CAGE : A Tool for Parallel Genetic Programming Applications

Gianluigi Folino, Clara Pizzuti and Giandomenico Spezzano

ISI-CNR
c/o DEIS, Via P. Bucci Cubo 41C,
Univ. della Calabria
87036 Rende (CS), Italy
{folino,pizzuti,spezzano}@si.deis.unical.it

Abstract. A new parallel implementation of genetic programming based on the cellular model is presented and compared with the island model approach. Although the widespread belief that cellular model is not suitable for parallel genetic programming implementations, experimental results show a better convergence with respect to the island approach, a good scale-up behaviour and a nearly linear speed-up.

1 Introduction

The capability of genetic programming (GP) in solving hard problems coming from different application domains has been largely recognized. Many problems have been succesfully solved by means of GP. It is well known that, in order to find a solution, the evaluation of the fitness is the dominant consuming time task for GP and evolutionary algorithms in general. The success of GP, furthermore, often depends on the use of a population of sufficient size. The choice of the size is determined by the level of complexity of the problem. When applied to large hard problems GP performances may thus drastically degrade because of the computationally intensive task of fitness evaluation of each individual in the population. In the last few years there has been an increasing interest in realizing high performance GP implementations to extend the number of problems GP can cope with. To this end different approaches to parallelize genetic programming have been studied and proposed [3–7, 11, 12, 15]. Extensive surveys on the subject can be found in [2, 18].

In this paper we present a tool for parallel genetic programming applications, called $CAGE$ (*CellulAr GEneting programming tool*), that realizes a fine-grained parallel implementation of genetic programming on distributed-memory parallel computers. Experimental results on some classical test problems shows that the cellular model outperforms both the sequential canonical implementation of GP and the parallel island model. Furthermore parallel cellular GP has a nearly linear speed-up and a good scale-up behaviour.

The paper is organized as follows. In section 2 a brief overview of the main parallel implementations proposed is given. In section 3 the cellular parallel implementation of GP is presented. In section 4 we show the results of the method on some standard problems.

2 Parallel Genetic Programming

Two main approaches to parallel implementations of GP have been proposed : the *coarse-grained* (*island*) model [10] and the *fine-grained* (*grid*) model [13].

The island model divides the population into smaller subpopulations, called *demes*. A standard genetic programming algorithm works on each deme and is responsible for initializing, evaluating and evolving its own subpopulation. The standard GP algorithm is augmented with a *migration* operator that periodically exchanges individuals among the subpopulations. How many individuals migrates and how often migration should occur are parameters of the method that have to be set [2, 18].

In the grid model (also called cellular [19]) each individual is associated with a spatial location on a low-dimensional grid. The population is considered as a system of active individuals that interact only with their direct neighbors. Different neighborhoods can be defined for the cells. The most common neighborhoods in the two-dimensional case are the 4-neighbor (*von Neumann neighborhood*) consisting of the North, South, East, West neighbors and 8-neighbor (*Moore neighborhood*) consisting of the same neighbors augmented with the diagonal neighbors. Fitness evaluation is done simultaneously for all the individuals and selection, reproduction and mating take place locally within the neighborhood. Information slowly diffuses across the grid giving rise to the formation of semi-isolated niches of individuals having similar characteristics.

In [18] it is noted that parallel evolutionary algorithms benefit of the multipopulation approach since the same solution quality can be obtained by using many populations instead of a single population with the same total number of individuals. The same results, however, have not been obtained for the coarse grained parallel implementations of genetic programming. In fact, though Koza [1, 9] reported a super-linear speedup for the 5-parity problem, in [14], for the ant and the royal-tree problems, Punch found poorer results of convergence with respect to the canonical *GP*. In the next section we present a parallel implementation of *GP* through the cellular model. Afterwards, we show that such an approach gives better convergence results with respect to both canonical and island model implementations of *GP*.

3 Parallel implementation of CAGE

This section describes the implementation of *CAGE* on distributed-memory parallel computers. To parallelize *GP CAGE* uses the cellular model. The cellular model is fully distributed with no need of any global control structure and is naturally suited to be implemented on parallel computers. It introduces fundamental changes in the way *GP* works. In this model, the individuals of the population are located on a specific position in a toroidal two-dimensional grid and the selection and mating operations are performed, cell by cell, only among the individual assigned to a cell and its neighbors. This local reproduction has the effect of introducing an intensive communication among the individuals that allows to disseminate good solutions across the entire population, but that influences the performance of the parallel implementation of GP. Moreover, unlike genetic algorithms, where the size of individuals is fixed, the genetic programs are individuals of varying sizes and shapes. This requires a large amount of local memory and introduces an unbalanced computational load per grid point. Therefore, an efficient representation of the program trees must be adopted and a load balancing algorithm must be employed to maintain the same computational load among the processing nodes.

