An Analysis of Synchronous and Asynchronous Parallel Distributed Genetic Algorithms with Structured and Panmictic Islands

Enrique Alba, José Mª Troya

Dpto. de Lenguajes y Ciencias de la Computación, Univ. de Málaga
Campus de Teatinos (2.2.A.6), 29071- MÁLAGA, España
{eat,troya}@lcc.uma.es

Abstract. In a parallel genetic algorithm (PGA) several communicating nodal GAs evolve in parallel to solve the same problem. PGAs have been traditionally used to extend the power of serial GAs since they often can be tailored to provide a larger efficiency on complex search tasks. This has led to a considerable number of different models and implementations that preclude direct comparisons and knowledge exchange. To fill this gap we begin by providing a common framework for studying PGAs. This allows us to analyze the importance of the synchronism in the migration step of parallel distributed GAs. We will show how this implementation issue affects the evaluation effort as well as the search time and the speedup. In addition, we consider popular evolution schemes of panmictic (steady-state) and structured-population (cellular) GAs for the islands. The evaluated PGAs demonstrate linear and even super-linear speedup when run in a cluster of workstations. They also show important numerical benefits when compared with their sequential counterparts. In addition, we always report lower search times for the asynchronous versions.

1 Introduction

Genetic algorithms (GAs) are stochastic search methods that have been successfully applied in many search, optimization, and machine learning problems [2]. Unlike most other optimization techniques, GAs maintain a population of encoded tentative solutions that are *competitively* manipulated by applying some *variation operators* to find a global optimum.

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

For non-trivial problems this process might require high computational resources, and thus, a variety of algorithmic issues are being studied to design efficient GAs. With this goal in mind, numerous advances are continuously being achieved by designing new operators, hybrid algorithms, and more. We extend one of such improvements consisting in using parallel models of GAs (PGAs).

Several arguments justify our work. First of all, PGAs are naturally prone to parallelism since the operations on the strings can be easily undertaken in parallel. The evidences of a higher efficiency [3], larger diversity maintenance, additional availability of memory/cpu, and multi-solution capabilities, reinforce the importance of the research advances with PGAs [9].

Using a PGA often leads to superior numerical performance and not only to a faster algorithm [5]. However, the truly interesting observation is that the use of a structured population, either in the form of a set of islands [3] or a diffusion grid [11], is the responsible of such numerical benefits. As a consequence, many authors do not use a parallel machine at all to run structured-population models, and still get better results than with serial GAs [5].

In traditional PGA works it is assumed that the model maps directly onto the parallel hardware, thus making no distinction between the *model* and its *implementation*. However, once a structured-population model has been defined it can be implemented in any monoprocessor or parallel machine. This separate vision of model vs. implementation raises several questions. Firstly, any GA can be run in parallel, although a linear speedup is not always possible. In this sense, our contribution is to extend the existing work on distributed GAs (we will use dGA for short) from generational island evolution to steady-state [10], and cellular GAs [11].

Secondly, this suggests the necessity of using a difficult and heterogeneous test suite. In this sense, we use a test suite composed of separable, non-separable, multimodal, deceptive, and epistatic problems (main difficulties of real applications).

Thirdly, the experiments should be replicable to help future extensions of our work. This led us to using a readily available parallel hardware such as a cluster of workstations and to describe the references, parameters, and techniques.

Finally, some questions are open in relation to the physically parallel execution of the models. In our case we study a decisive implementation issue: the synchronism in the communication step [6]. Parameters such as the communication frequency used in a PGA are also analyzed in this paper.

The high number of non-standard or machine-dependent PGAs has led to efficient algorithms in many domains. However, these unstructured approaches often hide their canonical behavior, thus making it difficult to forecast further behavior and to make knowledge exchange. This is why we adopt a unified study of the studied models.

The paper is organized as follows. In Section 2 we present sequential and parallel GAs as especial cases of a more general meta-heuristic. In Section 3 we describe the benchmark used. Section 4 presents a study on the search time of sync/async versions for different basic techniques and problems. Section 5 contains an analysis from the point of view of the speedup, and, finally, Section 6 offers some concluding remarks.

2 A Common Framework for Parallel Genetic Algorithms

In this section we briefly want to show how coarse (cgPGA) and fine grain (fgPGA) PGAs are subclasses of the same kind of parallel GA consisting in a set of communicating sub-algorithms. We propose a change in the nomenclature to call them *distributed* and *cellular* GAs (dGA and cGA), since the grain is usually intended to refer to their computation/communication ratio, while the actual differences can be also found in the way in which they both structure their populations (see Figure 1).

While a distributed GA has large sub-populations (>>1) a cGA has typically one single string in every sub-algorithm. For a dGA the sub-algorithms are loosely connected, while for a cGA they are tightly connected. In addition, in a dGA there exist only a few sub-algorithms, while in a cGA there is a large number of them.

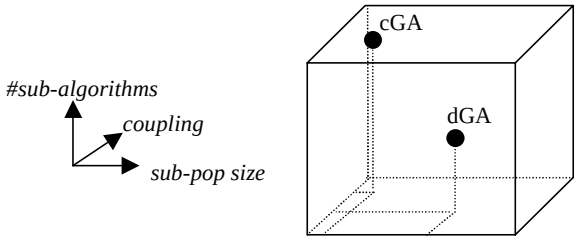


Figure 1. The Structured-Population Genetic Algorithm Cube

This approach to PGAs is *directly* supported in only a few real implementations of PGA software. However, it is very important since we can study the full spectrum of parallel implementations by only considering different coding, operators, and communication details. In fact, this is a natural vision of PGAs that is inspired in the initial work of Holland, in which a *granulated adaptive system* is a set of grains with shared structures. Every grain has its own reproductive plan and internal/external representations for its structures.

The outline of a general PGA is presented in the Algorithm 1. It begins by randomly creating a population $P(t=0)$ of μ structures -strings-, each one encoding the p problem variables, usually as a vector over $I\!B=\{0,1\}$ ($I=I\!B^{p\cdot l_X}$) or $I\!R$ ($I=I\!R^{p}$). An evaluation function Φ is needed to associate a quality real value to every structure.

The stopping criterion ι of the reproductive loop is to fulfill some condition. The solution is identified as the best structure ever found.

The *selection* s_{Θ_s} operator makes copies of every structure from $P(t)$ to $P'(t)$ attending to the fitness values of the structures (some parameters Θ_s might be required). The typical variation operators in GAs are *crossover* ($\otimes$), and *mutation* (m). They both are stochastic techniques whose behavior is governed by a set of parameters, e.g. an application probability: $\Theta_c=\{p_c\}$ -high- and $\Theta_m=\{p_m\}$ -low-.

Finally, each iteration ends by selecting the μ new individuals for the new population (*replacement policy* r). For this purpose the new pool $P'''(t)$ plus a set Q are considered. The best structure usually survives deterministically (*elitism*). Many panmictic variants exist, but two of them are especially popular [10]: the *generational GA* -genGA- ($\lambda=\mu$, $Q=\varnothing$), and the *steady-state GA* -ssGA- ($\lambda=1$, $Q=P(t)$).

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

This communication usually consists in exchanging a set of individuals or population statistics. All the sub-algorithms are though to perform the same reproductive plan. Otherwise the PGA is heterogeneous [2].

In particular, our steady-state panmictic algorithm (Figure 2a) generates one single individual in every iteration. It is inserted back in the population only if it is better (larger fitness than) the worst existing individual.

ALGORITHM 1: PARALLEL GENETIC ALGORITHM

$t := 0$;

initialize: $P(0) := \{\vec{a}_1(0),...,\vec{a}_\mu(0)\} \in I^\mu$;

evaluate: $P(0) : \{\Phi(\vec{a}_1(0)),...,\Phi(\vec{a}_\mu(0))\}$;

while not $\iota(P(t))$ **do** *// Reproductive Loop*

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: $P'''(t) : \{\Phi(\vec{a}'''_1(0)),...,\Phi(\vec{a}'''_\lambda(0))\}$;

replace: $P(t+1) := r_{\Theta_r}(P'''(t) \cup Q)$;

<***communication step***>

$t := t + 1$;

end while

In all the cGAs we use (Figure 2b) a NEWS neighborhood is defined (**N**orth-**E**ast-**W**est-**S**outh in a toroidal grid [11]) in which overlapping demes of 5 strings (4+1) execute the same reproductive plan. In every deme the new string computed after selection, crossover, and mutation replaces the current one only if it is better.

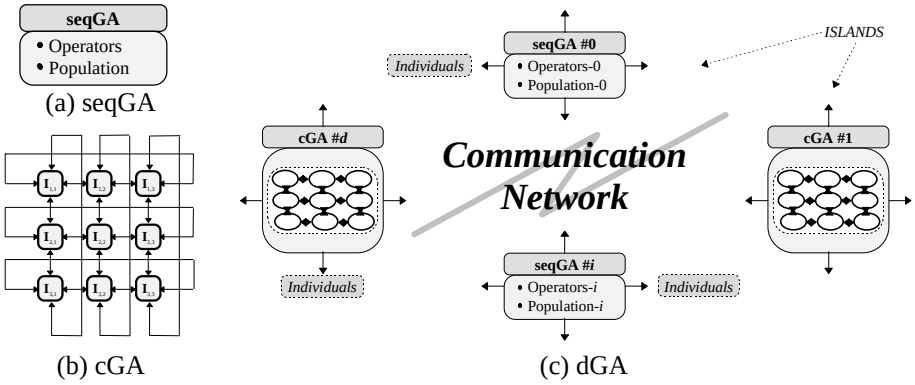


Figure 2. Two basic non-distributed models (a) (b), and a possible -merged!- distribution (c)

In all this paper we deal with MIMD implementations (Figure 2c) of homogeneous dGAs in which migrants are selected *randomly*, and the target island replaces its *worst* string with the incoming one only if it is better. The sub-algorithms are disposed in a unidirectional ring (easy implementation and other advantages).

In all the cases we will study the frequency of migrations in multiples of the global population size: $1 \cdot \mu$, $2 \cdot \mu$, $4 \cdot \mu$, $8 \cdot \mu$, ... The graphs in this paper show these multiples: 1, 2, 4, 8,... A value of 0 means an idle (*partitioned*) distributed evolution. Hence, the mig. frequency depends on the population size, and has not “general recipe” values.

3 The Test Suite

All the problems we test maximize a fitness function. We use the *generalized sphere problem* to help us in understanding the PGA models by defining a baseline for comparisons. The *subset sum problem* is NP-Complete and difficult for the algorithms. Finally, *training a Neural Network* needs an efficient exploration of the search space, final local tuning of the parameters in the strings, and shows high epistasis (weight correlation), and multimodality.

The first problem is an instance of the well-known sphere function (Equation 1). We use 16 variables encoded in 32 bits each one (l=512 bits), and call it SPH16-32.

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

The second optimization task we address is the subset sum problem [7]. It consists in finding a subset of values $V \subseteq W$ from a set of integers $W=\{\omega_1, \omega_2, ..., \omega_n\}$, such that the sum of this subset approaches, without exceeding, a constant C.

We use instances of n=128 integers generated as explained in [7] to increase the instance difficulty, and call it the SSS128 problem.

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

$$f(\vec{x}) = a \cdot (P(\vec{x})) + (1-a) \cdot \lfloor (C - 0{,}1 \cdot P(\vec{x})) \rfloor_0 \tag{2}$$

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

We finally use PGAs for learning the set of weights [1] which make a multilayer perceptron (MLP) to classify a set of patterns correctly. The first MLP we consider computes the parity of 4 binary inputs and it uses three layers of 4-4-1 binary neurons whose weights are encoded as real values in a string for the GA [8]. Every string thus contains l=4·4+4·1+4+1=25 variables (weights plus biases).

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

The second multilayer perceptron is trained to predict the level of urban traffic in a road (one only output) from three inputs: the number of vehicles per hour, the average height of the buildings at both sides of the road, and the road width [4]. The perceptron has a structure of 3-30-1 sigmoid neurons. This yields strings of l=151 real values. For this problem we *also* present results when encoding every weight in 8 bits (strings of l=1208 bit length) to experiment with very long strings. Every function evaluation needs to compute the error along a set of 41 patterns.

All the experiments compare sync/async versions of parallel dGAs (labeled with **s** and **a**, respectively, in the graphs). The experiments use equivalent parameterizations for all the algorithms solving the same problem (same global population, operators and probabilities). In addition, standard operators are used: proportional selection, double point crossover, and bit flip mutation for SPH16-32 and SSS128, and proportional plus random parent selection (to enhance diversity and hyper-plane recombination), uniform crossover, and real additive mutation for the FP-genes.

