International Journal of Foundations of Computer Science

On the Behavior of Parallel Genetic Algorithms for Optimal Placement of Antennae in Telecommunications

E. Alba

Departamento de Lenguajes y Ciencias de la Computación
Campus de Teatinos, E.T.S.I. Informática
University of Málaga, 29071 Málaga, Spain
eat@lcc.uma.es
http://polaris.lcc.uma.es/∼eat/

F. Chicano

Departamento de Lenguajes y Ciencias de la Computación
Campus de Teatinos, E.T.S.I. Informática
University of Málaga, 29071 Málaga, Spain
chicano@lcc.uma.es
http://neo.lcc.uma.es/staff/francis/index.html

In this article, evolutionary algorithms (EAs) are applied to solve the radio network design problem (RND). The task is to find the best set of transmitter locations in order to cover a given geographical region at an optimal cost. Usually, parallel EAs are needed to cope with the high computational requirements of such a problem. Here, we develop and evaluate a set of sequential and parallel genetic algorithms (GAs) to solve the RND problem efficiently. The results show that our distributed steady state GA is an efficient and accurate tool for solving RND that even outperforms existing parallel solutions. The sequential algorithm performs very efficiently from a numerical point of view, although the distributed version is much faster.

Keywords: parallel evolutionary algorithm; radio network design; performance evaluation.

1. Introduction

An important symbol of our present information society are telecommunications. With a rapidly growing number of user services, telecommunications is a field in which many open research lines are challenging the research community. Many of the problems found in this area can be formulated as optimization tasks. Some examples are assigning frequencies in radio link communications[1], predicting bandwidth demands in ATM networks[2], developing error correcting codes for transmission of messages[3], and designing the telecommunication network[4,5,6,7].

The problem tackled in this paper belongs to this broad class of network design tasks. When a geographically dispersed set of terminals needs to be covered by transmission antennae a key issue is to minimize the number and locations of these

antennae to cover a maximum area. This is usually called the *radio network design* problem (RND).

Some existing approaches for solving this problem include the utilization of an evolutionary algorithm, e.g., genetic algorithms[8]. But, as it happens frequently in practice, the high complexity of this task needs the computational power of several machines working together to find a feasible solution. This gives rise to the application of parallel algorithms to achieve high efficiency.

In this article, our goal is to analyze different genetic algorithms to find out which of them is more suited to solve efficiently and accurately the RND problem. In general, evolutionary algorithms encode tentative problem solutions in a population of individuals with an associated fitness (quality) value and then *evolve* them towards increasingly better search regions. Thus, numerical issues are quite important in order to make a fair comparison with future optimization techniques. In particular, we will analyze the relative advantages of using a single pool of solutions versus an algorithm having multiple (parallel) pools of solutions. Also, defining an appropriate fitness function is a capital issue in EAs that we analyze here, since it usually encapsulates as much problem knowledge as possible to better guide the algorithm. We will then analyze some alternatives to fitness function definitions for the same problem. Finally, the actual goal of a telecommunication network designer is to get an optimum design at a maximum speed; therefore, we will encompass a set of tests to find how efficient these algorithms could be, and analyze their scalability when faced to un seen problems.

The paper is organized as follows. In the second section we define and characterize the radio network design problem. In Section 3 we will briefly describe evolutionary algorithms as well as the necessary details to understand the proposed ones. Then, Section 4 will discuss some design issues relating the encoding and fitness functions we should use to drive the search. In Section 5 we will provide the results of the tests performed to compare algorithms and fitness functions, developing efficient parallel algorithms. Then, we perform a deeper analysis of the distributed algorithms in Section 6. Finally, some concluding remarks and future research lines are drawn in Section 7.

2. The Radio Network Design Problem (RND)

The radio coverage problem amounts to covering an area with a set of transmitters. The part of an area that is covered by a transmitter is called *a cell*. A cell is usually disconnected. In the following we will assume that the cells and the area considered are discretized, that is, they can be described as a finite collection of geographical locations (taken from a geo-referenced grid, for example). The computation of cells may be based on sophisticated wave propagation models, on measurements, or on draft estimations. In any case, we assume that cells can be computed and returned by an *ad hoc* function.

Let us consider the set L of all potentially covered locations and the set M of

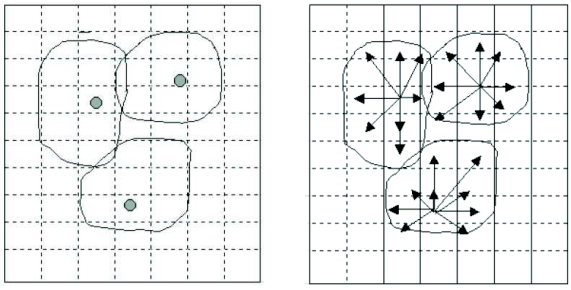

Fig. 1. (left) Three potentially transmitter locations and their associated covered cells on a grid, and (right) graph representing covered locations.

all potential transmitter locations. Let G be the graph, $(M \cup L, E)$, where E is a set of edges such that each transmitter location is linked to the locations it covers and let be the vector $\vec{x}$ a solution to the problem where $x_i \in \{0, 1\}$, and $i \in [1, |M|]$ indicates whether a transmitter is being used or not. As the geographical area needs to be discretized, the potentially covered locations are taken from a grid, as shown in the Fig. 1.