The best way to overcome the drawbacks associated with the implementation of the cellular model on a general purpose distributed-memory parallel computer is to use a partitioning technique based upon domain decomposition in conjunction with the *Single-Program-Multiple-Data* ($SPMD$) programming model. According to this model, an application on N processing elements (PEs) is composed of N similar processes, each of which operates on a different set of data. For an effective implementation, data should be partitioned in such a way that communication takes place locally and the computation load be shared among the PEs in a balanced way. This approach increases the granularity of the cellular model transforming it from a fine-grained model to a coarse-grained model. In fact, instead of assigning only one individual to a processor, the individuals are grouped by *slicing up* the grid and assigning a *slice* of the population to a node.

$CAGE$ implements the cellular GP model using a one-dimensional domain decomposition (in the x direction) of the grid and an explicit message passing to exchange information among the domains. This decomposition is more efficient than a two-dimensional one. In fact in the two-dimensional decomposition the number of messages sent is higher, though the size of the messages is lower. On the other hand, in one-dimensional decomposition, the number of messages sent is lower but their size is higher. Considering that the startup times are much greater than the transfer times, the second approach is more efficient than the former. The concurrent program which implements the architecture of $CAGE$ is composed of a set of identical *slice processes*. No coordinator process is necessary because the computational model is completely decentralized. Each slice process, which contains a strip of elements of the grid, runs on a single processing element of the parallel machine and executes the code, shown in figure 1, on each subgrid point thus updating all the individuals of the sub-population.

Each slice process uses the parameters read from a file (step 1) to configure the genetic programming algorithm that has to be executed on each subgrid point. The parameters concern the population size, the max depth that the trees can have after the crossover, the parsimony factor, the number of iterations, the number of neighbors of each individual, the replacement policy. We have implemented three replacement policies: *direct* (the best of the offspring always replaces the current individual), *greedy* (the replacement occurs only if offspring

```
1.    Read from a file the configuration parameters
2.    Generate a random sub-population
3.    Evaluate the individuals of the sub-population
4.    while not numGenerations do
5.      update boundary data
6.      for x =1 to length
7.        for y =1 to height
8.          select an individual k (located at position [x',y'])
            neighboring with i (located at position [x,y]);
9.          generate offspring from i and k ;
10.           apply the user-defined replacement policy to update i;
11.           mutate i with probability pmut;
12.           evaluate the individual i;
        end for
      end for
    end while
```

Fig. 1. Pseudocode of the slice process.

is fitter), *probabilistic* (the replacement happens according to difference of fitness between parent and offspring (*simulated annealing*)).

The size of the subpopulation of each slice process is calculated by dividing the population for the number of the processors on which CAGE is executed. Each slice process updates sequentially the individuals belonging to its subgrid. Initially, in each process, a random subpopulation is generated (step 2.) and its fitness is evaluated (step 3.). Then, steps 6-12 are executed for generating the new subpopulation for *numGeneration* iterations. The variables *lenght* and *height* define the boundaries of the 2D subgrid that is contained in a process. It should be noted that two copies of the data are maintained for calculating the new population. In fact, as each element of the current population is used many times, the current population cannot be overwritten.

Because of the data decomposition, physically neighboring strips of data are allocated to different processes. To improve the performances and to reduce the overhead due to the remote communications, we have introduced a local copy of boundary data in each process. This avoids to perform remote communication more than once on the same data. Boundary data are exchanged before applying the selection operator. In our implementation, the processes form a logical ring and each processor determines its right and left neighboring processes. Therefore, the communication between processes is local, only the outermost individuals are to be communicated between the slice processes.

All the communications are performed using the *MPI* (*Message Passing Interface*) portable message passing system so that *CAGE* can be executed across different hardware platforms. Since the processes are connected according to a ring architecture and each process has a limited buffer for storing boundary data, we use asynchronous communication in order to avoid processors to idle.

Each processor has two send buffers ($SRbuf, SLbuf$) and two receive buffers ($RRbuf$, $RLbuf$). The SRbuf and SLbuf buffers correspond to the outermost (right and left) individuals of the subgrid. The receive buffers are added to the subgrid in order to obtain a bordered grid. The exchange of the boundary data occurs, in each process, by two asynchronous send operations followed by two asynchronous receive operation to the right and left neighboring processes. After this, each process waits until the asynchronous operations are completed.

