Evolutionary Algorithms for Optimal Placement of Antennae in Radio Network Design

E. Alba
Departamento de Lenguajes y Ciencias de la Computación
Campus de Teatinos, E.T.S.I. Informática, 3-2-12
University of Málaga, 29071 Málaga
Spain
e-mail: eat@lcc.uma.es.

Abstract

In this paper, 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 in order to cope with the high computational requirements of such a problem. Here, we try to develop and evaluate a set of sequential and parallel genetic algorithms (GAs) in order to solve efficiently the RND problem. 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, with an observed linear speedup.

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 [11], predicting bandwidth demands in ATM networks [15], developing error correcting codes for transmission of messages [8], and designing the telecommunication network [9, 12, 13, 18].

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 [7]. 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 paper, our goal is to analyze different kinds of genetic algorithm to find out which of them is more suited to solve the RND problem efficiently and accurately. In general, evolutionary algorithms encode tentative problem solutions in a population of individuals with an associated fitness (quality) value and then *evolve* it 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 pools of solutions. Also, the fitness function is a capital feature 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 an efficient parallel algorithm that reduces the search time and, at the same time, that maximizes the quality of the solutions worked out.

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 report how our results relate to existing studies in literature 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 not connex. 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 all potential transmitter locations. Let G be the graph, $(M \bigcup L, E)$, where E is a set of edges such that each transmitter location is linked to the locations it covers and let be the vector $\overrightarrow{x}$ a solution to the problem where $x_i \in \{0, 1\}$, and $i \in [1, 149]$ 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 Figure 1.

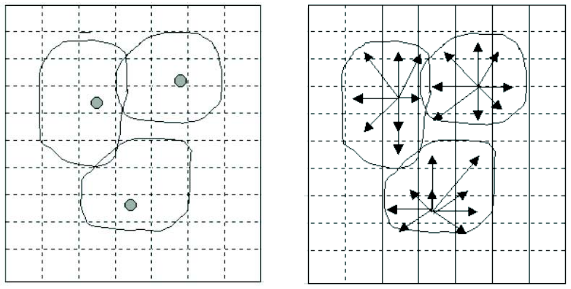

Figure 1. (left) Three potentially transmitter locations and their associated covered cells on a grid, and (right) graph representing covered locations.

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 $\mid M' \mid$ is minimum and such that $\mid Neighbors(M', E) \mid$ is maximum, where

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

$$M' = \{t \in M \mid x_t = 1\} \quad (2)$$

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: $M' = \phi$. If maximizing the number of covered locations was the primary goal, then problem would be the USCP. An objective function $f(\overrightarrow{x})$ to combine the two goals has been proposed in [7]:

$$f(\overrightarrow{x}) = \frac{CoverRate(\overrightarrow{x})^{\alpha}}{Number\ of\ transmitters\ selected(\overrightarrow{x})} \quad (3)$$

where

$$CoverRate(\overrightarrow{x}) = 100 \cdot \frac{Neighbors(M', E)}{Neighbors(M, E)} \quad (4)$$

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 in [7], we will use $\alpha = 2$, a 287×287 point grid representing an open-air flat area. Also, 49 primary transmitter locations are 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 locations. A hundred complementary transmitters locations were then randomly selected, associated to 41×41 point cells (fewer when clipped by the border of the area), and shuffled with the 49 primary ones (to avoid any kind of bias in the recombination phase of our algorithms). By construction, the best solution is the one that covers 100% of the area with the 49 primary transmitters (giving a fitness value of 204.08). See in Figure 2 the graphical representation of a partial (left) and optimal (right) solution for the RND problem.

