic schema theorem is not valid for them.

Distributed parallel GAs seem to be a widely useful and available technique that researchers can use to solve their particular optimization problems. When applied, there is no need of renouncing to the advantages of the structured populations found in massively parallel GAs. The distributed model can be combined with a diffusion model to outperform traditional panmictic islands in many aspects and problems.

The parallel asynchronous distributed versions of all the tested algorithms have worked out better results in time and speedup than the synchronous ones. If the reader has similar conditions to those used in this paper (similar workstations, parameters, or problems), and after other referenced papers, he/she should prefer asynchronous parallel distributed GAs for a novel application. Also, we have shown that superlinear speedup is possible in PGAs.

As a future work it would be interesting to discover if the merging of steady-state and cellular GAs in the same ring should profit from their relative advantages. Besides that, a study on the influence of using ranking or tournament selection would be useful for future applications. Of course, the ability of parallel GAs to face modern optimization domains such as dynamic optimization could be of interest due to their very good diversity and multi-solution capabilities.

Appendix: Nomenclature and testbed

In this Appendix we first present the nomenclature (see Table A.1) used to label the charts. Then we present the optimization problems we have used throughout this paper.

In Table A.1 we summarize how we refer in the paper to different algorithms, number of processors, selection techniques, etc.

For the test suite we have selected problems that satisfy a large set of requirements, namely

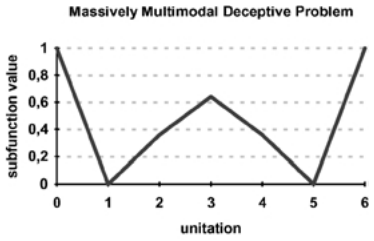
1. Lab and real world problems.
2. Problems showing epistasis i.e., high parameter linkage.
3. Problems showing multimodality i.e., having numerous local optima and multiple different global optima.
4. Deceptive problems, designed to mislead the search of a GA.
5. Simple problems, with the aim of offering baselines for comparisons.

The reader will easily classify the following problems into one of these classes. We have selected a heterogeneous test suite that contains some of the most popular features that a researcher can face in real world optimization. For our tests we have used various problems instances (Fig. A.1): the generalized Sphere (gene –parameter– length $l_x = 32$ bits), Rosenbrock ($l_x = 8$ and 16 bits), and Rastrigin ($l_x = 8$ and 16 bits) functions

$$f_{SPH}\left(x_i|_{i=1..n}\right) = \sum_{i=1}^{n} x_i^2 \qquad x_i \in [-5.12, 5.12]$$

$$Ras(x_i|_{i=1..n}) = 10 \cdot n + \sum_{i=1}^{n} x_i^2 - 10 \cdot \cos(2 \cdot \pi x_i)$$

$$Ros(x_i|_{i=1..n}) = \sum_{i=1}^{n-1} 100 \cdot \left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2$$



$$f_{MMDP}(\vec{x}) = \sum_{i=1}^{k} unitation(x_i)$$

Fig. A.1. *Sphere, Rastrigin, Rosenbrock, and massively multimodal deceptive problems*

(maximized). Also, several instances of a 6-bit massively multimodal deceptive problem ($l_x = 6$ bits).

This set of problems is expected to cover a wide and representative spectrum of search spaces. We use sometimes abbreviations like ROS, MMDP, or SPH for these functions since they help in clarifying the discussion.

We address an additional optimization problem consisting in training an artificial neural network (ANN) for urban traffic noise prediction defined in Cammarata *et al.* (1993). We will use our PGAs to solve this problem. Training neural networks with GAs is an alternative to gradient-descent techniques which frequently get stuck in local optima. Many approaches exit to genetically designed ANNs, although we are only interested in genetically *training* ANNs. This is a difficult, epistatic, and multimodal task that additionally requires a final tuning of the quality of the solution. See Whitley and Schaffer (1992) and Alba, Aldana and Troya (1993) for more details and interesting results in this kind of application.