$CAGE$ uses the standard tool for genetic programming sgpc1.1, a simple GP in the C language, freely available at [16], to apply the GP algorithm to each grid point. However, in order to meet the requirements of the cellular GP algorithm, a number of modifications have been introduced.

We used the same data structure of sgpc1.1 to store a tree in each cell. The structure that stores the population has been transformed from a one-dimensional array to a two-dimensional one and we duplicated this structure in order to store the current and the new generated tree. The selection procedure has been replaced with one that uses only the neighborhood cells and the three replacement policies have been added. Crossover is performed between the current tree and the best tree in the neighborhood. Two procedures to *pack* and *unpack* the trees that must be sent to the other processes have been added. The pack procedure is used to send the trees of the boundary data to the neighbor processes in a linearized form. Data are transmitted as a single message in order to minimize the message startup and transmission time. The unpack procedure rebuilds the data and stores them in the new processor's private address space.

The execution of a parallel program is composed of two phases: *computation* and *communication*. During the computation phase each process of the concurrent program implementing the run-time support executes computations which only manipulate data local to this process. These data can be local variables or boundary data received from neighboring processes. The effective data transmission between processes is done at the end of each local computation. This means that, during the computation phase, no data are exchanged.

In order to equally distribute the computational load among the processing nodes $CAGE$ introduces an intelligent partitioning of the grid. The partitioning strategy is a form of *block-cyclic* decomposition. The idea is to split the grid virtually in a number of folds and assign equal parts of each fold to each of the processes as shown in figure 2. This can lead to load balancing provided that the resulting granules (further referred to as strips) are fine enough to assure that the uneven load distribution across folds is statistically insignificant across processes. It should be noted, that the number of folds and processes should be chosen with caution, since the more strips are used, the bigger is the communication overhead among the processing elements. In the next section the experimental results obtained with our approach are presented.

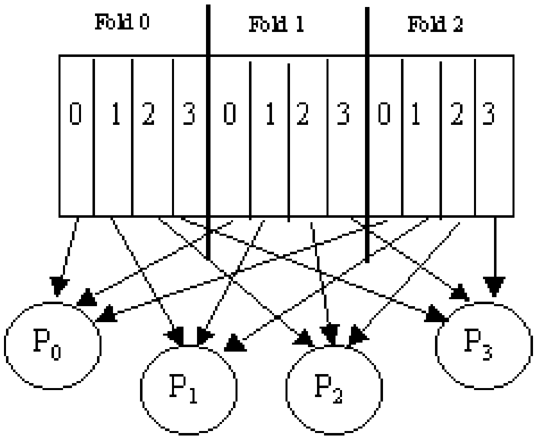


Fig. 2. Load balancing strategy: each fold is divided in four strips.

4 Experimental Results

This section shows the performances of our cellular parallel implementation on three test problems well known in the literature: *discovery of trigonometric identities, even-4-parity* and *artificial ants*. The parallel implementation has been realized on a multicomputer Meiko CS-2. For each problem we present the convergence results obtained with $CAGE$ and compare them with the sequential canonical GP and the island model implementation realized by [11] for the first two problems and the one reported in [14] for the latter problem. We used sgpc1.1 as the canonical sequential implementation of genetic programming. The parameters of the method are shown in table 1. Each problem was run 10 times. For all the experiments we used the $Moore$ neighborhood and the *direct* replacement policy.

Experiment 1. ([11]) The *symbolic regression* problem consists in searching for a non trivial mathematical expression that always has the same value of a given mathematical expression. In the experiment our aim was to discover a trigonometric identity for $cos2x$ [8]. 20 values x_i of the independent variable x have been randomly chosen in the interval $[0,2\pi]$ and the corresponding value $y_i = cos2x_i$ computed. The 20 pairs (x_i, y_i) constitute the fitness cases. The fitness is then computed as the sum of the absolute value of the difference between y_i and the value generated by the program on x_i. The maximum number of allowed generations has been set to 100.