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 pseudocode) 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

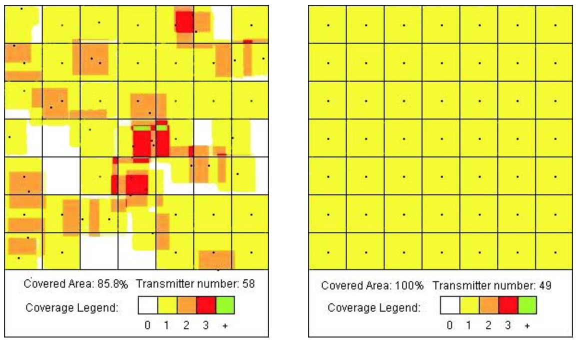
Covered Area: 85.8% Transmitter number: 58
Coverage Legend:
0 1 2 3 +
Covered Area: 100% Transmitter number: 49
Coverage Legend:
0 1 2 3 +

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

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

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;
```

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 [2].

There exist two quite popular classes of panmicitic EAs having different granularity at the reproductive step [16]. The first one is called "generational" model, in which a whole new population of λ individuals replaces the old one (right part of Figure 3, 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.

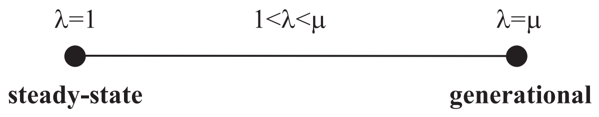

Figure 3. Panmictic EAs: from steady-state to generational algorithms.

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 recombination and/or mutation are parallelized), while still using a centralized selection performed sequentially in the main processor guiding the base algorithm [14]. 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 [2].

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 [6, 17]. In cellular EAs, the existence of overlapped small neighborhoods helps in exploring the search space [5]. 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 [4, 10].

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 kind of distributed EA being used. 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 their 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 new 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 (Equation 5) 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 Equation 6 the maximized function, by which we maximize the covered area and, at the same time, we penalize solutions having a large number of transmitters.

$$Evaluate1(\overrightarrow{x}) = f(\overrightarrow{x}) \tag{5}$$

$$Evaluate1(\overrightarrow{x}) = \frac{CoverRate(\overrightarrow{x})^{\alpha}}{Number\ of\ transmitters\ selected} \tag{6}$$

This first evaluation function was also used in [7], working out a considerably high time before computing a solution. Therefore, we will test 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 an optimal solution for this problem should not leave any point in the grid without coverage, and this means that the mean number of transmitters covering a single point should be 1.0. This is computed in a penalty term P_m explained in Equation 7. See in Table 2 the meaning of the symbols used in the equations.

$$P_m(\overrightarrow{x}) = \frac{\Phi}{100} \cdot (\bar{t}(\overrightarrow{x}) \cdot a_m + b_m) \tag{7}$$

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 Equation 8.

$$P_v(\overrightarrow{x}) = \frac{\Phi}{100} \cdot (\sigma_t^2(\overrightarrow{x}) \cdot a_v + b_v) \tag{8}$$

We have called these two terms *penalty* values since we plan to penalize the first evaluation function with they two in order to engineer the algorithm with a more accurate guide along the search space. See in Equation 9 the resulting evaluation function intended to be maximized by the analyzed algorithms.

$$Evaluate2(\overrightarrow{x}) = Evaluate1(\overrightarrow{x}) - P_m(\overrightarrow{x}) - P_v(\overrightarrow{x}) \quad (9)$$

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 in a smaller search time (despite the overload of computing additional statistics in this function).

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.

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

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

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

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
$\overrightarrow{x}$	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 $f(\overrightarrow{x})$
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 $(a_v = 0.01, b_v = 2.704)$

Table 2. Meaning of the symbols.

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, two-point crossover (tpx) with probability 1.0, bit-flip mutation with probability $1/strlen$, and replacement always of the worst string.

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 [3]. We will run the algorithms both in one single CPU (i.e., concurrently) and on 8 processors. Each processor is an Ultra Sparc 1 CPU at 143 MHz linked by an ATM communication network. Although it is not a cutting-edge cluster platform, we plan to extend and compare the results against a clusters or Pentium 4, and against a beowulf of machines aged differently. Thus, this work is a first step (the most important milestone) for the upcoming works in next months. Also, see a summary of the conditions for experimentation in Table 3. We perform 30 independent runs of each experiment, what represents a considerable time since, as we will see, some algorithms will need several hours to locate a solution.

Population Size	*512*
Selection	*roulette wheel*
Crossover	*tpx prob = 1.0*
Mutation	*bit-flip prob = 0.00671*
Replacement	*least fitted*
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*

Table 3. Parameters of the algorithms being used.

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

	Evaluate1		*Evaluate2*	
	# evals	*time (h)*	*# evals*	*time (h)*
dssGA (8 CPUs)	505624	1.38	579922	3.53
dssGA (1 CPU)	491496	10.87	573175	28.41
ssGA	173013	3.86	169016	8.49

Table 4. Number of evaluations (left) and time (right) with ssGA and dssGA for Evaluate1 and Evaluate2.

To asses 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 Table 4 we can notice several facts. Firstly, for Evaluate1, it is clear that the sequential ssGA is the best algorithm numerically, since it samples almost 3 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 dssGA heuristic on 8 processors, since it performs very quickly (1.38 hours) in comparison with the ssGA (3.86 hours), and the dssGA on a single processor (10.87 hours). This comes as no surprise since numerically, ssGA was better, although dssGA on 8 processors makes computations much faster. Since it is not fair to compute speedup against a panmictic ssGA [1] we compare the same algorithm, both in sequential and parallel (dssGA 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 result is to find out that speedup is 7.87, i.e. near linear speedup, which is very satisfactory.

For the second fitness function (see Table 4 again) we can notice that the results are much the same: ssGA is numerically advantageous with respect both execution scenarios of dssGA (p-values well below 0.05). These two last algorithms seem again to provide the same average number of evaluations (differences are not significant, since the t-test offers a p-value clearly larger than 0.05, the significance level). Again, the execution time relationship is just like with the first evaluation function (in fact, speedup is perfect, even slightly superlinear: 8.04).

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 do not reduce the number of sampled points before getting a solution significantly, despite we expected it. In addition, the execution time of all the algorithms is clearly enlarged 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 conclusion. See a graphical interpretation of the results in Figure 4, were we plot the average number of evaluations and execution time for ssGA and for the serial and parallel versions of dssGA.

From our results, it is clear that the two dssGA scenarios are similar in number of evaluations (p-value=0.738) while ssGA is clearly giving the best average evaluation number (t-tests much below 0.05). The set of t-tests for time values make us conclude that all the differences in time are significant, which supports our claims of a good speedup for the two evaluation functions.

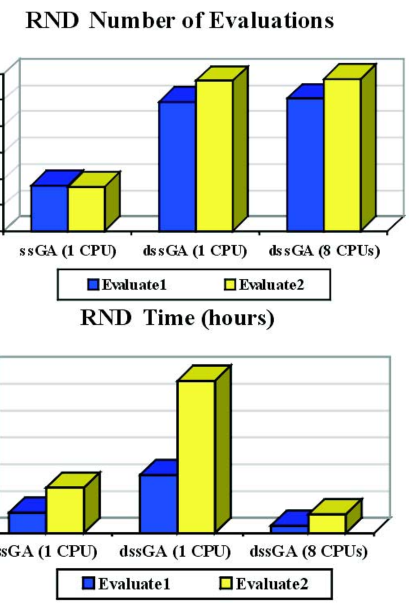

Figure 4. Number of evaluations (up) and time (down) needed by (d)ssGA to solve RND.

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. In [7] the authors do not solve them 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 here consider solving the problem at 100% and also 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 tuning steps.

Let us begin with a detailed comparison between using Evaluate1 and Evaluate2 when the problem is solved 100%. The results (see Table 5) clearly indicate that there is a numeric advantage for the first evaluation function, both for the sequential execution of dssGA (p-value=0.0502) and for its parallel distributed execution (p-value slightly above 0.05, i.e. 0.0867).

As to the execution time when the 100% case is being considered, we can notice in Table 6 that there is a clear advantage for the simplest first fitness function (p-values much below 0.05). Another conclusion is that speedups confirm we are using a good parallel implementation (almost linear —7.87— and slightly superlinear —8.04—), as we explained in the previous section.

# evals	*Evaluate1*	*Evaluate2*	*t-test*
dssGA (1 CPU)	491496	573175	0.0502
dssGA (8 CPUs)	505624	579922	0.0867

Table 5. Average number of evaluations for *RND* **problem** *dssGA***.**

Time (hours)	*Evaluate1*	*Evaluate2*	*t-test*
dssGA (1 CPU)	10.87	28.41	$4.930e^{-16}$
dssGA (8 CPUs)	1.38	3.53	$3.461e^{-12}$
speedup	7.87	8.04	

Table 6. Average time of execution for *RND* **problem** *dssGA***.**

Let us now turn to consider the case in which the problem is solved partially, at a 61% fitness value of the optimum. The interest of such tests comes from the practical application of the algorithm by an engineer, that usually is interested in a good guess in a small time. The results in [7] show a required time of 540 minutes, although our algorithm makes the same work in only 6.41 minutes. However, we need to remark that our disposition of the regular transmitters in the string is slightly different from that they used, which can be one reason for this difference.

In fact, our implementation is also faster than the one they used on the average, since they needed 540 minutes for making 40,000 evaluations in one processor (74 evaluations per minute), while we can perform 4287 evaluations in 6.41 minutes (669 evaluations per minute). This may be due, of course, to the fact that we are using different machines (not specially fast in our case), 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 confirms that the first evaluation function is the best in reducing the number of visited points.

	# evals	*Time (min.)*	*Fitness*
dssGA (1 CPU)	3940	6.78	127.19
dssGA (8 CPUs)	4021	0.85	126.79

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

The speedup (when solving the problem at 61%) between the sequential and parallel dssGA is 7.97 for Evaluate1 and 9.07 for Evaluate2. This result confirms that, while Evaluate1 is approaching linear speedup, Evaluate2 is better in profiting from a larger number of processors, which is generally considered an advantage in parallel programming. However, Evaluate1 is that 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.

	# evals	*Time (min.)*	*Fitness*
dssGA (1 CPU)	5128	21.32	127.14
dssGA (8 CPUs)	4094	2.35	128.45

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

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 [7] 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. The problem 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 one hour.

Since there exist many possible instances for this same problem, we are working in more difficult scenarios, and of course, we are exploring the use of some other kinds of evolutionary algorithms, specially cellular GAs [19] and multiobjective EAs. A clear improvement proceeds consisting in utilizing additional hardware platforms to know more on the generalization of the results presented in this paper.

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). The author also thanks M. Vallejo for his help.

References

[1] E. Alba. Parallel evolutionary algorithms can achieve superlinear performance. *Information Processing Letters*, 82(1):7–13, April 2002.

[2] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, October 2002.

[3] E. Alba and J. M. Troya. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems*, 17:451–465, January 2001.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] H. Chen, N. Flann, and D. Watson. Parallel genetic simulated annealing: A massively parallel SIMD algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):126–136, 1998.

[9] C. Chu, G. Premkumar, and H. Chou. Digital data networks design using genetic algorithms. *European Journal of Operational Research*, 127:140–158, 2000.

[10] 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.

[11] A. Kapsalis, V. Rayward-Smith, and G. Smith. Using genetic algorithms to solve the radio link frequency assigment problem. In D. Pearson, N. Steele, and R. Albretch, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 37–40. Springer-Verlag, 1995.

[12] N. Karunanithi and T. Carpenter. Sonet ring sizing with genetic algorithms. *Computers and Operations Research*, 24(6):581–591, 1997.

[13] 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.

[14] 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.

[15] N. Swaminathan, J. Srinivasan, and S. Raghavan. Bandwidth-demand prediction in virtual path in atm networks using genetic algorithms. *Computer Commnunications*, 22(12):1127–1135, 1999.

[16] 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.

[17] 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.

[18] 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.

[19] 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.