Evan Gerritz and Simon Van Der Weide

CPSC 424

Dr. Sherman

21 December 2022

Parallelizing Genetic Algorithms

Genetic algorithms are a class of artificial intelligence algorithms that attempt to replicate the success of nature's process of optimization: evolution. Many aspects of evolution are indeed already thought of in terms of function optimization, for example, convergence and survival of the fittest (which implies a measure of fitness). To find the maximum of a function using a genetic algorithm, we create a random population of potential solutions to the function (the encoding of each individual's solution is analogous to the DNA of an organism) and use the ideas of survival of the fittest (only the fittest portion of each generation reproduces), sexual reproduction (members of the next generation are some random combination of their parents), and mutation (local maxima can be avoided by causing individuals to be intentionally different from the current maxima with some probability). We wrote a genetic algorithm in C and then used two techniques of parallelization for genetic algorithms: a global single-population manager-worker model and a distributed genetic algorithm.

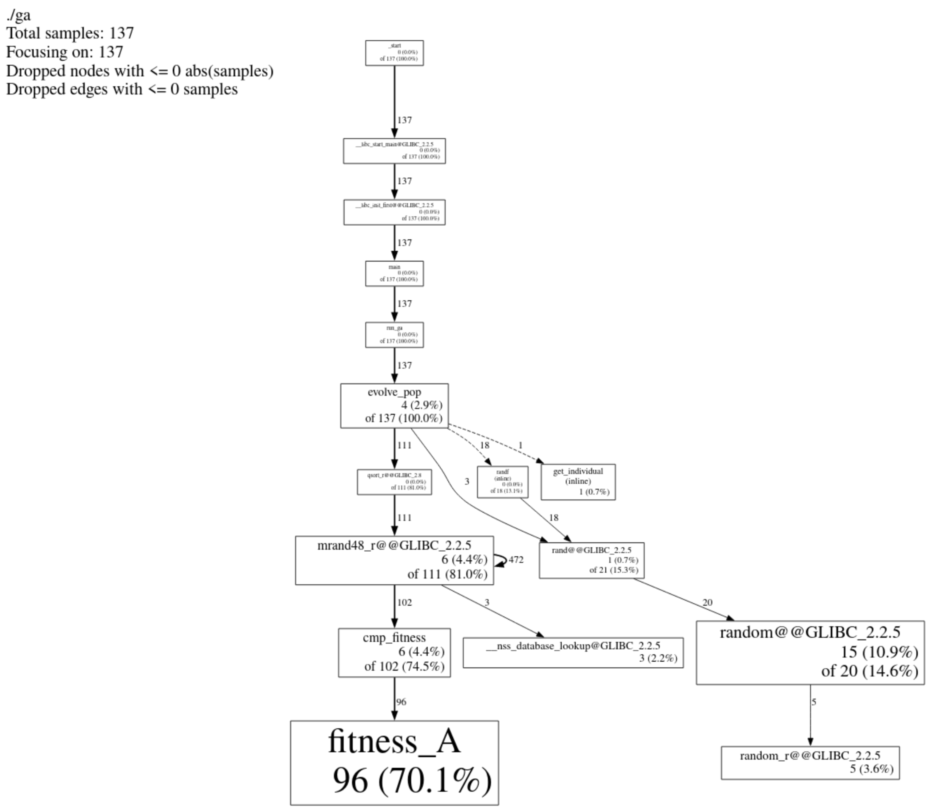
Our environment for this project consisted of the following modules: StdEnv, iccifort/2020.4.304, iimpi/2020b, GCCcore/10.2.0, numactl/2.0.13-GCCcore-10.2.0, imkl/2020.4.304-iimpi-2020b, zlib/1.2.11-GCCcore-10.2.0, UCX/1.9.0-GCCcore-10.2.0, intel/2020b, binutils/2.35-GCCcore-10.2.0, impi/2019.9.304-iccifort-2020.4.304. This report consists of 3 parts, one discussing our serial implementation (found in serial/), the next discusses our parallelization attempts using OpenMP (found in omp/), and the third discusses our implementation of a distributed population parallel genetic algorithm (found in mpi/). Each of these folders contains the relevant source code, a Makefile for building the program, the outputs used in the report, and a slurm batch script for running the program on a cluster node.