Experiment 2. ([11]) The *even-4 parity* problem consists in deciding the parity of a set of 4 bits [8]. A Boolean function receives 4 Boolean variables and it gives true only if an even number of variables is true. Thus the goal function to discover is $f(x_1, x_2, x_3, x_4) = x_1x_2x_3x_4 \vee \overline{x_1}x_2\overline{x_3}x_4 \vee \overline{x_1}x_2x_3\overline{x_4} \vee x_1\overline{x_2}\overline{x_3}x_4 \vee x_1\overline{x_2}x_3\overline{x_4} \vee x_1x_2\overline{x_3}\overline{x_4} \vee \overline{x_1}\;\overline{x_2}\;\overline{x_3}\;\overline{x_4}$. The fitness cases are the 2^4 combinations of the variables. The fitness is the sum of the Hamming distances between the goal

Table 1. CAGE Parameters

Parameters	Problem Name		
	Regression	**Even-4 parity**	**Ant (Santa Fe)**
Terminal symbols	{X,1.0}	$\{d_0, d_1, d_2, d_3\}$	{Forward, Right, Left}
Functions	{+, -, *, %, sin}	{AND, OR NAND, NOR}	{IfFoodAhead Prog2, Prog3}
Population size	3200	3200	3200
Max_depth_for_new_tree	6	6	6
Max_depth_after_crossover	6	17	17
Max_mutant_depth	4	4	2
Grow_method	RAMPED	GROW	RAMPED
Crossover_func_pt_fraction	0.2	0.1	0.8
Crossover_any_pt_fraction	0.2	0.7	0.1
Fitness_prop_repro_fraction	0.1	0.1	0.1
Parsimony_factor	0.0	0.0	0.0

function and the solution found. The maximum number of allowed generations has been set to 100.

Experiment 3. ([14]) The *artificial ant* problem consists in finding the best list of moves that an ant can do on a 32×32 matrix in order to eat all the pieces of food put on the grid [8]. In this experiment we used the *Santa Fe trail* that contains 89 food particles. The fitness function is obtained by diminishing the number of food particles by one every time the ant arrives in a cell containing food. The ant can see the food only if it is in the cell ahead its same direction ($IfFoodAhead$ move) otherwise it can move randomly ($left$ or $right$) for two ($Progn2$) or three ($Progn3$) moves. The maximum number of allowed moves has been set to 500.

In figure 3 the convergence results of $CAGE$ with respect to canonical GP are shown for experiment 1 (a) and experiment 2 (b). The figure clearly shows that, after 100 generations, canonical GP is far from the solution whether $CAGE$ after an average of 90 generations is able to discover the trigonometric identity and almost always finds the correct Boolean function for even-4 parity after 100 generations. To compare our method with the island model implementation, in figures 3(c) and 3(d) the results obtained by Niwa and Iba [11] are reported. With regard to cos2x the better implementation they obtain (ring topology) does not always reach a solution after 100 generations, while for the 4-parity they can not find the correct Boolean function neither after 500 generations.

In figure 3(e) the convergence results for the *Ant* problem are shown. The exact solution is almost always found after 100 generations with a population size of 3200 individuals. In figure 3(f) the effect of the population size on the convergence of the method is showed. It can be noticed that the bigger is the size of the population, the lower is the number of generations necessary to find