Searching for the minimum subset of transmitters that covers a maximum surface of an area comes to searching for a subset $M' \subseteq M$ such that $|M'|$ is minimum and such that $|Neighbors(M', E)|$ is maximum, where

$$Neighbors(M', E) = \{u \in L \mid \exists v \in M', (u, v) \in E\}. \tag{1}$$

$$M' = \{t \in M \mid x_t = 1\}. \tag{2}$$

The problem we consider recalls the unicost set covering problem (USCP) that is known to be NP-hard. The radio coverage problem differs, however, from the USCP in that the goal is to select a subset of transmitters that ensures a *good* coverage of the area and not to ensure a *total* coverage. The difficulty of our problem arises from the fact that the goal is twofold, no part being secondary. If minimizing was the primary goal, the solution would be trivial: $M' = \emptyset$. If maximizing the number of covered locations was the primary goal, then problem would be the USCP. An objective function $f(\vec{x})$ to combine the two goals has been proposed in Ref. 8:

$$f(\vec{x}) = \frac{CoverRate(\vec{x})^{\alpha}}{Number\ of\ transmitters\ selected(\vec{x})}. \tag{3}$$

where

$$CoverRate(\vec{x}) = 100 \cdot \frac{Neighbors(M', E)}{Neighbors(M, E)}. \tag{4}$$

the parameter α can be tuned to favor the cover rate item with respect to the number of transmitters. Just like Calégari et al. did[8], we will use $\alpha = 2$, and 287×287 point grid representing an open-air flat area. Also, 49 primary transmitter locations are

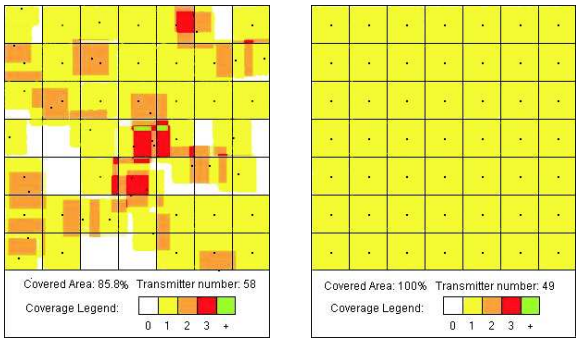

Fig. 2. (left) Graphical representation of a partial solution covering 85.8% of the whole area, and (right) a graphical representation of an optimal solution.

distributed regularly in this area in order to form a 7×7 grid structure, and each transmitter has an associated 41×41 point cell.

Consequently, the obtained coverage would be total if the algorithm happens to assign one transmitter to these optimal locations known beforehand. A hundred complementary transmitter locations were then randomly added, associated to 41×41 point cells. By construction, the best solution with total coverage is the one that covers the area with the 49 primary transmitters (giving the fitness value 204.08). See in Fig.2 the graphical representation of a partial (left) and optimal (right) solution for the RND problem. This problem admits numerous extensions like defining different types of antennae, finding their 3D orientation or propagating signals according to physical simulation of radio waves, what really make a hard real world application[9].

3. Evolutionary Algorithms

In this section we intend to provide a quick overview of the evolutionary algorithms family in order to classify and explain the class of algorithms we are using in the paper.

Let us begin by outlining the skeleton of a standard evolutionary algorithm. An EA (see the following pseudo-code) proceeds in an iterative manner by generating populations P(t) of μ individuals from the old ones (t=0, t=1, t=2, ...). Every individual in the population is the encoded (binary, real, ...) version of a tentative solution. An evaluation function associates a fitness value to every individual indicating its suitability to the problem. The canonical algorithm applies stochastic operators such as selection, recombination, and mutation on an initially random population in order to compute a whole generation of new individuals. In a general formulation, we apply *variation operators* to create a temporary population P'(t), evaluate the resulting individuals, and get a new population P(t+1) by either using P'(t) and, optionally P(t). The stop condition is usually set to reach a pre-

Evolutionary Algorithm

```
t := 0;
initialize and evaluate [P(t)];
while not stop_condition do
        P'(t) := variation [P(t)];
        evaluate [P'(t)];
        P(t+1) := select [P'(t),P(t)];
        t := t + 1;
end while;
```

Fig. 3. Psuedo-code of an evolutionary algorithm.

programmed number of iterations of the algorithm, or to find an individual with a preset (non-optimal) final quality.