For our serial implementation, we referenced an implementation in Python that only optimized functions of two variables and generalized it to optimize a function of any number of variables. We tested our implementation's correctness and tuned parameters by using the following two test functions (denoted here f_A and f_B):

$$f_A(x_1, x_2, \dots, x_n) = 1 - \sum_{k=1}^{n} \left(\sum_{i=1}^{n} (i^k + \beta)\left(\left(\frac{x_i}{i} \right)^k - 1 \right) \right)^2$$

$$f_B(x_1, x_2, \dots, x_n) = 1 - \sum_{k=1}^{n} \left(\sum_{i=1}^{n} \left(x_i^k \right) - \sum_{i=1}^{n} \left(i^k \right) \right)^2 = 1 - \sum_{k=1}^{n} \left(\sum_{i=1}^{n} \left(x_i^k - i^k \right) \right)^2$$

In C, we used nested loops to calculate the inner and outer sums, and a third nested loop to compute the *k*-th powers needed in the inner sum. We chose to implement the exponentiation using a third nested loop instead of the pow() function in the math library to avoid possible latency from the C library calls and to allow better register usage for the operation. Notably, these fitness functions require $O(n^3)$ time to compute, and they must be computed for each individual in each generation. Once we verified our fitness function was correct, we collected profiling information using the gperftools profiler (see Figure 1) and found that the algorithm spends roughly 70.1% of its time calculating the fitness of various candidate chromosomes, making the fitness calculations a good candidate for parallelization.


./ga
Total samples: 137
Focusing on: 137
Dropped nodes with <= 0 abs(samples)
Dropped edges with <= 0 samples
_start 0 (0.0%) of 137 (100.0%)
137
__libc_start_main@GLIBC_2.2.5 0 (0.0%) of 137 (100.0%)
137
__libc_init_first@@GLIBC_2.2.5 0 (0.0%) of 137 (100.0%)
137
main 0 (0.0%) of 137 (100.0%)
137
run_ga 0 (0.0%) of 137 (100.0%)
137
evolve_pop 4 (2.9%) of 137 (100.0%)
111
18
1
3
qsort_r@@GLIBC_2.8 0 (0.0%) of 111 (81.0%)
randf (inline) 0 (0.0%) of 18 (13.1%)
get_individual (inline) 1 (0.7%)
111
18
mrand48_r@@GLIBC_2.2.5 6 (4.4%) of 111 (81.0%)
472
rand@@GLIBC_2.2.5 1 (0.7%) of 21 (15.3%)
102
3
20
cmp_fitness 6 (4.4%) of 102 (74.5%)
__nss_database_lookup@GLIBC_2.2.5 3 (2.2%)
random@@GLIBC_2.2.5 15 (10.9%) of 20 (14.6%)
96
5
fitness_A 96 (70.1%)
random_r@@GLIBC_2.2.5 5 (3.6%)

(Figure 1: output of running the gperftools profiler on the serial implementation.)

However, our initial implementation's performance was highly variable and usually, it failed to find the correct global maximum of each function. Good values for certain parameters, such as the probability of mutation, are dependent on population size ($P(mutation) = 0.01$ might be enough with a population size of 1000, but it will lead to essentially no mutation in a population of size 10). The program's main parameters were population size, n, and . We also had several auxiliary parameters that determined the evolution of the genetic algorithm: the probability of mutation, the proportion of elite individuals, the proportion of parents for the next generation, the probability of crossover, and the maximum number of iterations allowed.

We tested our implementation on a wide variety of parameters, and there was no common trend on which settings would work for which population size and which problem, except for finding that small rates of mutation tended to be better in larger populations and larger rates of mutation were better in small populations. In the end, we could not find a set of auxiliary parameters that would work for all values of the main parameters. Looking into the literature, it does seem that this is a common problem for genetic algorithms: they require fine-tuning to generate the correct results and to avoid being stuck on local maxima. They cannot easily adapt the amount they change the current generation based on how good it is. A high mutation probability might be great for initially approaching the global maximum, but it will make it hard for the network to converge. Conversely, a low mutation probability is great once a good solution has been found but makes the algorithm prone to converging at a local maximum.

Since we were tasked with parallelizing the genetic algorithm, we decided that we would focus on the parallelization rather than 100% correctness across all cases. In this case, nature might have the solution: encode these parameters in each individual's DNA and allow the parameters themselves to be optimized in addition to the function the parameters are optimizing, though we did not explore this.

We were, however, able to find different parameters for various population sizes that consistently led to convergence. We also tweaked the algorithm from using single-point crossover (parents create two children by combining data with each other) to multiple-point crossover and implementing elitism (the fittest individuals of each generation are preserved in the next generation). We considered trying other well-studied and regarded forms of selection such as Roulette selection and tournament selection hoping that this would improve the network's consistency, but ultimately settled on using different values for different size problems.

Once our C implementation converged to the correct solution for fitness function A on $n = 4$ and in {0.5, 50} and fitness function B on $n = 5$ and in {0.5, 50}, we decided to modify the algorithm's termination conditions. Although in a less contrived situation, we may not know the maximum value the function attains, we do know the global maxima of the two fitness functions. The two functions subtract a nonnegative value (sum of squares) from 1, so we know that 1 must be the global maximum for both functions. Thus, rather than (or in addition to) checking whether the population has improved its fitness in the last *g* generations, we can check whether the average fitness of the population has reached 1, the global maximum.

Here are our results for the cases of $n = 4$ and in {0.5, 50} for fitness function A and B:

Fitness Function	Population Size	*n*	*Notes*	Avg Generations	Avg Time (ms)	Avg Time per Generation
A	10	4	=50	280.33	1.43	0.005
A	50	4	=50	182.00	7.22	0.040
A	100	4	=50	6.67	0.81	0.121
A	150	4	=50	347.33	50.58	0.146
A	1000	4	=50	3.67	6.89	1.877
A	10	4	=0.5	504.00	2.58	0.005
A	50	4	=0.5	965.00	38.01	0.039
A	100	4	=0.5	965.00	93.35	0.097

Fitness Function	Population Size	*n*	*Notes*	Avg Generations	Avg Time (ms)	Avg Time per Generation
A	150	4	=0.5	6.67	1.18	0.177
A	1000	4	=0.5	481.33	662.34	1.376
B	10	5	Random init.	24.00	0.18	0.008
B	50	5	Random init.	12.33	0.77	0.062
B	100	5	Random init.	6.33	1.04	0.164
B	150	5	Random init.	2.67	0.84	0.315
B	1000	5	Random init.	1.67	5.84	3.497
B	10	5	Semirandom	24.00	0.18	0.008
B	50	5	Semirandom	5.67	0.39	0.069
B	100	5	Semirandom	2.67	0.53	0.199
B	150	5	Semirandom	5.00	1.36	0.272
B	1000	5	Semirandom	1.67	5.84	3.497

(Figure 2: Serial times for our genetic algorithm to find the true maximum of the fitness function.)

Problem B was, in general, much easier to solve, and using the suggested initialization close to the true solution for 20% of the starting population (semirandom in the table above) tended to result in fewer generations being required to find the global maximum. The program struggled the most on problem A with =0.5, though it sometimes got lucky and found a solution very quickly (as with *n*=150). The relationship between the time required and the population size was obscure, as a small population may require more generations but each generation takes much less time to evaluate. Additionally, it is hard to compare times as there is a high degree of randomness in the algorithm.

Our first attempt to parallelize the algorithm was to use OpenMP to parallelize the fitness function, implementing a global single-population manager-worker model. Since the fitness function is the biggest "hot-spot" in our code, it seemed that by simply using *n* processors, one for each iteration of the outer loop, the code would be substantially faster while producing the same result. To avoid race conditions, each thread calculated its own sum stored in a separate location in an array of sums, and then these sums were combined by one thread. This resulted in the times below:

