Serial and Parallel Genetic Algorithms as Function Optimizers

V. Scott Gordon
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523

Darrell Whitley
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523

Abstract

Parallel genetic algorithms are often very different from the "traditional" genetic algorithm proposed by Holland, especially with regards to population structure and selection mechanisms. In this paper we compare several parallel genetic algorithms across a wide range of optimization functions in an attempt to determine whether these changes have positive or negative impact on their problem-solving capabilities. The findings indicate that the parallel structures perform as well as or better than standard versions, even without taking parallel hardware into account.

1 INTRODUCTION

A variety of schemes for parallelizing genetic algorithms have appeared in the literature. Some of the parallel implementations are very different from the "traditional" genetic algorithm proposed by Holland (1975), especially with regards to population structure and selection mechanisms. It is reasonable to assume that structural changes in a genetic algorithm will affect it's ability to solve problems. If the changes result in slower problem-solving, then it might be necessary to determine how much parallelism would have to be achieved in order to compensate. Conversely, if the changes result in faster problem-solving, then the parallel models might be preferable even in the absence of parallel hardware.

We have coded nine different genetic algorithms representing simple versions of existing parallel models. Each algorithm is run on a variety of optimization problems, and the results are normalized by the total number of function evaluations. The goal is to determine the effectiveness of various population structures and selection mechanisms, regardless of parallel execution. Thus we do not attempt to optimize the algorithms, and we do not examine hybrid versions that include local optimization. All algorithms are coded in a functional language (Sisal) to facilitate a concurrent study of the dataflow parallelism in the algorithms (Gordon, Whitley, and Böhm 1992).

2 PARALLEL GENETIC ALGORITHM MODELS

Current literature on parallel genetic algorithm implementations can be separated into three categories: *global*, *island*, and *cellular genetic algorithms*. *Cellular genetic algorithms* have sometimes been called massively parallel genetic algorithms or fine grain genetic algorithms, but it can be shown that these algorithms are in fact a subclass of cellular automata (Whitley 1993). We have encoded four global models, four island models, and one cellular model:

Global Models:	SGA, Elitist-SGA, pCHC, Genitor
Island Models:	I-SGA, I-Elitist-SGA, I-pCHC, I-Genitor
Massively Parallel:	Cellular-GA

SGA and **Elitist SGA.** Our implementation of Goldberg's Simple Genetic Algorithm (SGA) (Goldberg 1989) uses tournament selection (Goldberg and Deb 1991) to facilitate parallelism. Every two slots of each new generation are filled by the offspring of two selected parents from the previous generation. Thus $n/2$ processes are utilized (where n is the population size), each of which generates two members of the next generation. In SGA the very best individual in a population may not survive. In the *Elitist SGA* a copy of the best individual is always placed in the next generation.

pCHC is a parallelized version of the CHC algorithm developed by Eshelman (1991). CHC is similar to SGA except that the best n strings are extracted from both the generation t and generation $t+1$ to form the new generation. Parents are paired through incest prevention. Thus, in our implementation, tournament selection is used not to select the fittest strings, but to select pairs of individuals which are relatively dissimilar (in Hamming distance). After recombination, the offspring in generation $t+1$ are compared against two particular elements from generation t, and the best two of the four are retained. This parallelization results in an algorithm very similar to CHC, but with a weaker selective pressure. Note that this algorithm does not guarantee that the best n out of the $2n$ individuals will survive, but it does guarantee that at least the best *two* individuals will survive. In its original form CHC also utilizes restarts, which we do not consider.

Genitor. Normally in the Genitor algorithm rank-based selection is applied to a sorted population. Two parents are selected and a single offspring is produced that displaces the worst member of the population. Tournament selection and replacement eliminates the need to keep the population in sorted order. The winners of a small tournament recombine and the offspring replaces the loser of the tournament if the offspring has a higher fitness. In other implementations of Genitor, one of the two possible offsprings is chosen randomly before evaluation. In the implementation used here both offspring are evaluated and the best of the two offspring is retained. Otherwise, Genitor (Whitley and Starkweather 1990) is coded in its original form.

Island SGA and **Elitist Island-SGA.** The Island model involves running several single population genetic algorithms in parallel. Each "island" is an SGA with its own subpopulation. Migration between islands uses a *ring* topology, and a single individual is chosen for migration by tournament selection, where the losing individual is replaced by the winning individual from the adjacent subpopulation. The Elitist Island-SGA involves a straightforward insertion of the Elitist-SGA model into the Island model (in place of SGA). Unlike Island-SGA, the *best* string in each subpopulation is the only one that migrates, since elitism guarantees that the best string is already known.