It is usual for many EA families to manipulate the population as a single pool of individuals. By *manipulation* we mean to apply selection of the fittest individuals, recombination of slices of two individuals to yield one or two new children, and mutation of their contents. Frequently, EAs use these variation operators in conjunction with associated probabilities that govern their application in each step of the algorithm.

In general, any individual can potentially mate any other by applying a centralized selection operator. The same holds for the replacement operator, where any individual can potentially leave the pool and be replaced by a new one. This is called a *panmictic* population of individuals. A different (decentralized) selection model exists in which individuals are arranged spatially, therefore giving place to *structured EAs*. Most other operators, such as recombination or mutation, can be readily applied to any of these two models[10].

There exist two quite popular classes of panmicitic EAs having different granularity at the reproductive step[11]. The first one is called "generational" model, in which a whole new population of λ individuals replaces the old one (right part of Fig.4, where μ is the population size). The second type is called "steady state", since usually one or two new individuals are created at every step of the algorithm and then they are inserted back into the population, consequently coexisting with their parents. In the mean region, there exists a plethora of selection models, generically termed as "generation gap" algorithms, in which a given percentage of the individuals are replaced with the new ones. Clearly, generational and steady state selections are two special subclasses of generation gap algorithms.

Centralized versions of selection are typically found in serial EAs, although some parallel implementations have also used it. For example, the *global parallelism* approach evaluates in parallel the individuals of the population (sometimes also recom-

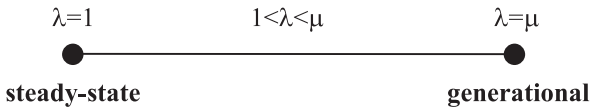

Fig. 4. Panmictic EAs: from steady-state to generational algorithms.

bination and/or mutation are parallelized), while still using a centralized selection performed sequentially in the main processor guiding the base algorithm[12]. This algorithm keeps the same behavior as the sequential centralized one, although it usually performs much faster for time-consuming objective functions.

Most parallel EAs found in the literature usually utilize some kind of spatial disposition for the individuals, and then parallelize the resulting chunks in a pool of processors. We must stress at this point of the discussion that parallelization is mainly achieved by first structuring the panmictic algorithm, and then parallelizing it. This is why we distinguish throughout the paper between structuring populations and making parallel implementations, since the same structured EA can admit many different implementations.

Therefore, we now turn to consider structured algorithms also in this work. There exists a long tradition in using evolutionary algorithms having structured populations, especially in association with parallel implementations. Among the most widely known types of structured EAs, *distributed* (dEA) and *cellular* (cEA) algorithms are very popular optimization procedures[10].

Decentralizing a single population can be achieved by partitioning it into several sub-populations, where island EAs are run performing sparse exchanges of individuals (distributed EAs), or in the form of neighborhoods (cellular EAs).

In distributed EAs, additional parameters controlling when migration occurs and how migrants are selected/incorporated from/to the source/target islands are needed[13,14]. In cellular EAs, the existence of overlapped small neighborhoods helps in exploring the search space[15]. These two kinds of EAs seem to provide a better sampling of the search space and to improve the numerical and runtime behavior of the basic algorithm in many cases[16,17].

In the present study we mainly focus on distributed EAs. A distributed EA is a multi-population (island) model performing sparse exchanges of individuals among the elementary populations. This model can be readily implemented in distributed memory MIMD computers, one main reason for its popularity. A migration policy controls the search. The migration policy must define the island topology, when migration occurs, which individuals are being exchanged, the synchronization among the sub-populations, and the kind of integration of exchanged individuals within the target sub-populations. The advantages of a distributed model (either running on separate processors or not) is that it is usually faster than a panmictic EA. The reason for this is that the run time and the number of evaluations are potentially reduced thanks to its separate search from several regions of the problem space.

A high diversity and species formation are two of the well reported features of distributed EAs.

In this work we analyze the performance of a steady state GA (ssGA) to solve the RND problem. Additionally, we have developed a distributed ssGA (dssGA) that can be run on any number of processors (also on one single processor) for solving the RND problem more efficiently. The dssGA falls into the distributed structured GA class allowing its concurrent or physically parallel execution on a cluster of workstations.

4. A Fitness Function for Solving RND

Designing a fitness function is one of the most important steps in applying an EA to a problem. The kind of representation being used in the algorithm influences its use inside the fitness function. In this section, we will address the definition of the fitness function and the interpretation of the resulting GA strings that hopefully will guide the algorithms towards the problem regions where the solution reside.

In our representation, every potential transmitter location is assigned a bit in the binary string manipulated in the algorithm. A 1 in a given string position means that the associated transmitter will be used in the placement, i.e., more points are potentially covered in the grid and one more transmitter needs to be considered in all the computations inside the fitness function.

The fitness function accounts for all problem knowledge details since it scans the binary string provided by the algorithm. In this way it computes the covered area as well as other statistical measures that could be needed to assess the quality of the evaluated individual as a tentative solution to the problem. In our case, we will investigate the relative performances of two evaluation functions.