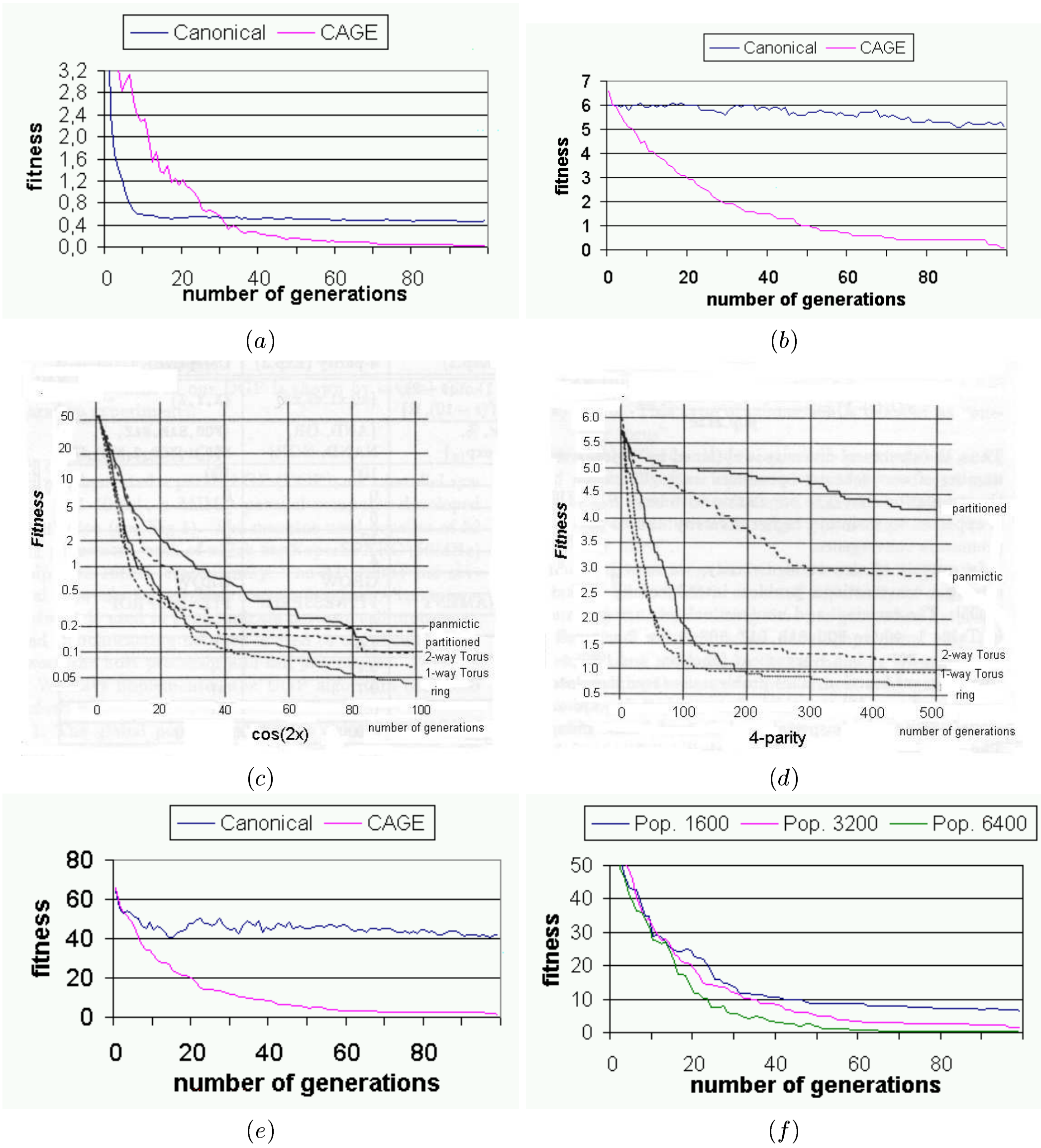
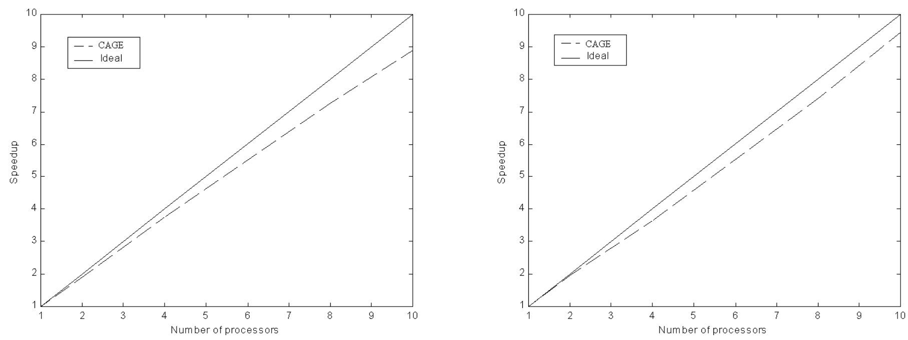


Fig. 3. Experimental results: CAGE for cos2x (a) and 4-parity (b), Niwa and Abi for cos2x (c)and 4-parity (d), CAGE for Ant (e) and Ant convergence for different population sizes.

the optimal solution. When the size is 6400, the solution is always found after 70 generations.

In [14] Punch presents the results of his experiments as the number of $Wins$ and $Losses$. The wins are denoted as $W:(x,y)$, where x represents the number of optimal solutions found before 500 generations and y the average generation in which the optimal solution was found. The losses are denoted as $L:(q,r,s)$, where q is the number of losses (no optimal solution found before 500 generations) r is the average best-of-run fitness, and s is the average generation when the best-of-run occurred. To compare our results with those of Punch, we computed the wins and losses by running $CAGE$ the same number of times (16) as Punch reported and the same population size (1000). The best result Punch obtained was $W:(7,240)$ and $L:(9,73,181)$. The result we obtained was $W:(8,98)$ and $L:(8,76,212)$ thus confirming the better performances of the $CAGE$ with respect to the island approach. Finally, in figure 4 the speed-up of the method is showed for experiments 1 and 2.