The network is trained to determine functional relationships between the road traffic noise and some physical parameters. Should this goal achieved, it is possible to modify the causes of traffic noise in order to sensibly reduce it. The neural network is capable of modeling non-linear systems such as this one.

The meaningful parameters of the traffic noise refer to the number of vehicles and flow velocity (traffic parameters), type of pavement and slope of the road (road parameters), and road width and building height (urban parameters). Combining trucks and cars into one single parameter (#cars + 6· #trucks), and averaging the height of the buildings in both sides of the road, we can define a *backpropagation* network (Kung 1993) with 3 inputs and 1 output:

INPUT: (*number of vehicles, average height of buildings, width of the road*)
OUTPUT: (*sound pressure level*)

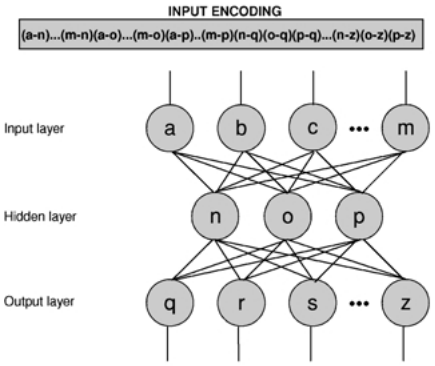
We consider a network with a 3-30-1 topology (input-hidden-output) for learning the set of examples on roads in Messina, Palermo and Catania, in Italy. The neurons have a sigmoid activation function that provides continuous values in the range $[0 \cdots 1]$. This poses a much difficult optimization problem than using boolean activation functions (discrete optimization). We code the weights and bias of the network (151 parameters) in a string of 151 real numbers as indicated by the input-output code of Fig. A.2 left.

The objective function (Fig. A.2 right) consists in finding a network (multilayer perceptron) that learns the pattern set i.e., that minimizes the error between the expected and actual output (supervised learning). This error is summed for all the n output neurons and for all the p patterns of the pattern set. We include this error into a maximization function by subtracting it from a constant (C).

We finally describe an interesting problem generator proposed in De Jong, Potter and Spears (1997) that is also used in the present paper. A problem generator is an easily parameterizable task which has a tunable degree of epistasis, thus allowing to derive very difficult instances at will. Also, using a problem generator removes the opportunity to hand-tune algorithms to a particular problem, therefore allowing a larger fairness when comparing algorithms. In addition, with a problem generator we evaluate our algorithms on a high number of random problem instances, thus increasing the predictive power of the results for the problem class as a whole. Here we utilize their multimodal epistatic generator that we call P-PEAKS.

The idea is to generate a set of P random N-bit strings that represent the location of P peaks in the search space. To evaluate an arbitrary bit string, first locate the nearest peak in Hamming space. Then the fitness of the bit string is the number of bits the string has in common with that nearest peak, divided by N (equation (A.1)). Problems with a small/large number of peaks are weakly/strongly epistatic. Our instance uses $P = 100$ peaks of $N = 100$ bits each, which represents a medium-high epistasis level.

$$f_{P\text{-}PEAKS}(\vec{x}) = \frac{1}{N} \max_{i=1}^{P} \{N - \text{HammingD}(\vec{x}, Peak_i)\} \qquad \text{(A.1)}$$



FITNESS = *C* - ERROR

$$\textbf{ERROR} = \sum_{i=1}^{p} \sum_{j=1}^{n} \left| desired_j - actual_j \right|$$

Fig. A.2. *Encoding the weights of a neural network into a GA string (left) and the fitness function we use (right)*

References

Adamidis P. 1994. Review of parallel genetic algorithms bibliography (v1.0). T.R., Aristotle University of Thessaloniki. available at http://www.control,ee.auth.gr/~panos/papers/pga_review.ps.gz).

Adamidis P. and Petridis V. 1996. Co-operating populations with different evolution behavior. In: Proceedings of the Second IEEE Conference on Evolutionary Computation. pp. 188–191.

Akl S.G. 1992. The Design and Analysis of Parallel Algorithms. Prentice Hall, Englewood Cliffs, NJ.

Alba E., Aldana J.F., and Troya J.M. 1993. Genetic algorithms as heuristics for optimizing ANN design. In: Albrecht R.F., Reeves C.R., and Steele N.C. (Eds.), Artificial Neural Nets and Genetic Algorithms. Springer-Verlag, Berlin, pp. 683–690.

Alba E., Aldana J.F., and Troya J.M. 1995. A genetic algorithm for load balancing in parallel query evaluation for deductive relational databases. In: Pearson D.W., Steele N.C., and Albrecht R.F. (Eds.), Artificial Neural Nets and Genetic Algorithms. Springer-Verlag, Berlin, pp. 479–482.

Alba E., Cotta C., and Troya J.M. 1996. Type-constrained genetic programming for rule-base definition in fuzzy logic controllers. In: Koza J.R., Goldberg D.E., Fogel D.B., and Riolo R.L. (Eds.), Proceedings of the First Annual Conference on Genetic Programming. MIT Press, Cambridge, MA, pp. 255–260.

Alba E., Cotta C., and Troya J.M. 2000. On the importance of the grid shape in 2D spatially structured GAs. Journal of Evolutionary Optimization, to appear.

Alba E. and Troya J.M. 1996. Genetic algorithms for protocol validation. In: Voigt H.M., Ebeling W., Rechenberg I., and Schwefel H.P. (Eds.), Proceedings of the Parallel Problem Solving from Nature IV. Springer-Verlag, Berlin, pp. 870–879.

Alba E. and Troya J.M. 1999. A survey of parallel distributed genetic algorithms. Complexity 4(4): 31–52.

Alba E. and Troya J.M. 2000a. Influence of the migration policy in parallel distributed GAs with structured and panmictic populations. Applied Intelligence 12(3): 163–181.

Alba E. and Troya J.M. 2000b. Cellular evolutionary algorithms: Evaluating the influence of ratio. In: Schoenauer M. *et al*. (Eds.), Proceedings of the Parallel Problem Solving from Nature VI, Springer-Verlag, Berlin, pp. 29–38.

Alba E. and Troya J.M. 2001. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. Future Generation Computer Systems, FGCS 17(4): 451–465.

Alba E. 2002. Parallel evolutionary algorithms can achieve super-linear performance. Information Processing Letters, Elsevier 82(1): 7–13.

Antonisse J. 1989. A new interpretation of schema notion that overturns the binary encoding constraint. In: Schaffer J.D. (Ed.), Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 86–91.

Bäck T. 1996. Evolutionary Algorithms in Theory and Practice. Oxford University Press, London.

Bäck T., Fogel D., and Michalewicz Z. (Eds.) 1997. Handbook of Evolutionary Computation. Oxford University Press, London.

Bäck T., Graaf J.M., Kok J.N., and Kosters W.A. 1997. Theory of genetic algorithms. Bulletin of the European Association for Theoretical Computer Science 63: 161–192.

Bäck T., Hammel U., and Schwefel H.P. 1997. Evolutionary computation: Comments on the history and currrent state. IEEE Transactions on Evolutionary Computation 1(1): 3–17.

Baker, J.E. 1987. Reducing bias and inefficiency in the selection algorithm. In: Grefenstette J.J. (Ed.), Proceedings of the Second International Conference on Genetic Algorithms. Lawrence Erlbaum Associates Publishers, pp. 14–21.

Baluja S. 1993. Structure and performance of fine-grain parallelism in genetic search. In: Forrest S. (Ed.), Proceedings of the Fith International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 155–162.

Belding T.C. 1995. The distributed genetic algorithm revisited. In: Eshelman L.J. (Ed.), Proceedings of the Sixth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 114–121.

Cammarata G., Cavalieri S., Fichera A., and Marletta L. 1993. Noise prediction in urban traffic by a neural approach. In: Mira J., Cabestany J., and Prieto A. (Eds.), Procedings of the International Workshop on Artificial Neural Networks. Springer-Verlag, Berlin, pp. 611–619.