Island-pCHC and **Island-Genitor.** These are straightforward insertions of pCHC and Genitor into the Island model (in place of SGA). Migration is the same as in I-SGA. The Island-Genitor model is analogous to Genitor-II (Whitley and Starkweather 1990).

Cellular Genetic Algorithms. Cellular Genetic Algorithms assign one individual per processor, and mating is limited to a deme (neighborhood) near the individual. Each individual is processed in parallel at each generation. In our implementation, strings reside on a two-dimensional grid, and demes consist of the four individuals directly above, below, left, and right of each individual. The best of these four is selected and crossover is performed with the individual. The offspring replaces the original individual if it has a higher fitness. Edge elements wrap around, forming a torus. It can be shown that this type of cellular genetic algorithm is a finite cellular automaton with probabilistic rewrite rules, where the alphabet of the cellular automaton is equal to the number of strings in the search space (Whitley 1993).

Other details of these implementations were previously reported (Gordon, Whitley, and Böhm 1992).

3 PERFORMANCE MEASUREMENT

In order for performance comparisons to be meaningful across the nine genetic algorithm implementations, normalization is done so that the amount of work expressed per time unit is comparable. We use *number of function evaluations* as the base work unit, and express all computation times in terms of *SGA generations*. Performance measures for Genitor and the cellular genetic algorithm are adjusted as follows.

In the current implementation, Genitor performs two function evaluations per generation, but only one string is retained. Thus it requires $n/2$ (where n is the population size) generations in Genitor to perform the same number of function evaluations as one SGA generation. Therefore we divide the number of Genitor generations by $n/2$ when comparing results. Similarly, for Island-Genitor we divide the number of generations by $s/2$ (where s is the subpopulation size).

Our implementation of the cellular genetic algorithm performs two function evaluations for each location in the 2-D grid every generation. Thus in one generation it performs twice the number of function evaluations as SGA. Therefore we multiply the number of generations performed by the cellular genetic algorithm by two.

4 TEST SUITE

The following optimization problems have been coded: DeJong's original test suite F1-F5 (DeJong 1975); Rastrigin, Schwefel, and Griewangk functions (Mühlenbein, Schomisch, and Born 1991); Ugly 3 and 4-bit deceptive functions; and zero-one knapsack problems of various sizes. We perform no local optimizations, as we only wish to compare the effectiveness of the genetic algorithms.

We run all nine genetic algorithms across the test suite of functions for 30 runs. Some of the test problems are easy for the genetic algorithms and we execute until the global optimum is found in all 30 runs. In this case we report the average number of generations that it takes for each genetic algorithm to solve the function, and the standard deviation. For harder problems, we run the algorithms for a set number of generations and report the number of runs (out of 30) in which the global optimum is found, along with the average fitness of the best strings found at the end of each of the 30 runs. Convergence graphs for the Rastrigin, Schwefel,

Generations to solve (*gen*) and standard deviation (*std*) on DeJong's Test Suite for various genetic algorithms										
function	F1		F2		F3		F4		F5	
algorithm	gen	std	gen	std	gen	std	gen	std	gen	std
SGA	30.7	7.4	284	198	16.7	4.2	161	40	14.6	4.4
ESGA	28.9	6.8	83	55	15.3	4.1	153	49	14.3	4.4
pCHC	28.4	6.5	153	139	16.9	3.7	223	104	16.0	3.9
Genitor	17.0	4.1	190	160	8.2	2.1	135	67	7.9	2.5
I-SGA	41.3	11.2	417	253	22.0	5.3	405	192	20.3	6.9
I-ESGA	32.3	7.6	81	40	18.3	5.0	375	197	13.8	4.7
I-pCHC	33.2	7.4	78	57	18.8	4.4	495	239	16.3	5.3
I-Genitor	23.2	5.3	112	94	12.3	3.6	208	162	11.2	3.7
Cellular	32.5	8.0	105	94	17.9	4.6	397	204	15.3	4.3

Table 1: Performance of nine GAs on DeJong's test suite

and Griewangk functions are given in appendix A.

Population size is fixed at 400 for all algorithms and all problems. Other parameters are adjusted to try to maximize speed. In all of our implementations we employ two-point reduced surrogate crossover and next-point mutation (determining the next bit to be mutated rather than calculating mutation for every bit). Swap intervals for the island models are set to 5 generations for easy problems and 50 generations for hard ones. All functions are converted to minimization problems.

4.1 DeJONG TEST SUITE

DeJong's suite contains five test functions (DeJong 1975). F1 is a unimodal function known to be easy for genetic algorithms. F2 is a harder multimodal function. F3 is a discontinuous "step ladder". F4 involves a large solution space (2^{240}) plus gaussian noise. Because of the presence of noise, we consider F4 solved when the best string in the population reaches -2.5. F5 is characterized by the presence of several local minima. We found it useful to use Gray coding on F1, F2, and F5.

Performance results on the DeJong test suite are shown in Table 1. For the island models the swap interval is 5, except F4 where the swap interval is 50.

4.2 RASTRIGIN, SCHWEFEL, AND GRIEWANGK FUNCTIONS

Rastrigin's function (F6) is a fairly difficult problem for genetic algorithms due to the large search space and large number of local minima. Schwefel's function (F7) is somewhat easier than Rastrigin's function, and is characterized by a second-best minimum which is far away from the global optimum. In F7, V is the negative of the global minimum, which is added to the function so as to move the global minimum to zero, for convenience. The exact value of V depends on system precision; for our experiments $V = 4189.829101$. Griewangk's function (F8) is difficult for genetic algorithms because the product term causes the 10 substrings to be strongly interdependent.

$$F6\!: f(x_i|_{i=1,20}) = 200 + \sum_{i=1}^{20} x_i^2 - 10cos(2\pi x_i), \quad x_i \in [-5.12, 5.12]$$

$$F7\!: f(x_i|_{i=1,10}) = V + \sum_{i=1}^{10} -x_i sin(\sqrt{|x_i|}), \quad x_i \in [-512, 512]$$

$$F8\!: f(x_i|_{i=1,10}) = \sum_{i=1}^{10} \frac{x_i^2}{4000} - \prod_{i=1}^{10} cos(\frac{x_i}{\sqrt{i}}) + 1, \quad x_i \in [-512, 512]$$

Table 2 shows performance data on F6, F7, and F8 after 1000 generations. We used Gray coding for F6 and F8, and a swap interval of 50 for the island models. F7 can also be solved faster using Gray coding, but the results below are without Gray coding.

Number of runs solved (*ns*) and average best of 30 runs (*avg*) after 1000 generations. for Rastrigin (F6), Schwefel (F7), and Griewangk (F8) functions.						
function	F6		F7		F8	
algorithm	ns	avg	ns	avg	ns	avg
SGA	0	6.8	0	17.4	0	.161
ESGA	2	1.5	16	17.3	1	.107
pCHC	23	0.3	15	5.9	0	.072
Genitor	0	7.9	20	13.2	3	.053
I-SGA	0	3.8	9	6.5	7	.050
I-ESGA	13	0.6	13	2.6	3	.066
I-pCHC	10	0.9	28	0.2	3	.047
I-Genitor	23	0.2	24	0.9	6	.035
Cellular	24	0.2	26	0.7	1	.106

Table 2: Performance of GAs on F6, F7, and F8

4.3 UGLY 3 AND 4-BIT DECEPTIVE FUNCTIONS

The ugly 3-bit problem (D3) is a 30-bit artificially-constructed problem introduced by Goldberg, Korb, and Deb (1989) in which ten fully-deceptive 3-bit subproblems are interleaved. In general, the three bits of each subproblem X appear in positions X, 10+X, and 20+X. The ugly 4-bit problem (D4) is a similarly constructed 40-bit problem in which ten fully-deceptive 4-bit subproblems are interleaved (Whitley 1991).

Ugly deceptive problems have very often been misunderstood. These problems isolate interactions in the hyperplane sampling abilities of a genetic algorithm as well as the linkage between bits. This linkage is related to the disruptive effects of crossover. As such, ugly deceptive problems should perhaps be viewed more as analytical tools rather than "hard test problems." Mühlenbein (1992) has presented results which indicate such problems can often be solved by hill-climbing; this does not contradict the goals behind the design of fully deceptive problems, since they were not designed to mislead hill-climbing algorithms.

The original ugly deceptive problems (Goldberg, Korb, and Deb 1989) were used to test genetic algorithm implementations that use *no mutation* and a 100% probability of crossover. Whitley, Das, and Crabb (1992) have shown analytically that using low crossover rates also makes ugly deceptive problems easier to solve (since it reduces the effects of the linkage problem). Also, ugly deceptive problems are particularly designed to mislead traditional simple genetic algorithms of the kind developed by Holland and DeJong and described by Goldberg (1989). It does not automatically follow that ugly fully deceptive problems are generally hard for other algorithms, or even for other forms of genetic algorithms.

In the current set of experiments, *all* of the algorithms tested solve the ugly order-3 deceptive problem. SGA, for example, reliably solves the ugly order-3 deceptive problem using a crossover rate of 0.7 and a mutation rate of 1.5%. Further, parallel genetic algorithms appear to be particularly well suited to solving ugly deceptive functions. This is because different subproblems are solved in various subpopulations (or virtual neighborhoods). The various subproblems can then be exploited to build up full solutions.

Table 3 shows performance data on the 3-bit and 4-bit deceptive functions. The 3-bit problem is easy enough that we can execute all runs until the global optimum is found. We report the average number of generations in which the optimum is found, and the standard deviation. The 4-bit problem, on the other hand, is harder. We execute each run for 5000 generations and report the number of runs in which the global optimum was found, along with the average of the best strings found at the end of each run. It is important to note that performance data is reported differently for the two problems. We used a swap interval of 5 for the 3-bit problem and 50 for the 4-bit problem on the island models.

Generations to solve (*gen*) and standard deviation (*std*) for Ugly 3-bit; Runs solved (*ns*) and avg best of 30 runs (*avg*) for Ugly 4-bit (5000 gens)				
	D3		D4	
	gen	std	ns	avg
SGA	1547	675	0	23.5
ESGA	1564	1549	0	10.1
pCHC	455	192	14	1.3
Genitor	399	133	10	1.5
I-SGA	944	273	0	24.1
I-ESGA	172	49	29	0.07
I-pCHC	345	139	12	1.7
I-Genitor	439	158	6	2.3
Cellular	572	240	7	2.4

Table 3: Performance of nine GAs on D3 and D4

4.4 ZERO-ONE KNAPSACK PROBLEMS

The zero-one knapsack problem is defined as follows. Given n objects with positive weights W_i and positive profits P_i, and a knapsack capacity M, determine a subset of the objects represented by a bit vector X such that:

$$\sum_{i=1}^{n} X_i W_i \leq M \quad and \quad \sum_{i=1}^{n} X_i P_i \quad maximal.$$

A simple genetic encoding scheme for zero-one knapsack problems is as follows. Let each bit represent the inclusion or exclusion of one of the n objects from the knapsack. This way, a bit string of length n can be used to represent candidate solutions. The difficulty with this representation is that it is possible to represent infeasible solutions. In other words, setting too many bits might overflow the capacity of the knapsack.

We consider two methods of handling overflow. The first *penalty* method assigns a penalty equal to the amount of overflow. The second method, *partial scan*, adds items to the knapsack one at a time, scanning the bitstring left to right, stopping at the end of the string or when the knapsack overflows, in which case the last item added is removed.

A *greedy approximation* to the global optimum can be found by selecting objects by profit/weight ratio until the knapsack cannot be filled any further. If the objects are sorted by profit weight ratio, then the greedy approximation appears as a series of "1" bits followed by a series of "0" bits. The partial scan method now has the interesting property that a string of all "1" bits always evaluates to the greedy approximation.

Algorithm	F1	F2	F3	F4	F5	F6	F7	F8	D3	D4	K20	K80	Avg	rnk
SGA	5	8	4	3	5	8	9	9	8	8	8	3	6.5	8
ESGA	4	3	3	2	4	6	8	8	9	7	7	9	5.8	7
pCHC	3	6	5	5	7	3	5	6	5	2	3	4	4.5	4
Genitor	1	7	1	1	1	9	7	4	3	3	1	1	3.3	2
I-SGA	9	9	9	8	9	7	6	3	7	9	6	8	7.5	9
I-ESGA	6	2	7	6	3	4	4	5	1	1	9	5	4.4	3
I-pCHC	8	1	8	9	8	5	1	2	2	4	5	7	5.0	5
I-Genitor	2	5	2	4	2	2	3	1	4	5	2	2	2.8	1
Cellular	7	4	6	7	6	1	2	7	6	6	4	6	5.2	6

Table 4: Ranking of performance of nine GAs on test suite

For this study we generated a 20-object problem and an 80-object problem. (Branch-and-bound methods are known to solve much larger knapsack problems, so our results are useful only for comparing the various genetic algorithms.) Our 20-object problem (K20) has a global optimum of 445 and a greedy approximation of 275. The 80-object problem (K80) has a global optimum of 25729 and a closer greedy approximation of 25713. Using our set of nine genetic algorithms, the 20-object knapsack problem is *harder* to solve than the 80-object knapsack problem. Further, the *penalty* evaluation method works better on the 20-object problem, and the *partial scan* method works better on the 80-object problem. Performance results are shown in Table 5. The swap interval for island models is 5.

gen and *std* for 20 and 40 object zero-one knapsack problems.				
	K20		K80	
	gen	std	gen	std
SGA	88.7	302.1	32.5	7.9
ESGA	62.2	64.5	41.9	11.8
pCHC	31.3	12.6	32.7	8.2
Genitor	24.7	9.9	17.8	4.8
I-SGA	43.0	31.2	41.0	10.6
I-ESGA	100.3	253.8	33.0	7.7
I-pCHC	34.7	12.9	37.2	8.9
I-Genitor	25.8	17.6	22.0	5.8
Cellular	32.3	12.5	34.7	7.9

Table 5: Performance of GAs on knapsack problems

4.5 SUMMARY OF RESULTS

Table 4 shows the relative ranking of the genetic algorithms on each of the functions. Table 8 shows the relative performance of the algorithms on each problem, normalized on a scale of 0 to 1 (where 1 is the worst). This is done by dividing each performance value by the worst performance score on that problem. Since some of the algorithms perform similarly on some problems, this gives a better picture of how the algorithms compare with each other. Aggregate values for Tables 4 and 8 are given in the rightmost columns. Table 6 gives the same aggregate information for only the hardest problems (F2, F4, Rastrigin, Schwefel, Griewangk, the deceptive functions, and the 20-object knapsack). Table 7 gives the same aggregate information for only the problems with long bitstrings (F4, Rastrigin, Schwefel, Griewangk, and the 80-object knapsack).

Algorithm	Avg	rnk	Nrm	rnk
SGA	7.6	9	.84	9
ESGA	6.3	7	.55	7
pCHC	4.4	4-5	.29	3
Genitor	4.4	4-5	.42	6
I-SGA	6.9	8	.63	8
I-ESGA	4.0	3	.34	5
I-pCHC	3.5	2	.28	2
I-Genitor	3.3	1	.21	1
Cellular	4.6	6	.32	4

Table 6: Performance of nine GAs on hard problems

Algorithm	Avg	rnk	Nrm	rnk
SGA	6.4	7-8	.79	9
ESGA	6.6	9	.63	8
pCHC	4.6	3-4	.41	2
Genitor	4.4	2	.56	6
I-SGA	6.4	7-8	.59	7
I-ESGA	4.8	5-6	.44	3
I-pCHC	4.8	5-6	.46	4
I-Genitor	2.4	1	.26	1
Cellular	4.6	3-4	.47	5

Table 7: Performance on long bitstring problems

The non-elitist algorithms (SGA and IslandSGA) clearly perform the worst overall, ranking 8th and 9th in every category. The parallel algorithms seem to perform better on harder functions. Island-Genitor gets the best marks, and the pCHC and cellular genetic algorithms also are consistently good. Interestingly, the cellular genetic algorithm and pCHC perform fairly well in spite of simplistic implementations.

Algorithm	F1	F2	F3	F4	F5	F6	F7	F8	D3	D4	K20	K80	Avg	rnk
SGA	.74	.68	.76	.33	.72	.86	1.00	1.00	.99	.98	.88	.78	.81	9
ESGA	.69	.19	.69	.31	.70	.19	.99	.66	1.00	.42	.62	1.00	.62	7
pCHC	.68	.37	.77	.45	.79	.04	.34	.45	.29	.05	.31	.78	.44	3
Genitor	.41	.46	.37	.27	.39	1.00	.76	.33	.26	.06	.25	.42	.42	2
I-SGA	1.00	1.00	1.00	.81	1.00	.48	.37	.31	.60	1.00	.43	.98	.75	8
I-ESGA	.78	.19	.83	.76	.68	.08	.15	.41	.11	.003	1.00	.79	.48	6
I-pCHC	.80	.18	.85	1.00	.80	.11	.01	.29	.22	.07	.35	.89	.46	4
I-Genitor	.56	.27	.56	.49	.55	.03	.05	.22	.28	.10	.26	.53	.33	1
Cellular	.79	.25	.81	.80	.75	.03	.04	.65	.37	.10	.32	.83	.48	5

Table 8: Normalized performance of nine GAs on test suite

5 CONCLUSIONS

In this paper we have compared several parallel genetic algorithms across a wide range of optimization functions. We note that, overall, elitist strategies perform better than non-elitist ones. More significantly, alternative population structures such as cellular, steady-state (i.e., Genitor), and CHC approaches are at least as effective as elitist versions of the standard "Holland-style" genetic algorithm, even without taking actual parallel execution into account. Since our versions of CHC and cellular genetic algorithms are greatly simplified, it is reasonable to expect the full implementations of these algorithms to perhaps perform even better. Results for island models are good, especially for harder problems.

The performance of SGA is relatively poor compared to the other alternative algorithms examined in this study. However, this paper has largely been concerned with optimization as measured by the best solution found during a single run. Much of Holland's original work was concerned with the *online* performance metric which considers the average value of all points sampled by the genetic algorithm. When considering *online* performance, consistently sampling good points takes on more importance than simply finding the best point in the space which may be isolated in an otherwise "poor" region of the search space.

The findings reported in this study are encouraging for parallel genetic algorithm research because they indicate that *performance benefits due to parallelism are not offset by declines in problem-solving capabilities*. In fact, parallel genetic algorithm using some form of restricted selection and mating based on locality that are executed serially often yield better performance than single population implementations with global "panmictic" mating. We would point out, however, that to refer to these results as an example of "superlinear speedup" is technically incorrect: we are not achieving superlinear speedup due to the parallelization of a single algorithm, but rather have examined different parallel algorithms that display superior problem solving capabilities.

References

A. Böhm and G. Egan (1992). Five Ways to Fill Your Knapsack. *Colorado State Univ tech report CS-92-127.*

K. DeJong (1975). An Analysis of the Behavior of a Class of Genetic Adaptive Systems. PhD thesis, U. of Michigan.

L. Eshelman (1991). The CHC Adaptive Search Algorithm. *Foundations of Genetic Algorithms and Classifier Systems*, Morgan-Kaufmann.

D. Goldberg (1989). *Genetic Algorithms in Search, Optimization and Machine Learning* Addison-Wesley.

D. Goldberg, B. Korb, and K. Deb (1989). Messy Genetic Algorithms: Motivation, Analysis, and First Results. *Complex Systems 3*.

D. Goldberg and K. Deb (1991). A Comparative Analysis of Selection Schemes used in Genetic Algorithms. *Foundations of Genetic Algorithms*, ed. G. Rawlins, Morgan Kaufmann.

V. Gordon, D. Whitley, and A. Böhm (1992). Dataflow Parallelism in Genetic Algorithms. *Parallel Problem Solving from Nature 2*, North Holland.

J. Holland (1975). *Adaption in Natural and Artificial Systems.* Univ of Michigan Press.

H. Mühlenbein, M. Schomisch, and J. Born (1991). The Parallel Genetic Algorithm as Function Optimizer. *Parallel Computing 7.* North-Holland.

H. Mühlenbein (1992). How Genetic Algorithms Really Work I: Mutation and Hillclimbing. *Parallel Problem Solving from Nature 2*, North Holland.

D. Whitley and T. Starkweather (1990). GENITOR II: a Distributed Genetic Algorithm. *J. Expt. Theor. Artif. Intell 2* pp 189-214.

D. Whitley (1991). Fundamental Principles of Deception in Genetic Search. *Foundations of Genetic Algorithms*, ed. G. Rawlins, Morgan Kaufmann.

D. Whitley, R. Das, and C. Crabb (1992). Tracking Primary Hyperplane Competitors During Genetic Search. to appear in *Annals of Mathematics and Artificial Intell.*.

D. Whitley (1993). Cellular Genetic Algorithms. *Genetic Algorithms: Proceedings of the Fifth International Conference (GA93)*, Morgan Kaufmann.

Appendix A – Convergence Graphs