(a) (b)

Fig. 4. Speedup measures for cos2x (a) and even-4 parity (b).

5 Conclusions and Future Work

A tool for parallel genetic programming applications that realizes a fine-grained parallel implementation of genetic programming through cellular model on distributed-memory parallel computers has been presented. Preliminary experimental results shows the good performances of the proposed approach. We are planning an experimental study on a wide number of benchmark problems to substantiate the validity of the cellular implementation.

References

1. Andre D., Koza, J. R. Exploiting the fruits of parallelism: An implementation of parallel genetic programming that achieves super-linear performance. *Information Science Journal*, Elsevier, 1997.
2. Cantú-Paz, E. *A summary of research on parallel genetic algorithms*, Technical Report 950076, Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana Champaign, Urbana, July 1995.
3. Dracopoulos, D. C., Kent, S. Speeding up Genetic Programming: A Parallel BSP implementation. *Genetic Programming 1996, Proceedings of the First Annual Conference*, pp 125-136, MIT Press, Stanford University, July 1996.
4. Fernández, F., Tomassini, M., Punch, W. F., Sánchez, J.M. Experimental Study of Multipopulation Parallel Genetic Programming. *European Conference on Genetic Progamming*, LNCS 1082, Springer, Edinburgh 1999.
5. Fernández, F., Tomassini, M., Vanneschi, L., Bucher, L. A Distributed Computing Environment for Genetic Programming Using MPI. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting*, Balatonfured, Hungary, September 2000.
6. Juillé, H., Pollack, J. B. Parallel Genetic Programming on Fine-Grained SIMD Architectures. *Working Notes of the AAAI-95 Fall Symposium on Genetic Programming"*, AAAI Press, 1995.
7. Juillé, H., Pollack, J. B. Massively Parallel Genetic Programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming: Volume 2*, MIT Press, Cambridge, 1996.
8. Koza, J. R. Genetic Programming: On the Programming of Computers by means of Natural Selection. MIT Press, Cambridge, 1992.
9. J. R. Koza and D.Andre (1995) Parallel genetic programming on a network of transputers. *Technical Report CS-TR-95-1542, Computer Science Department*, Stanford University.
10. W.N. Martin, J. Lienig and J. P. Cohoon (1997), Island (migration) models: evolutionary algorithms based on punctuated equilibria, in T. Bäck, D.B. Fogel, Z. Michalewicz (eds.), *Handbook of evolutionary Computation.* IOP Publishing and Oxford University Press.
11. Niwa, T., Iba, H. Distributed Genetic Programming -Empirical Study and Analisys - *Genetic Programming 1996, Proceedings of the First Annual Conference*, MIT Press, Stanford University, July 1996.
12. Oussaidéne, M., Chopard, B. Pictet, O., Tommasini, M. Parallel Genetic Programming and its Application to Trading Model Induction. *Parallel Computing*, vol. 23, n. 2, September 1997.
13. C. C. Pettey (1997), Diffusion (cellular) models, in T. Bäck, D.B. Fogel, Z. Michalewicz (eds.), *Handbook of evolutionary Computation.* IOP Publishing and Oxford University Press.
14. Punch, W. F. How Effective are Multiple Populations in Genetic Programming. *Genetic Programming 1998, Proceedings of the Third Annual Conference*, MIT Press, University of Winsconsin, July 1998.
15. Salhi, A., Glaser, H., De Roure, D. Parallel Implementation of a Genetic-Programming based Tool for Symbolic Regression. *Technical Report DSSE-TR-97-3*, Dept. Comp. Science, University of Souhampton, 1997.
16. Tackett, W. A., Carmi, A. Simple Genetic Programming in C, *Available through the genetic programmming archive at ftp://ftp.io.com/pub/genetic-programming/code/sgpc1.tar.Z.*

17. T. Toffoli and N. Margolus (1986). *Cellular Automata Machines A New Environment for Modeling*. The MIT Press, Cambridge, Massachusetts.
18. Tomassini M. *Parallel and Distributed Evolutionary Algorithms: A Review*, J. Wiley and Sons, Chichester, K. Miettinen, M. Mkel, P. Neittaanmki and J. Periaux (editors), pp. 113-133, 1999.
19. D.Whitley (1993). Cellular Genetic Algorithms. *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann.