Available online at www.sciencedirect.com

SCIENCE DIRECT®

Parallel Computing 30 (2004) 699–719

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Parallel heterogeneous genetic algorithms for continuous optimization

E. Alba [*], F. Luna, A.J. Nebro, J.M. Troya

Departamento de Lenguajes y Ciencias de la Computación, E.T.S. Ingeniería Informática, Campus de Teatinos, 29071 Málaga, Spain

Received 10 November 2003; accepted 15 December 2003
Available online 18 May 2004

Abstract

In this paper we address the physical parallelization of a very efficient genetic algorithm (GA) known as gradual distributed real-coded GA (GD-RCGA). This search model naturally provides a set of eight subpopulations residing in a cube topology having two faces for promoting exploration and exploitation. The resulting technique has been shown to yield very accurate results in continuous optimization by using crossover operators tuned to explore and exploit the solutions inside each subpopulation. Here, we encompass the actual parallelization of the technique, and get deeper into the importance of running a synchronous versus an asynchronous version of the basic GD-RCGA model. We also present the evaluation of the parallel execution of GD-RCGA over two local area networks, a Fast-Ethernet network and a Myrinet network. Our results indicate that the GD-RCGA model maintains a very high level of accuracy for continuous optimization when run in parallel, and we also demonstrate the relative advantages of each algorithm version over the two networks. Finally, we show that the async parallelization scales better than the sync one, what suggests future research lines for WAN execution and new models of search based on the original two-faced cube.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Parallel genetic algorithms; Heterogeneity; Continuous optimization

[*] Corresponding author.
E-mail addresses: eat@lcc.uma.es (E. Alba), flv@lcc.uma.es (F. Luna), antonio@lcc.uma.es (A.J. Nebro), troya@lcc.uma.es (J.M. Troya).

0167-8191/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2003.12.011

1. Introduction

The goal of this paper is to extend a previous work [1] dealing with a new model for optimization in continuous domains with genetic algorithms (GAs) [2]. GAs are stochastic search techniques that iteratively improve a set of tentative solutions (population of individuals) by applying a crossover operator (merging two or more parents to yield one or more offsprings) and a mutation of their contents (random alterations of the problem variables). However, if we mimic natural evolution, we should not operate on a single population in which a given individual has the potential to mate with any other partner in the entire population (panmixia). Instead, species are structured and tend to reproduce within subgroups or within neighborhoods. Among the existing types of structured GAs, distributed GAs (dGAs) [3] are specially popular. Their premise lies in partitioning the population into several subpopulations, each one being processed by a GA, independently of the others. Furthermore, a sparse migration of individuals produces an exchange of genetic material between the subpopulations that enhances diversity and usually improves the accuracy and efficiency of the algorithm.

Making different decisions on the subalgorithms of a dGA through the application of different search strategies, we obtain the so-called *heterogeneous* dGAs (multiresolution methods). This means that the search occurs at multiple exploration and exploitation levels.

The homo/heterogeneity can be firstly understood as a term referring to the hardware, where each island executes over a different computing platform [4]. However, we have additional levels for possible heterogeneity as regards to the kind of search that the islands are making. At this software level, we can also distinguish various sublevels according to the source of the heterogeneity:

(1) *Parameter level*. The first approach to achieve the software heterogeneity is to use the same GA in each island but varying the parameters of selection, recombination, mutation, and/or migration. These parameters could be initially fixed [5,6], randomly chosen during the evolution [7,8], or follow an adaptive strategy [9–11].
(2) *Operator level*. At this level the heterogeneity is introduced by using different genetic operators on the same GAs [12,13].
(3) *Genotype level*. This is a more subtle kind of heterogeneity where each subpopulation stores locally encoded solutions represented with different enconding schemata [14,15].
(4) *Algorithm level*. This is the most general heterogeneity class at the software level. Each subpopulation can potentially run a different (evolutionary or non-evolutionary) algorithm [16–18].

Note that the algorithm level heterogeneity contains all previous levels, since, for example, a dGA with parameter-based heterogeneity is also algorithm-based heterogeneous. There also exist tools for the production of evolutionary algorithms not directly matching this classification, e.g., by allowing the automatic distribution of the computation, thus facilitating the creation of heterogeneous dGAs [19,20].

Another orthogonal level of heterogeneity can be defined by rapport to the relationship maintained among the subpopulations of the dGA. Basically, if the amount of resources (individuals) of each subpopulation is not fixed during the evolution, i.e., the size of a subpopulation is made dependent on the current success of its strategy, then it can be considered that the subpopulations are competing. Otherwise, it seems that the subpopulations collaborate to find the optimum. Hence, we differentiate between competition-based heterogeneity [21–24] and collaboration-based heterogeneity [25,26].

In this paper we study the parallelization of a heterogeneous dGA called *gradual distributed real-coded GA* (GD-RCGA) [25]. This model of search is a kind of distributed technique that runs eight populations concurrently in a cubic topology with sparse migrations of individuals among them. Distributed evolutionary algorithms are a subclass of decentralized evolutionary algorithms [27] aimed at reducing the convergence to local optima, promoting diversity, and finding alternative solutions to the same problem. The GD-RCGA model is suitable for the optimization of continuous functions, because it includes in the basic improvement loop of the algorithm the utilization of crossover operators for float genes (variables), engineered with fuzzy logic technology to deal with the traditional "fuzzy" GA concepts of exploration and exploitation.

The whole configuration of GD-RCGA is a cube with eight populations, four of them targeted to improve the exploration of the algorithm and another four ones aimed at exploiting the neighborhood of the best solutions. It performs sparse migrations of individuals inside and outside these two sets of four subalgorithms (islands). The GD-RCGA model presents different levels of heterogeneity. On one hand, it is parameter-level heterogeneous since the subpopulations use different values of selection pressure. But its subpopulations also utilize different crossover operators, so it can also be considered as operator-level heterogeneous. On the other hand, the GD-RCGA model exhibits collaboration-based heterogeneity, since its subpopulations cooperate, not compete, in order to perform the search.

There exist some studies on GD-RCGA in the literature. However, although the algorithm shows a straightforward parallelization, only sequential implementations exist; in them, a concurrent execution of the islands is simulated at hand on a monoprocessor. The contribution of this work is, first, to provide a parallel implementation that really runs in a cluster of machines, which has been called *Hy3* (*Hypercube3*). This noun is appropriate since we have future plans for a *Hy4* algorithm. The parallelization should show an improvement in terms of wall-clock time, and therefore should enlarge the complexity of the tasks solved by the algorithm. Furthermore, the execution has been carried out over two local area networks, a Fast-Ethernet network and a Myrinet network. Myrinet is a cost-effective and high-performance technology that is widely used to interconnect clusters of machines. Characteristics that distinguish Myrinet from other networks include full-duplex Gigabit/second data rate links, flow control, and low latency. Additionally, we are interested in investigating the advantages that could outcome from an asynchronous design, instead of the synchronous search that the basic GD-RCGA suggests.

Thus, *Hy3* comes as a new parallel model in which new numerical and efficiency challenges need to be studied.

The paper is organized as follows. Section 2 presents the background to understand the *Hy3* model and a discussion on our parallel implementations. In Section 3, we briefly introduce the problems contained in our benchmark. In Section 4, we proceed to present the parameterization that we have used here, and to analyze the results from a numerical and run time point of view. Finally, we summarize the conclusions and discuss several lines for future research in Section 5.

2. The Hy3 model

In this section we describe the basic behavior of the GD-RCGA [25], and explain how it has been parallelized to yield the new *Hy3* algorithm as a set of concurrent objects using the JACO environment [28] in Java.

2.1. GD-RCGA

The present availability of crossover operators for real-coded genetic algorithms (RCGAs) allows the possibility of including in the same algorithm different exploration or exploitation degrees, which leads to the design of heterogeneous distributed RCGAs based on this kind of operators [12]. GD-RCGA is included into such class of heterogeneous algorithms, since it applies a different crossover operator in each of its component subpopulations. Fig. 1 contains a graphic outline of the algorithm.

The distribution scheme of GD-RCGA is based on a hypercube topology with three dimensions. There are two important faces in this hypercube that have to be considered:

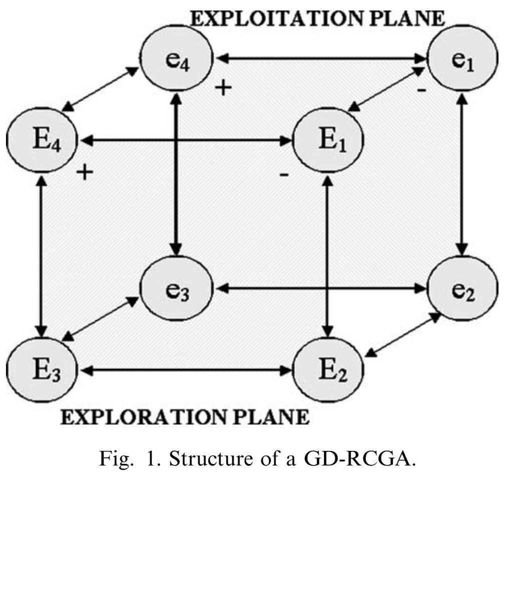

Fig. 1. Structure of a GD-RCGA.

- The *front side* is devoted to exploration. It is made up of four subpopulations $E_1, \ldots, E_4$, in which several exploratory crossovers are applied.
- The *rear side* promotes exploitation. It is composed of subpopulations $e_1, \ldots, e_4$, that apply exploitative crossover operators.

One salient feature of GD-RCGA is the use of an *elitist strategy* [29] in the subpopulations, an important factor that may have influence on a too rapid convergence. However, this is necessary in order to solve complex problems because otherwise the best so far individual could disappear due to crossover or mutation.

The resulting structure is a parallel-suited multi-resolution method using several crossover operators which allow to achieve simultaneously a diversified search (reliability), and an effective local tuning (accuracy). Furthermore, subpopulations are adequately connected for exploiting the multi-resolution in a *gradual* way, since the migrations between subpopulations belonging to different categories (front-rear migrations) may induce the refinement/expansion of the best emerging zones.

Let us explain the migration schema and the selection mechanism used in GD-RCGA for establishing a correct coordination between refinement and expansion.

2.1.1. Migration schema

Distributed genetic algorithms behavior is strongly determined by the migration mechanism [30–32]. GD-RCGA uses a migration model where copies of migrants are sent only towards immediate neighbors along a dimension of the hypercube, and each subsequent migration takes place along a different dimension of the hypercube. Particularly, the best element of each population is sent towards the corresponding subpopulation periodically, as shown in Fig. 2. The sequence of application is, first, the refinement migrations; second, the refinement/expansion migrations; third, the expansion migrations; and finally, the sequence starts again. The place of an emigrant is taken by the incoming individual.

This migration schema keeps a global elitist strategy, since the best element of all subpopulations is never lost, although it could be moved from one subpopulation to another.

2.1.2. Selection mechanism

We use the same selection mechanism as in [25]: *linear ranking selection* [33]. It is used because its selective pressure can be easily adjusted. In linear ranking selection,

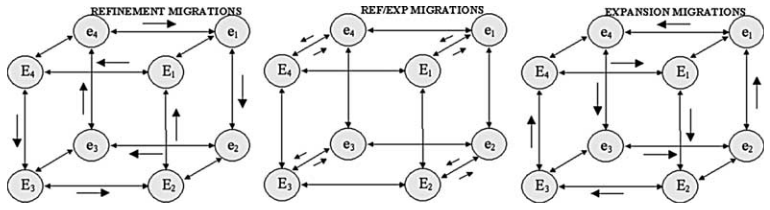

Fig. 2. Three types of migration in a GD-RCGA.

the individuals are sorted in order of raw fitness, and then the selection probability, p_s, of each individual I_i is computed according to its rank $\mathrm{rank}(I_i)$, with $\mathrm{rank}(I_{\mathrm{best}}) = 1$, by using the following non-increasing assignment function:

$$p_s(I_i) = \frac{1}{N} \cdot \left(\eta_{\max} - (\eta_{\max} - \eta_{\min}) \cdot \frac{\mathrm{rank}(I_i) - 1}{N - 1} \right), \tag{1}$$

where N is the population size, and $\eta_{\min} \in [0, 1]$ specifies the expected number of copies for the worst individual (the best one has $\eta_{\max} = 2 - \eta_{\min}$ expected copies). The selection pressure of linear ranking is determined by $\eta_{\min}$. If $\eta_{\min}$ is low, high pressure is achieved, whereas if it is high, the pressure is low. Different selection pressure degrees have been assigned to every subpopulation of the GD-RCGAs, by using the $\eta_{\min}$ values shown in Table 1.

Linear ranking is combined with *stochastic universal sampling* [34]. This procedure guarantees that the number of copies of any individual is bounded by the floor and ceiling of its expected number of copies.

The GD-RCGA has been implemented endowed with *fuzzy connective-based* crossover operators, called GD-FCB [25]. FCB-crossover operators have different exploration/exploitation degrees (Table 1), which allow us to produce gradual effects by configuring them correctly.

2.2. GD-RCGA parallelization using JACO: the Hy3 model

The parallelization of the GD-RCGA, that we have called *Hy3*, has been carried out by using the software components shown in Fig. 3. These components are the JACO runtime system and the *Hy3* Java design, which are discussed in the next two subsections.

2.2.1. The JACO runtime system

JACO (JAva-based Concurrent Object system) is a runtime system implemented in Java, although it has been also rewritten in C# on top of the .NET platform of Microsoft [28].

With JACO we can use Java to fast writing parallel programs according to a concurrent object model (high abstraction). This model considers a concurrent object as an active entity, with an internal state and a public interface. Objects can also have synchronization constraints, which disable some operations when they are not allowed. A typical example is disabling a `put` operation on a bounded buffer when

Table 1
Crossover exploration/exploitation degrees and $\eta_{\min}$ values for each subpopulation

	Exploitation				Exploration			
	e_4	e_3	e_2	e_1	E_1	E_2	E_3	E_4
Crossover	+		←	–	–		→	+
$\eta_{\min}$	0.8	0.7	0.6	0.5	0.3	0.2	0.1	0.0

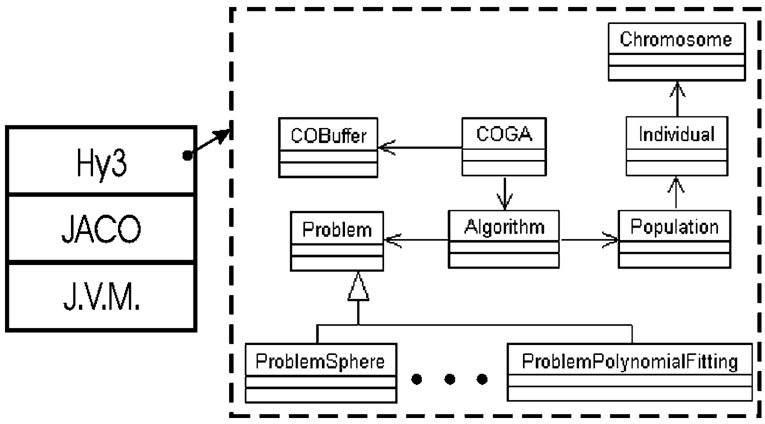

Fig. 3. Multilevel software architecture used to implement *Hy3*.

it is full. There are two kinds of operations: *commands*, which are asynchronous operations, and *queries*, which are read-only synchronous operations. To ensure that a number of operations are executed in mutual exclusion, objects need to be acquired before being used. There also are two types of acquire operations: exclusive and shared. Thus, concurrent objects are accessed according to a multiple-readers/single-writer scheme.

2.2.2. The Hy3 model

The UML class diagram in Fig. 3 depicts the design used in the *Hy3* model. It presents two kinds of objects: concurrent objects, which are supported by JACO and prefixed with *CO-*, and sequential objects, without prefixes.

The *COGA* class represents the islands of the distributed genetic algorithm. It manages the program execution by controlling the termination, the computation of new algorithm steps, and the migration schema. The inclusion of *COBuffer* class has been necessary to avoid deadlocks (uncoupling buffers). Each *COGA* has an associated *COBuffer* located in the same node, from which it takes migrated individuals. The rest of the classes are devoted to the computation of the problem, implementing selection, crossover, mutation, and evaluation of individuals.

Two versions of *Hy3* have been implemented: *Synchronous Hy3* and *Asynchronous Hy3*. In the first one, every subpopulation, for each migration phase, sends its best individual and then it waits for another one coming from the corresponding neighbor. In the async mode, this consideration is not taken into account; thus, any individual stored in its buffer can be included in the population at any time. Since the incoming individual always replaces the best one in the subpopulation in the two modes, the async version could lead to lose the global elitism due to its asynchronism (the best individual of a subpopulation could be replaced before it is migrated) although it is not an usual scenario on a homogeneous cluster of machines.

3. Problems

In this section, we present the benchmark used to test our algorithms. We have analyzed the results of minimization experiments on six test functions and three real-world problems in order to better sustain our claims. We selected the same benchmark that [25] because it is very complete and also for comparison purposes. They are described in Sections 3.1 and 3.2, respectively.

3.1. Test functions

We have considered six classical and well-known test functions: *sphere* model (f_{Sph}) [29,35], *generalized Rosenbrock's* function (f_{Ros}) [29], *Schwefel's problem 1.2* (f_{Sch}) [35], *generalized Rastrigin's* function (f_{Ras}) [36,37], *Griewangk's* function (f_{Gri}) [38], and *expansion of f10* (ef_{10}) [39]. Fig. 4 shows their formulation. The dimension of the search space is 10 for ef_{10} and 25 for the remaining test functions. Each one has its particular features:

- f_{Sph} is a continuous, strictly convex, and unimodal function.
- f_{Ros} is a continuous, non-separable (nonlinear interactions among variables), and unimodal function, with the optimum located in a steep parabolic valley with a flat bottom (i.e., hard progress to the optimum).
- f_{Sch} is a continuous and unimodal function. Its difficulty concerns the fact that searching along the coordinate axes only gives a poor rate of convergence because the gradient of f_{Sch} is not oriented along the axes. It presents similar difficulties to f_{Ros}, but its valley is much narrower.
- f_{Ras} is a scalable, continuous, and multimodal function, which is made from f_{Sph} by modulating it with $a \cdot \cos(\omega \cdot x_i)$.
- f_{Gri} is a continuous and multimodal function. This function is difficult to optimize because it is non-separable.
- f_{10} is a function that has nonlinear interactions between two variables. Its expanded version ef_{10} is built in such a way that it induces nonlinear interactions across multiple variables. It is non-separable as well.

3.2. Real-world problems

We have chosen the following three real-world problems: *systems of linear equations* [40], *frequency modulation sounds parameter identification problem* [18], and *polynomial fitting problem* [41]. They all are described in next subsections.

3.2.1. Systems of linear equations

The problem may be stated as solving for the elements of a vector X, given the matrix A and the vector B in the expression $A \cdot X = B$. The evaluation function used for these experiments is

f_{Sph}	$f_{Sph}(\vec{x}) = \sum_{i=1}^{n} x_i^2$ $-5.12 \leq x_i \leq 5.12$ $f_{Sph}(x^*) = 0$
f_{Ros}	$f_{Ros}(\vec{x}) = \sum_{i=1}^{n-1}(100 \cdot (x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$ $-5.12 \leq x_i \leq 5.12$ $f_{Ros}(x^*) = 0$
f_{Sch}	$f_{Sch}(\vec{x}) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j\right)^2$ $-65.536 \leq x_i \leq 65.536$ $f_{Sch}(x^*) = 0$
f_{Ras}	$f_{Ras}(\vec{x}) = a \cdot n + \sum_{i=1}^{n}(x_i^2 - a \cdot \cos(\omega \cdot x_i))$ $a = 10, \omega = 2\pi$ $-5.12 \leq x_i \leq 5.12$ $f_{Ras}(x^*) = 0$
f_{Gri}	$f_{Gri}(\vec{x}) = 1 + \frac{1}{d}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$ $d = 4000$ $-600.0 \leq x_i \leq 600.0$ $f_{Gri}(x^*) = 0$
ef_{10}	$ef_{10}(\vec{x}) = f_{10}(x_n, x_1) + \sum_{i=1}^{n-1} f_{10}(x_i, x_{i+1})$ $f_{10}(x,y) = (x^2 + y^2)^{0.25} \cdot \left(\sin^2(50 \cdot (x^2 + y^2)^{0.1}) + 1\right)$ $x, y \in (-100, +100]$ $ef_{10}(x^*) = 0$

Fig. 4. Test functions.

$$f_{\mathrm{sle}}(x_1, \ldots, x_n) = \left| \sum_{i=1}^{n} \left(\sum_{j=1}^{n} (a_{ij} \cdot x_j) - b_i \right) \right|. \tag{2}$$

Clearly, if the system of equations is solvable, the best value for this objective function is $f_{\mathrm{sle}}(x^*) = 0$. Furthermore, the range of parameters is $[-9.0, +11.0]$. Inter-parameter linkage (i.e., nonlinearity) is easily controlled in systems of linear equations, their nonlinearity does not deteriorate as increasing the number of parameters used, and they have proven to be quite difficult. We have considered a ten-parameter problem instance. Its matrices are included in Fig. 5.

3.2.2. Frequency modulation sounds parameter identification problem

The problem is to specify six parameters a_1, ω_1, a_2, ω_2, a_3, ω_3 of the frequency modulation sounds model represented by

$$y(t) = a_1 \cdot \sin(\omega_1 \cdot t \cdot \theta + a_2 \cdot \sin(\omega_2 \cdot t \cdot \theta + a_3 \cdot \sin(\omega_3 \cdot t \cdot \theta))) \tag{3}$$

with $\theta = (2 \cdot \pi/100)$. The fitness function is defined as the summation of the square errors between the evolved data and the model data, as follows:

$$f_{\text{fms}}(a_1, \omega_1, a_2, \omega_2, a_3, \omega_3) = \sum_{t=0}^{100} (y(t) - y_0(t))^2, \tag{4}$$

where the model data are given by the following equation:

$$y_0(t) = 1.0 \cdot \sin(5.0 \cdot t \cdot \theta - 1.5 \cdot \sin(4.8 \cdot t \cdot \theta + 2.0 \cdot \sin(4.9 \cdot t \cdot \theta))). \tag{5}$$

Each parameter is in the range $[-6.4, +6.35]$. This is a highly complex multimodal problem having strong epistasis, with minimum value $f_{\text{fms}}(x^*) = 0$.

$$\begin{bmatrix} 5&4&5&2&9&5&4&2&3&1 \\ 9&7&1&1&7&2&2&6&6&9 \\ 3&1&8&6&9&7&4&2&1&6 \\ 8&3&7&3&7&5&3&9&9&5 \\ 9&5&1&6&3&4&2&3&3&9 \\ 1&2&3&1&7&6&6&3&3&3 \\ 1&5&7&8&1&4&7&8&4&8 \\ 9&3&8&6&3&4&7&1&8&1 \\ 8&2&8&5&3&8&7&2&7&5 \\ 2&1&2&2&9&8&7&4&4&1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 40 \\ 50 \\ 47 \\ 59 \\ 45 \\ 35 \\ 53 \\ 50 \\ 55 \\ 40 \end{bmatrix}$$

Fig. 5. Matrices of the linear equations problem instance.

3.2.3. Polynomial fitting problem

This problem lies in finding the coefficients of the following polynomial in z:

$$P(z) = \sum_{j=0}^{2k} c_j \cdot z^j, \quad k \in \mathbf{Z}^+ \tag{6}$$

such that, $\forall z \in [-1, +1]$

$$P(z) \in [-1, +1], \quad P(1.2) \geqslant T_{2k}(1.2), \quad P(-1.2) \geqslant T_{2k}(-1.2), \tag{7}$$

where $T_{2k}(z)$ is a Chebyshev polynomial of degree $2k$.

The solution to the polynomial fitting problem consists of the coefficients of $T_{2k}(z)$. This polynomial oscillates between -1 and $+1$ when its argument z is between -1 and $+1$. Outside this region, the polynomial rises steeply in the direction of high positive ordinate values. This problem has its roots in electronic filter design, and it challenges an optimization procedure by forcing it to find parameter values with grossly different magnitudes, something very common in industrial systems. The Chebyshev polynomial employed here is

$$T_8(z) = 1 - 32 \cdot z^2 + 160 \cdot z^4 - 256 \cdot z^6 + 128 \cdot z^8. \tag{8}$$

It is a nine-parameter problem. A small correction is needed in order to transform the constraints of this problem into an objective function to be minimized, called f_{Cheb} (see [25] for the details). Each parameter (coefficient) is in the range $[-512.0, +512.0]$. The objective function value of the optimum is $f_{\text{Cheb}}(c^*) = 0$.

4. Experiments

In this section, we present the parameters used in the *Hy3* model (Section 4.1). Then we discuss the experimental results obtained (Section 4.2) and perform a further analysis of the importance of the polling frequency in the asynchronous *Hy3* model (Section 4.3).

4.1. Hy3 parameters

All our *Hy3* models use 20 individuals per subpopulation and perform a migration every five generations. The probability of update an individual by mutation (p_{m}) is 0.125, and the crossover probability is 0.6. The mutation operator applied is *non-uniform* mutation [42]. This operator needs two parameters: b (set to value 5), which determines the degree of dependency on the number of iterations, and T, which is the maximum number of generations.

The original GD-RCGA work imposed a predefined number of iterations (5000), but we cannot do the same because we want to measure the time to find equivalent solutions between the sync/async versions and with respect to the original work. Thus, we have defined our goal as to reach the fitness values appearing in Table 2 (that correspond to the average of the best fitness function found in [25]). All the

Table 2
Target fitness (TF) for the *Hy3* models

Problem	TF
f_{Sph}	2e−13
f_{Ras}	4e−11
f_{sle}	4e1
f_{Ros}	9e0
f_{Gri}	2e−2
f_{fms}	1e1
f_{Sch}	4e0
ef_{10}	2e−3
f_{Cheb}	2e2

tests perform 100 independent runs. The presented results are the average over successful executions, i.e., executions that reach the target fitness.

Our computing system is a cluster of eight personal computers running SuSE Linux 8.1 (i386)—Kernel 2.4.19-4GB (1), each one having an Intel Pentium IV 2.4 GHz processor and 512 Mb of memory. The machines are interconnected by a Fast-Ethernet (100 Mbps) network and a Myrinet (2 Gbps) network. We have used JDK 1.4.1_01-b01 and compiled the programs with the *-O* optimization flag.

4.2. Results

Let us now proceed with the analysis of the results. The execution times of synchronous and asynchronous versions of *Hy3* over Fast-Ethernet and Myrinet are presented in Table 3. Fig. 6 also shows the difference, $\Delta_{\bar{t}}$, between async and sync execution times ($\Delta_{\bar{t}} = \bar{t}_{\mathrm{Async}} - \bar{t}_{\mathrm{Sync}}$). In this figure, positive values mean that the asynchronous *Hy3* is slower than the synchronous one, while negative values show that the asynchronous model is faster than the sync one.

We consider first the monoprocessor case. We can observe that the synchronous algorithm has produced a faster execution than the asynchronous one (positive val-

Table 3
Execution times of *Hy3*

Time (ms)	1 CPU			8 CPUs					
	Sync	Async	p	Fast-Ethernet			Myrinet		
				Sync	Async	p	Sync	Async	p
f_{Sph}	171050	189203	+	105182	57428	+	86389	77143	+
f_{Ros}	20492	18608	−	12451	6952	+	11431	6675	+
f_{Sch}	29359	30207	−	14742	7956	+	12341	12061	−
f_{Ras}	182252	190949	+	97606	56130	+	87439	80878	+
f_{Gri}	57313	65460	−	27970	21237	+	28222	26080	−
ef_{10}	199749	217838	+	112793	63584	+	99481	86743	+
f_{sle}	1142	1129	+	1081	1051	+	1002	1025	+
f_{fms}	23167	21840	−	9219	4487	+	7793	4529	+
f_{Cheb}	164717	228855	−	20371	24059	−	13658	27810	+

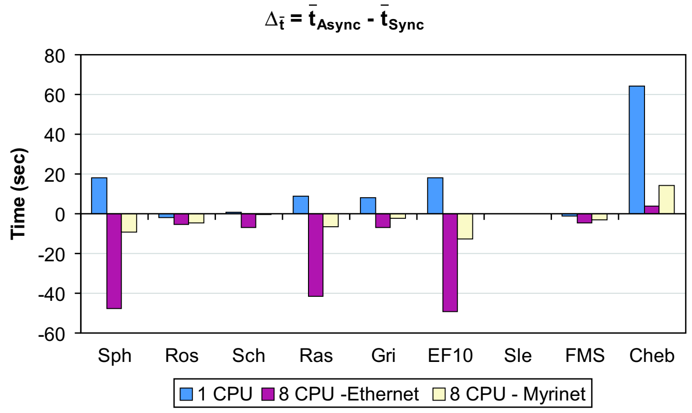

Fig. 6. Difference between asynchronous and synchronous times.

ues in first columns of Fig. 6), and there exists statistical confidence (see "+" symbols meaning significance of t-test) for three problems. The complex instance f_{sle} is an exception, although the difference is negligible. These results can be explained because of the extra computation carried out by the subalgorithms in the asynchronous *Hy3*. This extra computation is provoked by the polling operation used to check for incoming individuals from other subpopulations.

But if we observe the right part of Table 3, we can notice that when running the algorithm in eight CPUs, using Fast-Ethernet as well as Myrinet, the asynchronous version is usually faster than the sync one (negative values in Fig. 6). The reason is that the synchronization induces larger parallel waiting times. We conclude that the run times provided by the asynchronous parallel *Hy3* model are in general lower than the synchronous ones, since there exists a large plethora of possible fine-tuned more efficient parallel asynchronous configurations while there exists a single sequential or parallel synchronous one.

Table 4 allows us to measure the performance of parallel *Hy3* executions over Fast-Ethernet and Myrinet. In this table we present two metrics, the parallel efficiency (η) and the serial fraction (sf) [43], in order to enrich our understanding of the effects of parallelism on the *Hy3* models. If we consider that N is the number of processors ($N = 8$ in this case) and s_N is the speedup ($s_N = \bar{t}_{1\mathrm{CPU}}/\bar{t}_{N\mathrm{CPUs}}$), the two metrics can be defined, respectively, as

$$\eta = \frac{s_N}{N} = \frac{\frac{\bar{t}_{1\,\mathrm{CPU}}}{\bar{t}_{N\,\mathrm{CPUs}}}}{N} \tag{9}$$

$$\mathrm{sf} = \frac{1/s_N - 1/N}{1 - 1/N}. \tag{10}$$

One can notice that, first, the async parallel efficiency is greater than the sync one, whatever network we use. It is also observed the globally poor efficiency of the parallel *Hy3*. This undesirable feature of the parallelization can be explained because

Table 4
Parallel efficiency of *Hy3*

	Parallel efficiency				Serial fraction			
	$\eta^{\text{Eth}}_{\text{Sync}}$	$\eta^{\text{Eth}}_{\text{Async}}$	$\eta^{\text{Myr}}_{\text{Sync}}$	$\eta^{\text{Myr}}_{\text{Async}}$	$\text{sf}^{\text{Eth}}_{\text{Sync}}$	$\text{sf}^{\text{Eth}}_{\text{Async}}$	$\text{sf}^{\text{Myr}}_{\text{Sync}}$	$\text{sf}^{\text{Myr}}_{\text{Async}}$
f_{Sph}	0.20	0.41	0.25	0.31	0.56	0.20	0.43	0.32
f_{Ros}	0.21	0.33	0.22	0.35	0.55	0.28	0.49	0.27
f_{Sch}	0.25	0.47	0.30	0.31	0.43	0.16	0.34	0.31
f_{Ras}	0.23	0.43	0.26	0.30	0.47	0.19	0.41	0.34
f_{Gri}	0.26	0.39	0.25	0.31	0.42	0.23	0.42	0.31
ef_{10}	0.22	0.43	0.25	0.31	0.50	0.19	0.43	0.31
f_{sle}	0.13	0.13	0.14	0.14	0.94	0.92	0.86	0.89
f_{fms}	0.31	0.61	0.37	0.60	0.32	0.09	0.24	0.09
f_{Cheb}	1.01	1.19	1.51	1.03	−0.001	−0.02	−0.05	−0.003

the original GD-RCGA model is targeted to run on a monoprocessor (each island contains only 20 individuals). However, we want to remark that this metric grows with problem complexity (leading to 100% efficiency, even super-linear speedup), as it is justified by the increase of the local computation/communication ratio introduced by complex instances (f_{Cheb}).

The high values of serial fraction point out that the parallel algorithm can be still further improved to better profit from the parallel platform. However, acceptable values are obtained in functions f_{fms}, where $\text{sf}^{\text{Eth}}_{\text{Async}} = \text{sf}^{\text{Myr}}_{\text{Async}} = 0.09$, and f_{Cheb}, where $\text{sf} < 0$. This last negative value indicates that the parallel algorithm has achieved super-linear speedup [44]. The serial fraction also shows that the async models are more suitable for parallel execution than the sync ones, since in general $\text{sf}_{\text{Async}} < \text{sf}_{\text{Sync}}$. Additionally, if this metric remains constant for a different number of processors, it could allow us to consider as "good" results those with small values of parallel efficiency, because the loss of efficiency is due to the limited parallelism of the model (although we can conclude nothing in this sense here, since *Hy3* runs on a fixed number of processors).

Now, in Table 5 we turn to the analysis of the number of evaluations (the numerical effort to solve problems). We also present in Fig. 7 the difference, $\Delta_{\bar{e}}$, between async and sync number of evaluations ($\Delta_{\bar{e}} = \bar{e}_{\text{Async}} - \bar{e}_{\text{Sync}}$). In this case, positive values indicate that the async model performs a greater number of evaluations than the sync one, while negative values show that the number of evaluations accomplished by the sync *Hy3* is greater. The two versions, sync and async, need a similar effort to solve all the optimization tasks, what it was expected, since all the machines have an identical computational power, thus inducing similar behavior for the sync and async migrations. Despite the positive values in Fig. 7 ($\bar{e}_{\text{Async}} > \bar{e}_{\text{Sync}}$), neither the sync *Hy3* model nor the async *Hy3* are more efficient numerically ("–" symbol in statistics of Table 5), i.e., they two require a similar effort to locate a solution.

In order to evaluate the parallel execution of *Hy3* over the two local area networks, we have performed some additional significance tests, that do not appear in this paper to reduce the length of the section. These tests consist in comparing, on the one hand, the synchronous parallel versions of the algorithm running over

Table 5
Number of evaluations of *Hy3*

Time (ms)	1 CPU			8 CPUs					
	Sync	Async	p	Fast-Ethernet			Myrinet		
				Sync	Async	p	Sync	Async	p
f_{Sph}	661421	720684	+	691239	723302	+	668086	681971	–
f_{Ros}	74058	66339	–	75885	75903	–	81289	52363	+
f_{Sch}	108975	110664	–	92308	89542	–	88240	99888	–
f_{Ras}	693733	713550	–	645777	701516	+	666534	707829	+
f_{Gri}	215380	240381	–	181614	258359	+	208478	225343	–
ef_{10}	724190	767917	–	728343	774928	+	747161	742087	–
f_{sle}	320	320	–	320	320	–	320	321	–
f_{fms}	51613	45373	–	54431	29962	+	47449	28623	+
f_{Cheb}	153756	171836	–	93654	111138	–	60336	129094	+

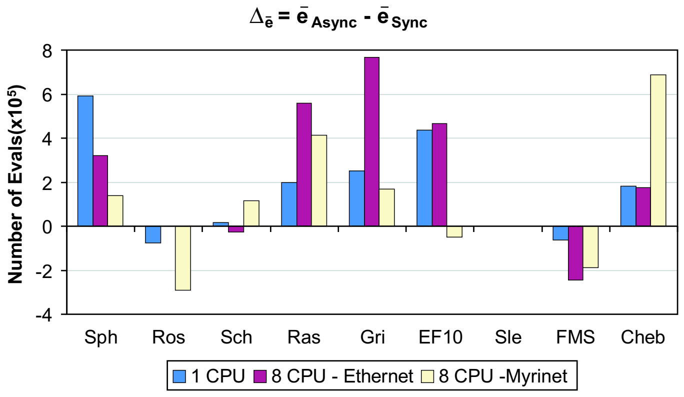

Fig. 7. Difference between asynchronous and synchronous number of evaluations.

the two networks and, on the other hand, the asynchronous ones. Again, the results of the significance tests reveal that these parallel executions over the two networks are similar. This can be explained because the amount of information transmitted is small (only an individual per migration) and therefore the *Hy3* model does not take advantage of the higher transfer rates of Myrinet network.

Finally, in Table 6 we show the percentage of successful executions of the *Hy3* models. The results presented do not allow us to conclude anything about the superiority of any of them for our varied benchmark. The sync version has a larger hit rate than the async one for the problems f_{Ros}, f_{Gri}, and f_{Cheb}, while the async *Hy3* performs better for f_{Sph}, f_{Ras}, and ef_{10}. For the rest of the problems, either both models always reach the target fitness (f_{Sch} and f_{sle}) or the behavior is quite different between the monoprocessor case and the parallel case (f_{fms}). The suitability is thus clearly problem-dependent.

Table 6
Hit rate of *Hy3*

	1 CPU		8 CPUs			
	Sync	Async	Fast-Ethernet		Myrinet	
			Sync	Async	Sync	Async
f_{Sph}	0.79	1.00	0.70	1.00	0.71	0.99
f_{Ros}	0.90	0.81	0.91	0.85	0.93	0.83
f_{Sch}	1.00	1.00	1.00	1.00	1.00	1.00
f_{Ras}	0.80	0.97	0.90	0.96	0.79	0.97
f_{Gri}	0.85	0.73	0.88	0.78	0.82	0.77
ef_{10}	0.82	0.95	0.86	0.94	0.83	1.00
f_{sle}	1.00	1.00	1.00	1.00	1.00	1.00
f_{fms}	0.36	0.30	0.35	0.49	0.38	0.46
f_{Cheb}	0.50	0.21	0.76	0.40	0.81	0.28

4.3. Influence of the polling frequency in the asynchronous Hy3

As we stated before, the subpopulations in the asynchronous *Hy3* need to perform a polling operation to check its buffer for incoming individuals. This polling operation is carried out on every iteration of the algorithm, i.e., if we define the polling gap (PG) as the number of iterations between polling operations, then it can be said that $\mathrm{PG} = 1$. This high polling frequency (note that it is the maximum possible value) leads the async *Hy3* to do a high extra computation. In this section we want to analyze the behavior of the asynchronous models when we increase PG (i.e., decrease the polling frequency). Particularly, we execute the async *Hy3* with $\mathrm{PG} = 5$. In order to differentiate both models, we have called them async^1 *Hy3* and async^5 *Hy3*, respectively.

The first results that we present in this section (Table 7) concern the successful runs of the async^1 and async^5 models. They show that, for functions where the algo-

Table 7
Hit rate of Async^5 *Hy3* model

	1 CPU		8 CPUs			
	Async^1	Async^5	Fast-Ethernet		Myrinet	
			Async^1	Async^5	Async^1	Async^5
f_{Sph}	1.00	0.00	1.00	0.82	0.99	0.00
f_{Ros}	0.81	0.97	0.85	0.99	0.83	0.99
f_{Sch}	1.00	1.00	1.00	1.00	1.00	1.00
f_{Ras}	0.97	0.00	0.96	0.00	0.97	0.00
f_{Gri}	0.73	1.00	0.78	1.00	0.77	0.98
ef_{10}	0.95	0.00	0.94	0.02	1.00	0.26
f_{sle}	1.00	1.00	1.00	1.00	1.00	1.00
f_{fms}	0.30	0.56	0.49	0.67	0.46	0.57
f_{Cheb}	0.21	0.48	0.40	0.86	0.28	0.80

rithm has to achieve high accuracy (f_{Sph}, f_{Ras}, and ef_{10}) the async^5 *Hy3* either never find the optimum or the hit rate is small. We must remark that this behavior does not mean that the algorithm never finds the solution, but the conditions imposed by these problems are very restrictive. For example, the target fitness for f_{Ras} is 4e−11 (see Table 2) and all the algorithms find easily 1e−10, but it is very difficult to progress until 4e−11. Another subtle reason that can explain this small number of successful runs is because of the implementation of the polling gap. If a polling operation finds an empty buffer due to the asynchronism of the algorithm, then the subpopulation does not incorporate any individual at least in PG + PG (10) complete generations, and therefore a new individual may be grossly incompatible with the target subpopulation [15,25].

For all of the problems but f_{Sph}, f_{Ras}, and ef_{10}, the performance of the async^5 always showed a higher or equal hit rate than the async^1 model. This can be explained because of the incorporation of incompatible individuals by migration (the mule effect): this leads the async^5 *Hy3* to evolve subpopulations independently during a larger number of generations (more exploitation); this feature, along with the low restrictions of this functions (high target fitness), allows the algorithm to rapidly find acceptable solutions that fulfil the demanding target value.

Let us now begin with the run time and numerical effort analyses. Fig. 8 shows the difference, $\Delta_{\bar{t}}$, between the execution times of async^1 and async^5 models ($\Delta_{\bar{t}} = \bar{t}_{\mathrm{Async}^1} - \bar{t}_{\mathrm{Async}^5}$). Analogously, Fig. 9 presents the difference, $\Delta_{\bar{e}}$, between the number of evaluations of async^1 and async^5 *Hy3* ($\Delta_{\bar{e}} = \bar{e}_{\mathrm{Async}^1} - \bar{e}_{\mathrm{Async}^5}$). In this figures we do not include the resulting values for the tests with a zero hit rate; thus, for example, there not exist columns for the problem f_{Ras}.