Fitness Function	Population Size	*n* (search space dimension, # of processors)	*Notes*	Avg Generations	Avg Time (ms)
A	10	4	=50	280.33	33.49
A	50	4	=50	182.00	206.35
A	100	4	=50	6.67	22.04
A	150	4	=50	347.33	1455.22
A	1000	4	=50	3.67	200.10
A	10	4	=0.5	504.00	62.51
A	50	4	=0.5	965.00	1024.08
A	100	4	=0.5	965.00	2599.23
A	150	4	=0.5	6.67	32.40
A	1000	4	=0.5	481.33	20315.50

Fitness Function	Population Size	*n* (search space dimension, # of processors)	*Notes*	Avg Generations	Avg Time (ms)
B	10	5	Random init.	24.00	4.14
B	50	5	Random init.	12.33	20.68
B	100	5	Random init.	6.33	27.17
B	150	5	Random init.	2.67	23.39
B	1000	5	Random init.	1.67	173.87
B	10	5	Semirandom	24.00	4.14
B	50	5	Semirandom	5.67	10.53
B	100	5	Semirandom	2.67	13.70
B	150	5	Semirandom	5.00	38.12
B	1000	5	Semirandom	1.67	166.83

(Figure 3: OpenMP-parallelized fitness function times for our genetic algorithm to find the true maximum of the fitness function.)

These are all substantially slower than the serial times, suggesting that by splitting the sum across multiple processors, either some highly effective compiler optimization was lost, or (more likely) the amount of work being done by each parallel was much too small relative to the overhead of creating and synchronizing 4 threads. These times were from using the option “static,1” for the parallel for loop scheduler as this seemed to be the only sensible option for our case with a number of iterations equal to the number of processors. Other options tested included “dynamic” and “guided”; these options all produced similar results, as expected since “static,1” should already be exactly what we want. For brevity, these times are not included in this report, however, the raw output file is in omp/gaomp_fit.out.

While a majority of the time was being spent in the fitness function, we parallelized the algorithm at too fine a granularity, resulting in a drastic decrease in performance. In the second OpenMP implementation, we tried to increase the granularity of the parallelization by instead parallelizing the sorting of each generation by fitness at the beginning of each time step. This was done by writing a recursive parallel quicksort function that creates an OpenMP task for each half of the array it sorts. The timing results of this implementation follow:

Fitness Function	Population Size	*n* (search space dim, # of processors)	*Notes*	Avg Generations	Avg Time (ms)	Avg Time per Generation
A	10	4	=50	280.33	1.64	0.006
A	50	4	=50	182.00	16.00	0.088
A	100	4	=50	6.67	0.78	0.117
A	150	4	=50	341.00	274.02	0.804
A	1000	4	=50	3.67	6.54	1.782
A	10	4	=0.5	127.00	0.74	0.006
A	50	4	=0.5	15.33	0.81	0.053
A	100	4	=0.5	8.33	0.93	0.108
A	150	4	=0.5	8.00	2.23	0.279
A	1000	4	=0.5	4.67	8.83	1.891
B	10	5	Random	42.67	0.34	0.008

Fitness Function	Population Size	*n* (search space dim, # of processors)	*Notes*	Avg Generations	Avg Time (ms)	Avg Time per Generation
			init.			
B	50	5	Random init.	7.33	0.51	0.070
B	100	5	Random init.	6.00	0.95	0.158
B	150	5	Random init.	3.00	0.80	0.267
B	1000	5	Random init.	1.67	4.82	2.886
B	10	5	Semirandom	42.67	0.33	0.008
B	50	5	Semirandom	5.00	0.32	0.064
B	100	5	Semirandom	5.00	0.77	0.154
B	150	5	Semirandom	2.67	0.72	0.270
B	1000	5	Semirandom	1.67	5.02	3.006

(Figure 4: OpenMP-parallelized quicksort function times for our genetic algorithm to find the true maximum of the fitness function.)

This implementation is consistently faster than the previous two implementations. In particular, many of the $\beta = 0.5$ runs have drastically reduced in time. This seems to be mostly due to the average number of generations decreasing, which is somewhat anomalous as we have only switched from the standard library's quicksort to our own implementation, which theoretically should not have any impact on the progress of each generation. Comparing the average time per generation for this implementation against the serial, there is a slight increase for many of the problem A times, but a larger decrease in the times for problem B. We are still unsure as to the cause of these differences in the timing. It is at least clear that the changes in time per generation within a given problem correspond to the population size used, as a larger population requires more time to be sorted. The difference between the randomly initialized and semi-randomly initialized population is quite small in these trials. Overall, this parallelization improved the performance of the algorithm a fair amount.

Increasing the granularity of the parallelization even further, we experimented with a distributed population genetic algorithm using OpenMPI. We followed an approach outlined by (Cantu-Paz, 1998): "[Marin, Trelles-Salazar, and Sandoval] proposed a centralized scheme in which slave processors execute a GA on their population and periodically send their best partial results to a master process. Then, the master process chooses the fittest individuals found so far (by any processor) and broadcasts them to the slaves. Experimental results on a network of workstations showed that near-linear speed-ups can be obtained with a small number of nodes (around six), and the authors claim that this approach can scale up to larger networks because the communication is infrequent." For our purposes, we decided that we would collect the fittest individual from each process and broadcast it to all the MPI processes at every generation. We have to do this at every generation and, specifically, before the termination condition checks because (1) we are collecting and distributing the fittest individuals with MPI collective calls, and (2) every MPI collective call must be executed by all processes in the communicator for a valid MPI program.

Our MPI "island" approach takes in the initial population size and all the same parameters as the serial algorithm. However, instead of creating a single population with that size, it initializes the population and then distributes it over the number of MPI processes in the communicator. Initially, we distributed the population using *MPI_Scatter()* but then switched to *MPI_Scatterv()* to allow us to use population sizes that are not evenly divisible by the number of MPI processes. There is one case, however, that the “island” approach could not handle: a population size of 10 individuals. It does not work for this implementation because, with more than 2 MPI processes, there will be no children created in each generation, so the population will never change, and optimal fitness cannot be reached. At every generation, as noted above, there is a section that collects the fittest individual from every process with an *MPI_Gather()* call, sorts them using *qsort()*, and then broadcasts the fittest with an *MPI_Bcast()* call. We then overwrite the fittest individual in each generation with this individual, which cannot decrease the fitness because, for each generation, the fittest individual from all the islands is either its own fittest individual or fitter than its own fittest individual. After this, the termination condition is checked as in the serial code, and there is a final *MPI_Gatherv()* call to collect the distributed populations onto the root process for the final results.

Migrating from the *MPI_Scatter()* and *MPI_Gather()* calls to the *MPI_Scatterv()* and *MPI_Gatherv()* calls proved somewhat more troublesome than expected. Since the fitness algorithm requires the same amount of time for every possible individual with the same number of variables, load-balancing can be achieved by making the number of individuals in each island as close to equal as possible. We achieved rudimentary load balancing using an approach outlined in a StackOverflow post[1]. Once we figured out the number of individuals in each MPI process, we had to communicate that number to each process and make sure that they allocated enough space to hold that number’s worth of genes; the conversion between individuals and genes was rather difficult and caused several

[1] https://stackoverflow.com/questions/20348717/algo-for-dividing-a-number-into-almost-equal-whole-numbers

errors. Once it was working for the simple test case of $n = 4$ and $\beta = 50$ on all the population sizes in {50, 100, 150, 1000}, we figured it would work in general. This implementation cannot handle test cases where the island size (total population size / number of MPI processes) dips below the number of parents because the population never changes. However, although it cannot handle small population sizes, it can handle much larger values of n than the previous algorithms. With fitness function A, we had success up to the limit $n = 8$, mostly due to the limit on total number of generations / iterations. With fitness function B, we had success up to $n = 20$ with 5 MPI processes, receiving correct results within the maximum number of generations / iterations.

We report below some of the results we obtained from experimenting with this algorithm. The results do not cover the cases with a population of 10 or the semi-random seeding with fitness function B, but they do cover a few different cases that we thought were interesting. For fitness function A, we ran with 4 MPI processes. For fitness function B, we ran with 5 MPI processes with strictly random seeding of the population (i.e., no semi-random seeding).