The first one is a fitness function that directly follows the problem definition given in Section 2, thus computing the ratio between the covered and total area in the problem (the grid). See in (5) the used function, by which we maximize the covered area and, at the same time, we penalize solutions having a large number of transmitters.

$$Evaluate1(\vec{x}) = \frac{CoverRate(\vec{x})^{\alpha}}{Number\ of\ transmitters\ selected(\vec{x})}. \tag{5}$$

This first evaluation function was also used in Ref. 8, working out a considerably high time before computing a solution. However, we are interested in solving the problem for total coverage and that is the reason for designing a second fitness function that could directly account for the features that an optimal solution should exhibit, thus hopefully reducing the number of sampled points. In particular, we know that the mean number of transmitters covering a single point should be 1.0 in an optimal solution for total coverage. This is computed in a penalty term P_m explained in (6). See in Table 2 the meaning of the symbols used in the equations.

$$P_m(\vec{x}) = \frac{\Phi}{100} \cdot (\overline{t}(\vec{x}) \cdot a_m + b_m). \tag{6}$$

But it is not enough to have a mean number of transmitters of 1.0 over each point in the target area. In addition, it should hold that the standard deviation have a value of 0.0, since in this way we could ensure that each point is covered by one and only one transmitter in the solution. See this second penalty term P_v in (7).

$$P_v(\vec{x}) = \frac{\Phi}{100} \cdot (\sigma_t^2(\vec{x}) \cdot a_v + b_v). \tag{7}$$

We have called these two terms *penalty* values since we plan to penalize the first evaluation function with both of them in order to engineer the algorithm with a more accurate guide along the search space. See in (8) the resulting evaluation function intended to be maximized by the analyzed algorithms.

$$Evaluate2(\vec{x}) = Evaluate1(\vec{x}) - |P_m(\vec{x})| - |P_v(\vec{x})|. \tag{8}$$

Therefore, we are going to compare the results by using these two evaluation functions throughout our tests. It is expected for the second function to be harder to compute, since we need to calculate more statistical info inside the problem simulator embedded in the fitness function; on the contrary, we expect it to find out a solution with a smaller numerical effort.

See in Table 1 the numeric ranges of the fitness values returned by each of the maximized functions. We set constants $a_i,\ b_i$ to have values in the same range, thus making more understandable the tests.

Table 1. Numeric ranges for the result of every fitness function.

Function	*Ranges*
Evaluate1	[0,204.08]
Evaluate2	[0,204.08]

In Table 2 we deploy a list with all the used symbols in order to help the reader to understand the previous equations.

5. Tests and Results

In this section we present the results of performing an assorted set of tests by using sequential and parallel GAs to solve the RND problem (the two evaluation functions are considered throughout).

First, we will analyze the number of evaluations and time needed by each configuration. Let us begin by considering a sequential implementation of a steady state GA with a single population of 512 individuals, utilizing a usual parameterization, namely roulette wheel selection of each parent, double-point crossover (dpx) with probability 1.0, bit-flip mutation with probability $1/strlen$, and replacement always of the worst string.

Table 2. Meaning of the symbols.

Symbol	*Description*
G	Graph representing the problem
L	Set of all potentially covered locations
M	Set of all potential transmitter locations
E	Set of edges linking a transmitter to its covered locations
$\vec{x}$	Solution vector to the problem
$\overline{t}$	Mean number of transmitters associated to a location
σ_t^2	Variance of the mean number of transmitters for a location
P_m	Penalty value for $\overline{t}$
P_v	Penalty value for σ_t^2
Φ	Maximum value of $f(\vec{x})$
a_m, b_m	Coefficients for penalty P_m ($a_m = 4.878, b_m = -4.878$)
a_v, b_v	Coefficients for penalty P_v ($a_v = 2.404, b_v = 0.0$)

Then we will analyze the results of a parallel steady state GA having 8 islands, each one with 64 individuals performing in parallel the mentioned basic evolutionary step, with an added migration operation. The migration will occur in a unidirectional ring manner, sending one single randomly chosen individual to the neighbor sub-population. The target population incorporates this individual only if it is better than its presently worst solution. The migration step is performed every 2048 iterations in every island in an asynchronous way, since it is expected to be more efficient than a synchronous execution over the pool of available processors[18]. We will run the algorithms both in one single CPU (i.e., concurrently) and on 8 processors. Each processor is a Pentium 4 at 2.8 GHz linked by a Fast Ethernet communication network. Also, see a summary of the conditions for experimentation in Table 3. We perform 30 independent runs of each experiment.

Table 3. Parameters of the algorithms being used.

Population Size	*512*
Selection	*roulette wheel*
Crossover	*dpx prob = 1.0*
Mutation	*bit-flip prob = 0.00671*
Replacement	*least fitted*
Stop Criterion	*find a solution*
Number of islands	*8*
Migration policy for selection	*random selection*
Migration policy for replacement	*replace if better*
Migration gap	*2048*
Number of migrants	*1*
Synchronization	*asynchronous mode*

Now, let us begin the analysis by presenting in Table 4 and 5 the average number of evaluations and the average running time for two algorithms: the ssGA and the distributed ssGA with 8 islands (this last in two configurations: one running on 1 processor and another on 8 processors).

Table 4. Average number of evaluations for *RND* problem.

#evals	*Evaluate1*	*Evaluate2*	*t-test*
ssGA	158794	190240	0.1757
dssGA(1 CPU)	611381	631035	0.6714
dssGA(8 CPUs)	785893	785100	0.9901

Table 5. Average time of execution for *RND* problem.

time (sec.)	*Evaluate1*	*Evaluate2*	*t-test*
ssGA	145	228	$1.445 \cdot 10^{-03}$
dssGA(1 CPU)	489	709	$3.106 \cdot 10^{-05}$
dssGA(8 CPUs)	80	111	$3.758 \cdot 10^{-04}$
speedup	6.11	6.39	

Table 6. Student *t-tests* comparing the different algorithms.

	Evaluate1		*Evaluate2*	
	time	*#evals*	*time*	*#evals*
ssGA-*dssGA*(1CPU)	$2.722 \cdot 10^{-15}$	$< 2.2 \cdot 10^{-16}$	$1.561 \cdot 10^{-13}$	$1.765 \cdot 10^{-14}$
ssGA-*dssGA*(8CPUs)	$8.935 \cdot 10^{-05}$	$3.077 \cdot 10^{-14}$	$3.982 \cdot 10^{-06}$	$5.866 \cdot 10^{-16}$
dssGA(1CPU)-*dssGA*(8CPUs)	$2.345 \cdot 10^{-16}$	$3.715 \cdot 10^{-03}$	$3.115 \cdot 10^{-15}$	$6.066 \cdot 10^{-03}$

To assess the statistical significance of the results we not only performed 30 independent runs, but also computed a Student t-test analysis so that we could be able to distinguish meaningful differences in the average values. The significance *p-value* is assumed to be 0.05, in order to indicate a 95% confidence level in the results.

If we interpret the results in the tables we can notice several facts. Firstly, for Evaluate1, it is clear that the sequential ssGA is the best algorithm numerically, since it samples almost 4 times less number of points in the search space than the second best algorithm before locating an optimal solution. As to the search time, the best algorithm is the dssGA8 heuristic on 8 processors, since it performs very quickly (80 seconds) in comparison with the ssGA (145 seconds), and the dssGA8 on a single processor (489 seconds). This comes as no surprise since numerically, ssGA was better, although dssGA8 on 8 processors makes computations much faster. Since it is not fair to compute speedup against a panmictic ssGA[19] we compare the same algorithm, both in sequential and parallel (dssGA8 on 8 versus 1 processors), with the stopping criterion of getting a solution of the same quality. We then are allowed to perform a fair comparison, whose conclusion is that speedup is 6.11 with 8 processors which is a desirable result.

For the Evaluate2 fitness function (see the tables again) we can notice that the results are much the same: ssGA is numerically advantageous with respect to any execution scenario of dssGA8. Again, the execution time relationship is just like with the first evaluation function (in fact, speedup is slightly higher: 6.39). The t-tests

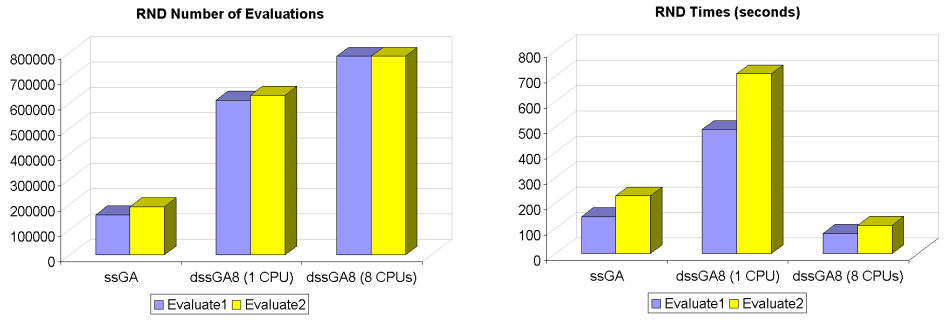

Fig. 5. Number of evaluations (left) and time (right) needed by (d)ssGA to solve RND.

(Table 6) comparing the three scenarios make us conclude that all de differences in time and number of evaluations are significant in the two cases (using Evaluate1 and Evaluate2).