One can observe that the async^5 model is always faster and performs a lower number of evaluations than the async^1 one, both in Fast-Ethernet and Myrinet. This can be justified by two facts. First, the async^5 *Hy3* is faster because the extra computation is reduced by increasing the polling gap; and, second, the async^5 model can find acceptable solutions faster than the async^1 one since it is more exploitative, so it has to evaluate a smaller number of individuals. This same reasons work against the hit rate of async^5 versus async^1 for some problems.

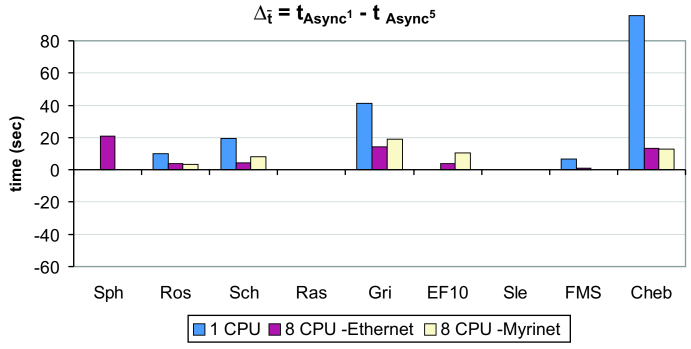

Fig. 8. Difference between async^1 and async^5 execution times.

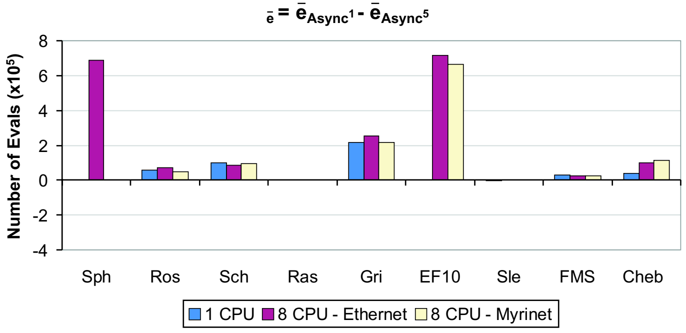

Fig. 9. Difference between async[1] and async[5] number of evaluations.

5. Conclusions and future work

In this paper we include a further study on parallelizing a sequential algorithm called GD-RCGA. The new model has been called *Hy3*. With this model we investigate the advantages that provides an asynchronous design versus a synchronous one. We also present the evaluation of the parallel execution of *Hy3* over two local area networks: a Fast-Ethernet network and a Myrinet network. The motivation for analyzing its parallel execution is that the algorithm has a great accuracy for optimization problems coming from the continuous domains in mathematics. Since the algorithm performs a search based in the separate execution of eight subpopulations with migrations in a cube, it has been readily direct its physical parallelization.

Then, we have focused on the time and numerical efficiency. We have performed all the analysis under the assumption that our parallel versions must reach the same average solution quality as the one reported by the basic reference work. With this goal in mind, we have solved nine problems.

If we analyze the 1 CPU case, the sync model is faster; but when we shift to use 8 CPUs, this model is slower than the asynchronous one. A clear conclusion is that the global parallel efficiency of the parallel *Hy3* is increased with the difficult instances such as f_{fms} and f_{Cheb}. Thus the model will allow us to tackle really complex problems. The lack of efficiency in the simple instances can be explained because the original GD-RCGA model is targeted to run on a monoprocessor. The comparison between parallel executions of sync and async *Hy3* models over the two networks also shows that they do not take advantage of faster networks such as Myrinet, since the amount of information transmitted by the migration mechanism is small.

We have also studied the influence of the polling frequency in the asynchronous *Hy3* model. We have define the polling gap, PG, as the number of iterations between polling operations. Then we have considered $\text{PG} = 1$ and $\text{PG} = 5$, calling the models async[1] *Hy3* and async[5] *Hy3*, respectively. The results indicate that the async[5] model is faster and performs a smaller number of evaluations than the async[1] *Hy3*, but at the cost of a smaller hit rate sometimes.

As a future work, we will apply the model to combinatorial optimization. Also, we plan to introduce a restart technique, and a new modified model of search to improve the results on even more complex problems. This new model will consist in a hypercube of dimension n (with $n = 4, 5, 6, \ldots$). Finally, the better parallel efficiency of async *Hy3* models, along with the present availability of a large number of computing resources in the Internet, suggest the execution over a WAN platform in an asynchronous manner.

Acknowledgements

This work has been partially funded by the Ministry of Science and Technology and FEDER under contracts TIC2002-04498-C05-02 (the TRACER project) and TIC2002-04309-C02-02.

References

[1] E. Alba, F. Luna, A.J. Nebro, Parallel heterogeneous genetic algorithms for continuous optimization, in: IPDPS-NIDISC'03, Nize, France, 2003, p. 147.

[2] J.H. Holland, Adaptation in natural and artificial systems, Ph.D. thesis, University of Michigan, Ann Arbor, 1975.

[3] E. Alba, J.M. Troya, A survey of parallel distributed genetic algorithms, Complexity 4 (1999) 31–52.

[4] E. Alba, A.J. Nebro, J.M. Troya, Heterogeneous computing and parallel genetic algorithms, Journal of Parallel and Distributed Computing 62 (2002) 1362–1385.

[5] P. Adamidis, V. Petridis, Co-operating populations with different evolution behaviors, in: Proc. of the Third IEEE Conf. on Evolutionary Computation, IEEE Press, New York, 1996, pp. 188–191.

[6] P. Adamidis, V. Petridis, On modelling evolutionary algorithm implementations through co-operating populations, in: Parallel Problem Solving from Nature (PPSN VII), Springer-Verlag, 2002, pp. 321–330.

[7] T. Hiroyasu, M. Miki, M. Negami, Distributed genetic algorithms with randomized migration rate, in: Proc. of the IEEE Conf. of Systems, Man and Cybernetics, vol. 1, IEEE Press, 1999, pp. 689–694.

[8] M. Miki, T. Hiroyasu, M. Kaneko, K. Hatanaka, A parallel genetic algorithm with distributed environment scheme, in: Proc. of the 1999 IEEE Conf. of Systems, Man and Cybernetics, IEEE Press, 1999, pp. 695–700.

[9] R. Hinterding, Z. Michalewicz, T.C. Peachey, Self-adaptive genetic algorithm for numeric functions, Parallel Problem Solving from Nature (PPSN IV), 1996, pp. 420–429.

[10] D. Schlierkamp-Voosen, H. Mühlenbein, Strategy adaptation by competing subpopulations, in: Y. Davidor, H.-P. Schwefel, R. Männer (Eds.), Parallel Problem Solving from Nature (PPSN III), Springer-Verlag, Berlin, Germany, 1994, pp. 199–208.

[11] V. Schnecke, O. Vornberger, An adaptive parallel algorithm for VLSI-layout optimization, in: Parallel Problem Solving from Nature (PPSN IV), Berlin, Germany, 1996, pp. 22–27.

[12] F. Herrera, M. Lozano, Heterogeneous distributed genetic algorithms based on the crossover operator, in: 2nd IEE/IEEE Int. Conf. Genetic Algorithms Eng. Syst.: Innovations Appl., 1997, pp. 203–208.

[13] F. Herrera, M. Lozano, C. Moraga, Hybrid distributed real-coded genetic algorithms, in: Parallel Problem Solving from Nature (PPSN V), 1998, pp. 879–888.

[14] U. Aickelin, L. Bull, Partnering strategies for fitness evaluation in a pyramidal evolutionary algorithm, in: Proc. of the Genetic and Evolutionary Computation Conference GECCO'02, Morgan Kaufmann, New York, USA, 2002, pp. 263–270.

[15] S.-L. Lin, W.F. Punch III, E.D. Goodman, Coarse-grain parallel genetic algorithms: categorization and new approach, in: Sixth IEEE Symp. on Parallel and Distributed Processing, IEEE Press, 1994, pp. 28–37.
[16] J.C. Potts, T.D. Giddens, S.B. Yadav, The development and evaluation of an improved genetic algorithm based on migration and artificial selection, IEEE Trans. Syst., Man Cybernet. 24 (1) (1994) 73–86.
[17] M. Sefrioui, J. Périaoux, A hierarchical genetic algorithm using multiple models for optimization, in: Parallel Problem Solving from Nature (PPSN VI), Paris, France, 2000, pp. 879–888.
[18] S. Tsutsui, Y. Fujimoto, Forking genetic algorithm with blocking and shrinking modes (fGA), in: S. Forrest (Ed.), Proc. of the Fifth Int. Conf. on Genetic Algorithms, Morgan Kaufmann, 1993, pp. 206–213.
[19] M.G. Arenas, P. Collet, A.E. Eiben, M. Jelasity, J.J. Merelo, B. Paechter, M. Preub, M. Schoenauer, A framework for distributed evolutionary algorithms, in: Parallel Problem Solving from Nature (PPSN VII), Springer-Verlag, 2002, pp. 665–675.
[20] Tierra Home Page: www.hip.atr.co.jp/~ray/tierra/tierra.html.
[21] J.J. Hu, E.D. Goodman, The hierarchical fair competition (HFC) model for parallel evolutionary algorithms, in: Proc. of the 2002 Congress on Evolutionary Computation CEC2002, IEEE Press, 2002, pp. 49–54.
[22] S.-K. Oh, C.-Y. Lee, J.-J. Lee, A new distributed evolutionary algorithm for optimization in nonstationary environments, in: Proc. of the 2002 Congress on Evolutionary Computation, IEEE Press, 2002, pp. 378–383.
[23] D. Schlierkamp-Voosen, H. Mühlenbein, Adaptation of population sizes by competing subpopulations, in: Proc. of the Int. Conf. on Evolutionary Computation, Nagoya, Japan, 1996, pp. 330–335.
[24] W. Yi, Q. Liu, Y. He, Dynamic distributed genetic algorithms, in: Proc. of the 2000 Congress on Evolutionary Computation, IEEE Press, 2000, pp. 1132–1136.
[25] F. Herrera, M. Lozano, Gradual distributed real-coded genetic algorithms, IEEE Trans. Evolution. Comput. 4 (1) (2000) 43–63.
[26] R. Venkateswaran, Z. Obradović, C.S. Raghavendra, Cooperative genetic algorithm for optimization problems in distributed computer systems, in: Proc. of the Second Online Workshop on Evolutionary Computation, 1996, pp. 49–52.
[27] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Trans. Evolution. Comput. 6 (5) (2002) 443–462.
[28] A.J. Nebro, E. Alba, F. Luna, J.M. Troya, .NET as a platform for implementing concurrent objects, in: B. Monien, R. Feldmann (Eds.), 8th Euro-Par, Springer-Verlag, Paderborn, GE, 2002, pp. 125–130.
[29] K.A. de Jong, An analysis of the behavior of a class of genetic adaptive systems, Ph.D. thesis, University of Michigan, Ann Arbor, 1975.
[30] E. Alba, J.M. Troya, Influence of the migration policy in parallel distributed GAs with structured and panmictic populations, Appl. Intell. 12 (3) (2000) 163–181.
[31] E. Cantú-Paz, A summary of research on parallel genetic algorithms, Tech. Rep. 95007, GA Laboratory, University of Illinois, Urbana-Champaign, IL, 1995.
[32] D.E. Goldberg, H. Kargupta, J. Horn, E. Cantú-Paz, Critical deme size for serial and parallel genetic algorithms, Tech. Rep. 95002, GA Laboratory, University of Illinois, Urbana-Campaign, IL, 1995.
[33] J.E. Baker, Adaptive selection methods for genetic algorithms, in: J.J. Grefenstette (Ed.), Proc. of 1st Int. Conf. Genetic Algorithms Appl., Lawrence Erlbaum, Hillsdale, NJ, 1985, pp. 101–111.
[34] J.E. Baker, Reducing bias and inefficiency in the selection algorithm, in: J.J. Grefenstette (Ed.), Proc. of 2nd Int. Conf. Genetic Algorithms Appl., Lawrence Erlbaum, Hillsdale, NJ, 1987, pp. 14–21.
[35] H.-P. Schwefel, Numerical Optimization of Computer Models, Wiley, Chichester, UK, 1981.
[36] A. Töorn, Ž. Antanas, Global Optimization, in: LNCS, vol. 350, Springer, Berlin, Germany, 1989.
[37] T. Bäck, Self-adaptation in genetic algorithms, in: F.J. Varela, P. Bourgine (Eds.), Proc. of 1st Euro. Conf. Artif. Life, MIT Press, Cambridge, MA, 1992, pp. 263–271.
[38] A.O. Griewangk, Generalized descent of global optimization, J. Optim. Theory Appl. 34 (1981) 11–39.

[39] D. Whitley, R. Beveridge, C. Graves, K. Mathias, Test driving three genetic algorithms: new test functions and geometric matching, J. Heuristics 1 (1995) 77–104.

[40] L.J. Eshelman, K.E. Mathias, J.D. Schaffer, Convergence controlled variation, in: R. Belew, M. Vose (Eds.), Foundations of Genetic Algorithms 4, Morgan Kaufmann, San Mateo, CA, 1997, pp. 203–224.

[41] R. Storn, K. Price, Differential evolution—a simple efficient adaptive scheme for global optimization over continuous spaces, Tech. Rep. 95-012, Int. Compt. Sci. Inst., Berkeley, CA, 1995.

[42] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution programs, Springer, Berlin, Germany, 1992.

[43] A.H. Karp, H.P. Flatt, Measuring parallel processor performance, Comm. ACM 33 (1990) 539–543.

[44] E. Alba, Parallel evolutionary algorithms can achieve super-linear performance, Inform. Process. Lett. 82 (2002) 7–13.