Cantú-Paz E. 1997. A survey of parallel GAs. IlliGAL R. 97003. A revised version of 1995. A summary of research on parallel genetic algorithms. TR#95007.

Cantú-Paz E. and Goldberg D.E. 1997. Predicting speedups of idealized bounding cases of parallel genetic algorithms. In: Bäck T. (Ed.), Proceedings of the Seventh International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 113–120.

Davidor Y. 1991. A naturally occurring niche & species phenomenon: The model and first results. In: Belew R.K. and Booker L.B. (Eds.), Procs. of the 4th ICGA, pp. 257–263.

Deb K. and Spears W.M. 1997. Speciation methods. In: Bäck T., Fogel D.B., and Michalewicz Z. (Eds.), Handbook of Evolutionary Computation. Oxford University Press, London, pp. C6.2: 1-C6.2:5.

De Jong K.A., Potter M.A., and Spears W.M. 1997. Using problem generators to explore the effects of epistasis. In: Bäck T. (Ed.), Proceedings of the Seventh International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 338–345.

De Jong K. and Sarma J. 1995. On decentralizing selection algorithms. In: Eshelman L.J. (Ed.), Proceedings of the Sixth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 17–23.

East I.R. and Mcfarlane D. 1993. Implementation in occam of parallel genetic algorithms on transputer networks. In: Stender J. (Ed.), Parallel Genetic Algorithms. IOS Press, pp. 43–63.

Erickson J.A., Smith R.E., and Goldberg D.E. 1991. SGA-Cube, a simple genetic algorithm for ncube 2 hypercube parallel computers. TCGA Report #91005. The University of Alabama.

Goldberg D.E. 1989a. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, MA.

Goldberg D.E. 1989b. Sizing populations for serial and parallel genetic algorithms. In: Schaffer J.D. (Ed.), Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 70–79.

Goodman E.D. 1996. An introduction to GALOPPS v3.2. TR#96-07-01. GARAGe, I.S. Lab., Dpt. of C.S. and C.C.C.A.E.M., Michigan State University, East Lansing.

Gordon V.S. and Whitley D. 1993. Serial and parallel genetic algorithms as function optimizers. In: Forrest S. (Ed.), Proceedings of the Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 177–183.

Gorges-Schleuter M. 1989. ASPARAGOS an asynchronous parallel genetic optimisation strategy. In: Schaffer J.D. (Ed.), Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 422–427.

Hart W.E., Baden S., Belew R.K., and Kohn S. 1997. Analysis of the numerical effects of parallelism on a parallel genetic algorithm. Proceedings of the Workshop on Solving Combinatorial Optimization Problems in Parallel. IEEE (Ed.), CD-ROM IPPS97.

Herrera F. and Lozano M. 2000. Gradual distributed real-coded genetic algorithms. IEEE Transactions on Evolutionary Computation 4(1): 43–63.

Holland J. 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor.

Kung S.Y. 1993. Digital Neural Networks. Prentice Hall, Englewood Cliffs, NJ.

Levine D. 1994. A parallel genetic algorithm for the set partitioning problem. TR#ANL-94/23. Argonne National Laboratory, Mathematics and Computer Science Division.

Levine D. 1996. Users guide to the PGAPack parallel genetic algorithm library. TR# ANL-95/18.

Levine D. 1997. Genetic algorithms: A practitioner's view. INFORMS Journal of Computing 9(3): 256–259.

Lin S., Punch W.F., and Goodman E.D. 1994. Coarse-grain parallel genetic algorithms: Categorization and new approach. Sixth IEEE SPDP, pp. 28–37.

Maruyama T., Hirose T., and Konagaya A. 1993. A fine-grained parallel genetic algorithm for distributed parallel systems. In: Forrest S. (Ed.), Proceedings of the Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 184–190.

Mejía-Olvera M. and Cantú-Paz E. 1994. DGENESIS-software for the execution of distributed genetic algorithms. In Proceedings of the XX Latinoamerican Conference on Computer Science. Monterrey, México. pp. 935–946.