Therefore, the conclusions are that, numerically speaking, ssGA is the best algorithm for any of the evaluation functions. It can also be concluded that the second evaluation function does not reduce the number of sampled points significantly (counterintuitive). In addition, the execution time of all the algorithms is clearly increased when using Evaluate2. It might be possible that a multiobjective redefinition of the problem based in the ideas of such function could lead to a more efficient algorithm. The statistical significance ($>> 0.05$) indicates us that Evaluate1 and Evaluate2 have much the same numerical performance, although it seems that there exists a trend to Evaluate1 being slightly better. See a graphical interpretation of the results in Fig.5, where we plot the average number of evaluations and execution time for ssGA and for the serial and parallel versions of dssGA8.

6. Further Understanding on the Distributed Algorithms for RND

In this section we want to discuss some other aspects of the process of solving RND with a dssGA. First, we explore the behaviour of the algorithms when the problem is solved only at 61% of the optimum in the next subsection. Then, we solve the problem with distributed genetic algorithms using different parameterizations such as increasing the number of CPUs (Subsection 6.2) or varying the migration gap (Subsection 6.3). Finally, we solve larger problem instances with different numbers of (useless, redundant) transmitter locations in Subsection 6.4 to analyze the scalability of the algorithms to increasingly harder instances.

6.1. *Solving RND at 61%*

In Ref. 8 the authors did not solve the problem completely (optimum at 204.08), but only partially at different percentages of this optimum, in particular at 61% of the maximum fitness (optimum at/above 125.4). We consider in this subsection

solving the problem at a 61% of the maximum, to stress the point that the difficulty of RND resides not only in locating a solution, but also in the very slow progress of the algorithm towards an optimum, what needs small refining steps.

The interest of such tests comes from the practical application of the algorithm by an engineer, that usually is interested in getting a good guess in a small time. The results in Ref. 8 show a required time of 540 minutes and 40,000 evaluations, and our algorithm makes the same work in only a few seconds and 5,600 evaluations approximately (see Table 7).

Our implementation is faster than the one they used, since they perform 74 evaluations per minute in one processor while we can perform 70,000 evaluations per minute. This is due, of course, to the fact that we are using different machines, but also since they implement graphs and other data structures that we have taken care of in our implementations from the complexity point of view, in order to speedup the computations of the number of covered points by a solution.

In our results, the numeric data in Tables 7 and 8 confirm that the first evaluation function is the best in reducing the number of visited points for the simplified problem (the differences are statistically significant).

Table 7. Results for Evaluate1 *RND* 61% *dssGA*.

	#evals	*time(sec.)*	*Fitness*
dssGA(1 CPU)	5614	5.03	127.41
dssGA(8 CPUs)	5571	0.80	126.91

The speedup (when solving the problem at 61%) between the concurrent and parallel dssGA8 is 6.29 for Evaluate1 and 8.49 (superlinear) for Evaluate2. This result confirms that Evaluate2 is better in profiting from a larger number of processors, which is generally considered an advantage in parallel programming. However, Evaluate1 is so fast in locating a 61% of the solution with 8 processors that we can almost forget about this small advantage of Evaluate2, at least for this RND problem instance.

Table 8. Results for Evaluate2 *RND* 61% *dssGA*.

	#evals	*time(sec.)*	*Fitness*
dssGA(1 CPU)	6349	7.90	126.76
dssGA(8 CPUs)	6274	0.93	126.99

6.2. *Influence of the Number of CPUs*

To perform a deeper study on the influence of the number of CPUs we have implemented a distributed genetic algorithm with 16 islands *dssGA16* that we run on 1, 2, 4, 8, and 16 machines. For this experiment and the forthcoming ones we use only

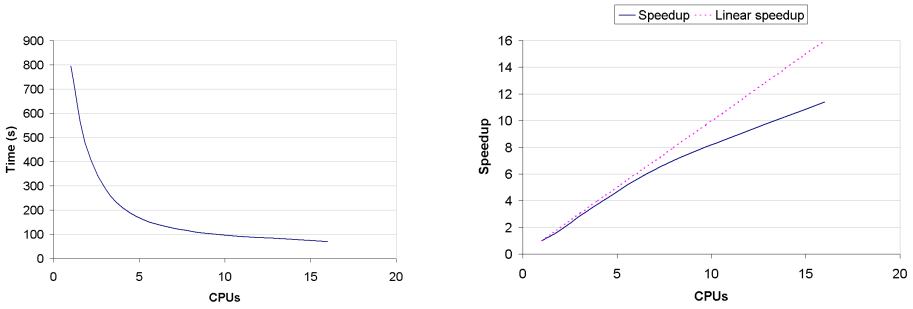

Fig. 6. Execution time (left) and speedup (right) of *dssGA16* for different number of CPUs.

Evaluate1 due to its advantages in time and number of evaluations. The parameters are equal to those used with 8 islands except for the migration gap (1024). In Table 9 and Fig. 6 we present the execution time and number of evaluations for the experiments (the average of 30 independent runs).

Table 9. Average Time and Number of Evaluations for dssGA16.

dssGA16	*time(sec.)*	*#evals*	*speedup*
1CPU	794.97	929177	1.00
2CPUs	445.27	1020001	1.79
4CPUs	211.40	986502	3.76
8CPUs	113.20	1039914	7.02
16CPUs	69.70	1251756	11.41

All the time differences are statistically significant (p-values below 0.05). However, the number of evaluations is not statistically significant (only dssGA16 on 16 CPUs provides statistically significant differences with the rest). The speedup is always sublinear and it slightly moves away from the linear speedup when the number of CPUs increases. That is, when we increment the number of CPUs we have a moderate loss of efficiency.

6.3. *Influence of the Migration Gap*

In this subsection we study the influence of the migration gap in the basic dssGA8 algorithm executed on 8 CPUs. Migration gap is an important parameter of dssGAs controlling how coupled are the separate algorithms, and most problems show very different results for different gaps. In Table 10 and Fig. 7 we show the results.

Our conclusion is that the number of evaluations and execution time of the algorithm dramatically decrease as we increase the migration gap, that is, a loose coupling among the islands highly improves the efficiency. With a low migration gap the algorithm needs more time (and evaluations) for solving the problem. In this

Table 10. Average Time and Evaluations for dssGA8 with different migration gap.

dssGA8	*time(sec.)*	*#evals*
64	2244.20	11817769
256	487.70	2694576
1024	94.43	818582
2048	80.43	785893
4096	74.57	765824

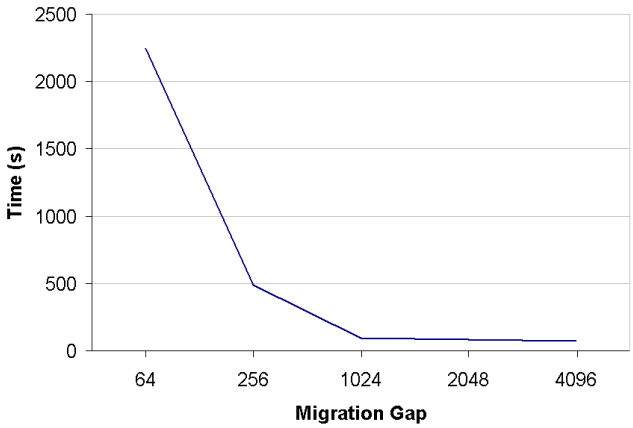

Fig. 7. Execution time of *dssGA8* with different migration gap.

case, the algorithm behaves like a "quasi-panmictic" algorithm. When the migration gap is high the subpopulations have time to evolve toward different regions of the search space, and the combination of solutions from different regions is beneficial for the search. For this reason the time and the number of evaluations decrease as the migration gap increases. However, the curve of the Fig. 7 is a negative exponential-like and it seems to have an asymptote at 70 seconds in the ordinate axis.

6.4. *Algorithm Scalability*

The dimension of the instance affects the execution time and the number of evaluations of the problem. We perform in this subsection a set of experiments to study the influence in the results of the dimension of the instance. The algorithm used is dssGA8 on 8 CPUs. Our aim is to tackle increasingly larger instances of the problem with exactly the same algorithm, in order to analyze its resistance to scalability.

We have added to the initial instance (149 transmitters) additional random transmitter locations to get four new instances with 199, 249, 299, and 349 transmitter locations. In all the instances the optimum solution is the same as for the canonical 149 problem, and the random transmitters are added to deceive the algo-

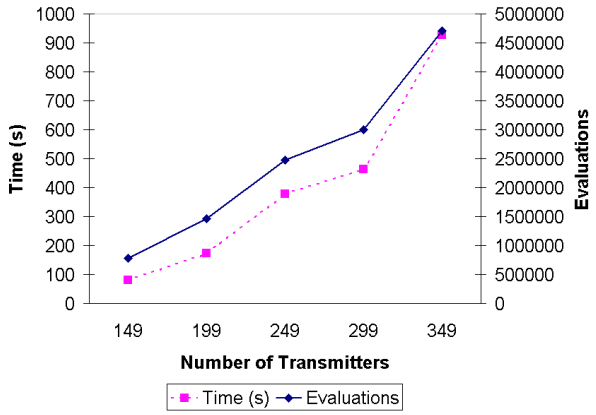

Fig. 8. Execution time of *dssGA8* with different number of transmitters.

rithm (redundant positions). Parameters are the same as shown in Table 3 except that mutation probability has been modified in each instance to maintain the equation $p_m = 1/strlen$. We can see the results in Table 11. In Fig. 8 we trace the times and number of evaluations of the problem for increments of 50 transmitters.

Table 11. Average Time and Evaluations for dssGA8 with different number of transmitters.

dssGA8	*time(sec.)*	*#evals*
149	80	785893
199	174	1467050
249	378	2480883
299	463	2997987
349	927	4710304

In an exhaustive search algorithm the addition of one bit to the length of the string means to double the search time, that is, the search time would be exponential with respect to the dimension of the instance. However, as we can deduce from the results, in an heuristic like our dssGA8 the increment in the search time (and the number of evaluations) for linearly larger instances is only linear with respect to the number of transmitter locations (the chromosome length). This highlights clearly an advantage of metaheuristic algorithms over enumerative ones.

7. Conclusions

In this paper we intended to design better algorithms to solve the RND problem. We have tried two fitness functions and different sequential and parallel genetic algorithms to find an optimal solution to the placement of antennae in a geographical

area, which is an important issue in telecommunications.

Our results show that the evaluation function reported in Ref. 8 works properly without needing additional help coming from explicit penalty terms (as usual in other complex problems). The used steady state GA provides a very good sampling of the search space. His drawback is that it is slow. This drawback has been fixed by using a distributed multi-population version (dssGA) running on an network of workstations (NOW) that, despite making a somewhat less efficient search space sampling, provides a much faster way of execution, reducing the wait time for a solution to just about 80 seconds.

We performed additional experiments to study the behavior of the algorithms when some parameters are changed. In particular, we studied the influence of the number of CPUs in the distributed genetic algorithms, the migration gap, and the dimension of the instance. The results show that we can reduce the search time by increasing the number of CPUs, although the increment is sublinear. A second conclusion is that a high migration gap (isolation) is beneficial for the search. Finally, the increment in the dimension of the problem instances is managed very well by dssGA8, suggesting a good scalability of the algorithm against unseen problems.

Since there exist many possible instances for this same problem, we are working in more difficult scenarios, and of course, we are exploring the utilization of some other kinds of evolutionary algorithms for it, specially cellular GAs[20] and multiobjective EAs.

Acknowledgements

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

References

1. A. Kapsalis, V.J. Rayward-Smith, and G.D. Smith. Using genetic algorithms to solve the radio link frequency assigment problem. In D.W. Pearson, N.C. Steele, and R.F. Albretch, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 37–40. Springer-Verlag, 1995.
2. N. Swaminathan, J. Srinivasan, and S.V. Raghavan. Bandwidth-demand prediction in virtual path in atm networks using genetic algorithms. *Computer Commnunications*, 22(12):1127–1135, 1999.
3. H. Chen, N.S. Flann, and D.W. Watson. Parallel genetic simulated annealing: A massively parallel SIMD algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):126–136, 1998.
4. C.H. Chu, G. Premkumar, and H. Chou. Digital data networks design using genetic algorithms. *European Journal of Operational Research*, 127:140–158, 2000.
5. N. Karunanithi and T. Carpenter. Sonet ring sizing with genetic algorithms. *Computers and Operations Research*, 24(6):581–591, 1997.
6. S. Khuri and T. Chiu. Heuristic algorithms for the terminal assignment problem. In *Proceedings of the 1997 ACM Symposium on Applied Computing*, pages 247–251. ACM Press, 1997.

7. C. Vijayanand, M.S. Kumar, K.R. Venugopal, and P.S. Kumar. Converter placement in all-optical networks using genetic algorithms. *Computer Communications*, 23:1223–1234, 2000.
8. P. Calégari, F. Guidec, P. Kuonen, and D. Kobler. Parallel island-based genetic algorithm for radio network design. *Journal of Parallel and Distributed Computing*, 47:86–90, 1997.
9. E-G. Talbi and P. Reininger. A Multiobjective Genetic Algorithm for Radio Network Optimization. In *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 317–324. IEEE Press, 2000.
10. E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, October 2002.
11. G. Syswerda. A study of reproduction in generational and steady-state genetic algorithms. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 94–101. Morgan Kaufmann, 1991.
12. D. Levine. Users guide to the PGAPack parallel genetic algorithm library. Technical Report ANL-95/18, Argonne National Laboratory, Mathematics and Computer Science Division, January 31 1995.
13. T.C. Belding. The distributed genetic algorithm revisited. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 114–121. Morgan Kaufmann, 1995.
14. R. Tanese. Distributed genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–439. Morgan Kaufmann, 1989.
15. S. Baluja. Structure and performance of fine-grain parallelism in genetic search. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 155–162. Morgan Kaufmann, 1993.
16. E. Alba and J.M. Troya. Gaining new fields of application for OOP: the parallel evolutionary algorithm case. *Journal of Object Oriented Programming*, December (web version only) 2001.
17. V.S. Gordon and D. Whitley. Serial and parallel genetic algorithms as function optimizers. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 177–183. Morgan Kaufmann, 1993.
18. E. Alba and J.M. Troya. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems*, 17:451–465, January 2001.
19. E. Alba. Parallel evolutionary algorithms can achieve superlinear performance. *Information Processing Letters*, 82(1):7–13, April 2002.
20. D. Whitley. Cellular genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 658. Morgan Kaufmann Publishers, San Mateo, California, 1993.