4 Real Time Analysis of the Impact of the Synchronism

Synchronous and asynchronous parallel dGAs perform both the same algorithm. However, sync islands wait for every incoming string they must accept, while async ones do not. In general, no differences should appear in the search, provided that all the machines are of the same type (our case). On the contrary, the execution time may be modified due to the continuous waits induced in a synchronous model.

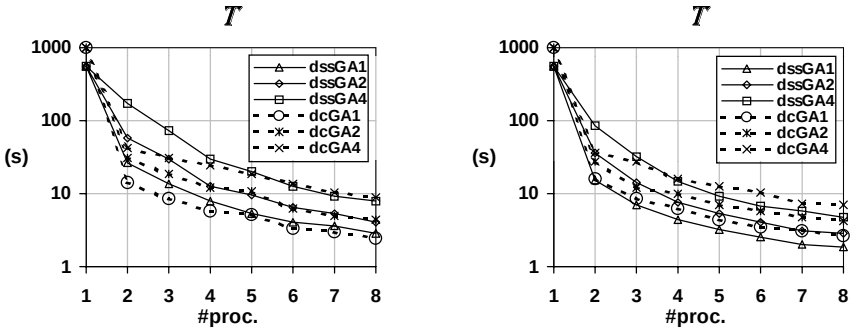


Figure 3. Time in solving SPH16-32 with synchronous (a) and asynchronous versions (b)

In Figure 3 we show the expected execution time (50 independent runs) to solve SPH16-32. The asynchronous models (3b) consistently lessen the search time with respect to their equivalent synchronous algorithms (3a) for any of the tested migration frequencies (1, 2, 4), and island evolution modes (steady-state or cellular).

The effects of the synchronization on the subset sum (Figures 4a and 4b) clearly demonstrate the drawbacks of a tight coupling for non-trivial problems. The sync and async models (100 independent runs) show similar numeric performances for a variety of techniques and processors, thus confirming their similarities.

Sync and async versions needed a similar effort except when migration frequency is 1. Values of 16 and 32 provided a better numeric efficiency and 100% efficacy in all the experiments. Also, it is detected a better resistance of dcGA to bad migration frequencies. For example, for 2 processors and migration frequency 1, dssGA had a 57% of success while dcGA had 100% with a smaller number of evaluations.

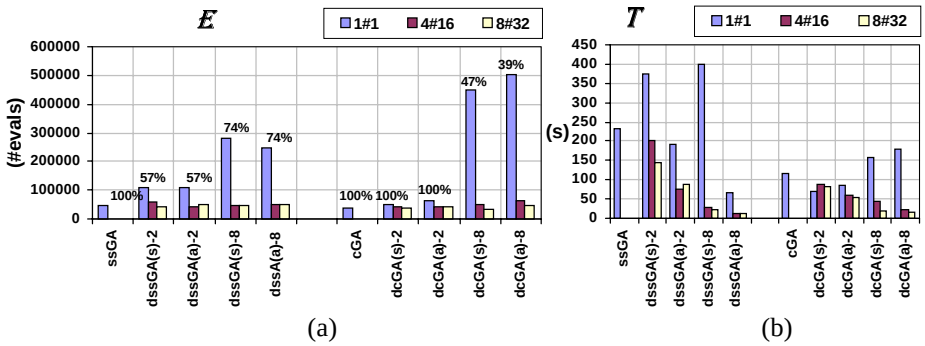


Figure 4. Effects of synchronism on SSS128 (1, 2 and 8 processors): number of evaluations (a), and time to get a solution (b)

In almost all the cases the time to get a solution is consistently reduced by the parallel distributed models. As an exception, we can see a scenario for SSS128: excessive interaction among the parallel islands is slower than the sequential version.

The same holds for the experiments with the "traffic" problem (Figure 5). Only a parallel version (8 processors) is showed since no sequential version was able to solve it. For this network the migration interval has a minor impact on the effort (Figure 5a). The main difficulty is in making the final tuning of the real genes, and in overcoming the lost of functionality in the resulting crossed children strings [1].

The search time of dssGA (Figure 5b) is very good due to its high exploitation. For dcGA, the loosely connected and the partitioned variants are the better ones. This is true since the cGA islands are able of a timely sustained exploitation without important diversity losses, and since they have a relatively faster reproductive loop.

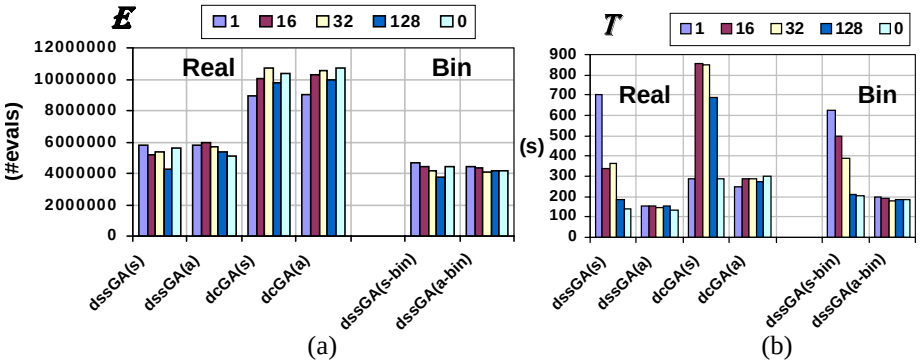


Figure 5. Effects of synchronism on the "traffic" MLP: number of evaluations (a), and time to get a solution (b). We plot dssGA and dcGA with real encoding, and dssGA with a binary code

Notice that, for the "traffic" problem, the search time is also influenced by the encoding (only dssGA is showed): a binary representation is slightly slower than a real encoding due to the longer strings the algorithm has to manage. However, the number of evaluations is somewhat smaller since binary strings induce a pseudo-Boolean search in a discrete space.

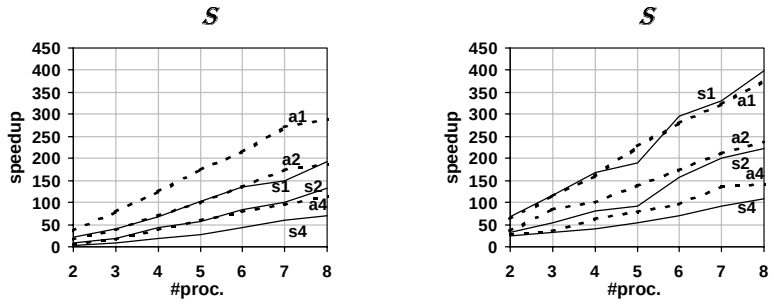
It can be seen that the number of needed evaluations for SSS128 and "traffic" (Figures 4a and 5a) is almost not affected by the synchronization (also for SPH16-32). On the contrary, the search time is considerably lessen by all the async algorithms.

5 Speedup

Before discussing the results on the speedup of the analyzed algorithms, we need to make some considerations. As many authors have established [6], sequential and parallel GAs *must* be compared by running them until a solution of the same quality has been found, *and not* until the same number of steps has been completed. In deterministic algorithms the speedup (Equation 3) is upper bounded by the number of processors n_{proc}, but for a PGA it might not, since it reduces *both* the number of necessary steps *and* the expected execution time $T_{n_{proc}}$ in relation to the sequential one T_1. This means that super-linear speedups are possible [9].

$$\mathcal{S}\left(n_{nproc}\right) = \frac{T_1}{T_{n_{proc}}} \quad (3)$$

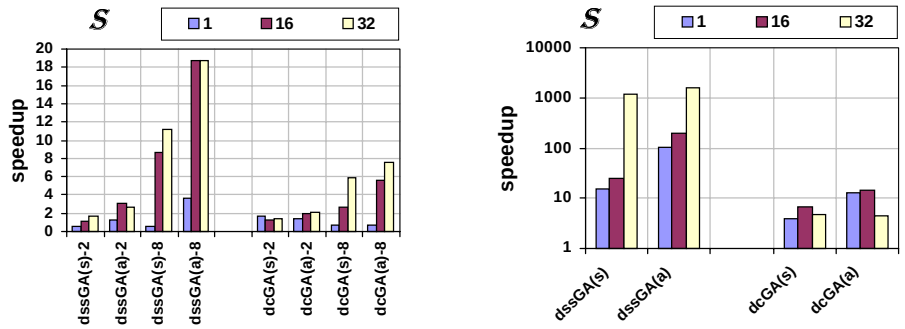
In Figure 6 we show the speedup of dssGA and dcGA in solving SPH16-32 (50 independent runs). Two conclusions can be drawn due to the fact that the problem is GA-easy. On the one hand, the speedup is clearly super-linear. On the other hand, highly coupled islands are better (like using a single population, but faster). This does not hold for other problems, as shown in the previous section (see also [3]). Besides that, the asynchronous versions show definitely better speedup values than the synchronous ones for dssGA (Figure 6a), and slightly better for dcGA (Figure 6b) for equivalent migration frequencies (a1>s1, a2>s2, ...).