Menke R. 1997. A revision of the schema theorem. TR#14/97. University of Dortmund.

Michalewicz Z. 1992. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, Berlin.

Mühlenbein H., Schomisch M., and Born J. 1991. The parallel genetic algorithm as function optimizer. Parallel Computing 17: 619–632.

Munetomo M., Takai Y., and Sato Y. 1993. An efficient migration scheme for subpopulation-based asynchronously parallel GAs. TR#HIER-IS-9301. Hokkaido University.

Poli R. 1999. Probabilistic schema theorems without expectation, left-to-right convergence and population sizing in genetic algorithms. TR#CSRP-99-3. University of Birmingham, School of Computer Science.

Potts J.C., Giddens T.D., and Yadav S.B. 1994. The development and evaluation of an improved genetic algorithm based on migration and artificial selection. IEEE Transactions on Systems, Man, and Cybernetics 24(1): 73–86.

Radcliffe N.J. and Surry P.D. 1994. The reproductive plan language RPL2: Motivation, architecture and applications. In: Stender J., Hillebrand E., and Kingdon J. (Eds.), Genetic Algorithms in Optimisation, Simulation and Modelling. IOS Press.

Reeves C.R. (Ed.), 1993. Modern Heuristic Techniques for Combinatorial Problems. Blackwell Scientific Publications, Oxford.

Ribeiro Filho J.L., Alippi C., and Treleaven P. 1993. Genetic algorithm programming environments. In: Stender J. (Ed.), Parallel GAs: Theory & Applications, IOS Press.

Robbins G. 1992. EnGENEer—The evolution of solutions. In Proceedings of the 5th Annual Seminar on Neural Networks and Genetic Algorithms.

Rogers A. and Prügel-Bennett A. 1999. Genetic drift in genetic algorithm selection schemes. IEEE Transactions on Evolutionary Computation 3(4): 298–303.

Sarma J. and De Jong K.A. 1996. An analysis of the effects of neighborhood size and shape on local selection algorithms. In: Voigt H.M., Ebeling W., Rechenberg I., and Schwefel H.P. (Eds.), Parallel Problem Solving from Nature IV. Springer-Verlag, Berlin, pp. 236–244.

Sarma J. and De Jong K. 1997. An analysis of local selection algorithms in a spatially structured evolutionary algorithm. In: Bäck T. (Ed.), Proceedings of the Seventh International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 181–186.

Shonkwiler R. 1993. Parallel genetic algorithms. In: Forrest S. (Ed.), Proceedings of the Fifth ICGA. Morgan Kaufmann, San Mateo, CA, pp. 199–205.

Spiessens P. and Manderick B. 1991. A massively parallel genetic algorithm. In: Belew R.K. and Booker L.B. (Eds.), Proceedings of the 4th International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 279–286.

Stender J. (Ed.), 1993. Parallel Genetic Algorithms: Theory and Applications. IOS Press.

Syswerda G. 1991. A study of reproduction in generational and steady-state genetic algorithms. In: Rawlins G.J.E. (Ed.), Foundations of Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 94–101.

Tanese R. 1989. Distributed genetic algorithms. In: Schaffer J.D. (Ed.), Proceedings of the 3rd International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp. 434–439.

Voigt H.M., Santibáñez-Koref I. and Born J. 1992. Hierarchically structured distributed genetic algorithms. In: Männer R. and Manderick B. (Eds.), Proceedings of the International Conference Parallel Problem Solving from Nature 2. North-Holland, Amsterdam, pp. 155–164.

Whitley D. 1993. A Genetic Algorithm Tutorial. TR#CS-93-103 (revised version). Department of Computer Science, Colorado State University.

Whitley D. and Schaffer J.D. (Eds.), 1992. Proceedings of COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks. IEEE Computer Society Press.

Whitley D. and Starkweather T. 1990. GENITOR II: A distributed genetic algorithm. Journal Expt. Theory Artificial Intelligence 2: 189–214.

Wolpert D.H. and Macready W.G. 1997. No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1(1): 67–82.