(a) (b)

Figure 6. Speedup in solving SPH16-32 with dssGA (a) and dcGA (b)

The speedup for the subset sum problem (Figure 7a) confirms the benefits of a sparse migration (100 independent runs): the speedup is better for larger isolation times (16 and 32). Super-linear speedups for dssGA, and almost linear speedup for dcGAs with migration frequency of 32 have been obtained. More reasonable speedup values are observed due to the difficulty of this problem. As expected, a tight coupling sometimes prevent of having linear speedups.



(a) (b)

Figure 7. Speedup in solving SSS128 with dssGA and dcGA with 2 and 8 processors (a), and speedup in solving the "parity4" training problem with 8 processors (b)

We finally plot in Figure 7b the results in training the "parity4" neural network. Again, the asynchronous models get a larger speedup (and lower absolute execution times). Also, a large isolated evolution is the best global choice.

6 Concluding Remarks

In this paper we have stated the advantages of using parallel distributed GAs and of performing a common analysis of different models. Parallel distributed GAs almost always outperformed sequential GAs. We have distributed both, a sequential steady-state GA, and a cellular GA with the goal of extending the traditional studies.

In all the experiments asynchronous algorithms outperformed their equivalent synchronous counterparts in real time. This confirms other existing results with different PGAs and problems, clearly stating the advantages of the asynchronous communications [6].

Our results take into account the kind of performance analysis proposed also in [6]. After that, we are able to confirm that these algorithms are sometimes capable of showing super-linear speedup, as shown in some other precedent works [3] [9] that did not follow these indications.

As a future work it would be interesting to study (1) the influence of the migration policy, (2) a heterogeneous PGA search, and (3) the importance of the selected island model for a given fitness landscape (generational, steady-state or cellular).

References

1. Alba E., Aldana J. F., Troya J. M.: "Genetic Algorithms as Heuristics for Optimizing ANN Design". In Albrecht R. F., Reeves C. R., Steele N. C. (eds.): *Artificial Neural Nets and Genetic Algorithms*. Springer-Verlag (1993) 683-690
2. Bäck T., Fogel D., Michalewicz Z. (eds.): *Handbook of Evolutionary Computation*. Oxford University Press (1997)
3. Belding T. C.: "The Distributed Genetic Algorithm Revisited". In Eshelman L. J. (ed.): *Proceedings of the Sixth International Conference on Genetic Algorithms*. Morgan Kaufmann, San Francisco, CA (1995) 114-121
4. Cammarata G., Cavalieri S., Fichera A., Marletta L.: "Noise Prediction in Urban Traffic by a Neural Approach". In Mira J., Cabestany J., Prieto A. (eds.): *Proceedings of the International Workshop on Artificial Neural Networks*, Springer-Verlag (1993) 611-619
5. Gordon V. S., Whitley D.: "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 (1993) 177-183
6. Hart W. E., Baden S., Belew R. K., Kohn S.: "Analysis of the Numerical Effects of Parallelism on a Parallel Genetic Algorithm". In IEEE (ed.): CD-ROM IPPS97 (1997)
7. Jelasity M.: "A Wave Analysis of the Subset Sum Problem". In Bäck T. (ed.): *Proceedings of the Seventh International Conference on Genetic Algorithms*. Morgan Kaufmann, San Francisco, CA (1997) 89-96
8. Romaniuk, S. G.: "Evolutionary Growth Perceptrons". In Forrest S. (ed.): *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA (1993) 334-341
9. Shonkwiler R. "Parallel Genetic Algorithms". In Forrest S. (ed.): *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA (1993) 199-205
10. Syswerda G.: "A Study of Reproduction in Generational and Steady-State Genetic Algorithms". In Rawlins G. (ed.): *Foundations of GAs*, Morgan Kaufmann (1991) 94-101
11. Whitley D.: "Cellular Genetic Algorithms". In Forrest S. (ed.): *Proceedings of the Fifth International Conference on GAs*. Morgan Kaufmann, San Mateo, CA (1993) 658