Population Size	n	β	Generations	Time (ms)	Avg. Time (ms)
50	4	50	86	6.82	0.0793
50	4	0.5	12	1.35	0.1125
100	4	50	4	1.70	0.4250
100	4	0.5	4	1.70	0.4250
150	4	50	4	1.75	0.4375
150	4	0.5	3	1.74	0.4350
150	8	0.5	91	11.84	0.1301

Population Size	*n*	*β*	Generations	Time (ms)	Avg. Time (ms)
1000	4	50	3	2.58	0.8600
1000	4	0.5	5	2.64	0.5280
1000	<u>8</u>	0.5	37	45.36	1.2259

(Figure 4: MPI "island" implementation run on fitness function A with 4 distributed populations/processors)

Population Size	*n*	Generations	Time (ms)	Avg. Time
50	5	13	1.40	0.1077
50	10	243	10.93	0.0450
100	5	4	1.81	0.4525
100	10	35	4.62	0.1320
150	5	2	2.99	1.4950
150	10	31	5.94	0.1916
1000	5	1	5.82	5.8200
1000	10	23	37.67	1.6378
1000	<u>15</u>	101	330.51	3.2724
1000	<u>20</u>	4381	25736.39	48.8083
1000	<u>25</u>	71058	737835.42	10.3836

(Figure 5: MPI "island" implementation run on fitness function B with 5 distributed populations/processors.)

It is clear from these results that the distributed "island" version of the genetic algorithm, implemented with MPI, is significantly faster than the other two parallelization attempts. The MPI version is fast on each iteration of the evolution process because each MPI process works with a subset of the overall population. There should be an almost linear speedup in the evolution function because each process sorts, in parallel, a fraction of the total population proportional to the total number of processes. Additionally, there is the benefit that each "island" can explore a different portion of the problem space, and all the MPI processes share the fittest candidates of each generation, so no process lags behind the others too much. The exploration of different regions of the problem space, coupled with the sharing, allows the algorithm to reach the maximum fitness value in fewer generations, saving time on top of the individual iteration performance increase. However, the communication and synchronization costs between the processes are probably what make the algorithm run longer than the parallel OMP sorting algorithm on certain inputs.

The best indication that this MPI implementation works well is that it was able to compute a set of correct values for fitness function B with a population size of 1000 and $n = 25$, the largest value, by far, for which we tested the algorithms (previous attempts to solve it). Each of the implementations scales the allowed maximum number of iterations based on the value of n, the arity of the fitness function and number of data points in a single individual. While this value of n may have worked on the other algorithms if we had increased their maximum iteration counts beyond this normal scaling, we did not have to do so for the MPI implementation to achieve the results. In fact, it only took a fraction of the allowed 318750 generations. It did not achieve the same accuracy on fitness function A, but, in each case, it seemed to hit the maximum number of iterations, so it may have performed equally well if allowed more iterations and more time. The MPI version also scales with the addition of more MPI processes, subdividing the population into more islands and creating even more genetic diversity in the overall population.

After implementing the serial and several parallel versions of the standard genetic algorithm, we now understand the benefits and challenges of several methods of parallelizing genetic algorithms. Certain fitness functions and genetic algorithms would require prohibitively large computation times without parallelism. Even these simple fitness functions are quite computationally expensive, especially as the number of genes (i.e., function variables) increases. Indeed, genetic algorithms may need to be parallelized for them to have any practical use at all, since a simple algorithm like gradient ascent could solve our problem much faster. One of the most intensive parts of these genetic algorithms is the fine-tuning of the parameters for the evolution function, the fitness function, and several other pieces. If it takes hours to run the algorithm, then it takes hours to check for the correctness of the algorithm for each variation in the parameters, another problem that parallelism attempts to solve.

Of course, the two fitness functions we used for our project were already known quantities. We knew the maximum value they attained over their input domain and even the set of inputs for which the function attains that maximum. This allowed us to shortcut the general question of convergence for the genetic algorithms, an incredibly important question and topic of research. Whether or not the populations in our algorithm converged is not tested by the algorithm at all. In a more realistic situation, the convergence of the population is often an essential—if not *the* essential—test of whether a solution has been found. For our purposes, it was too difficult to parameterize the different components such that the population would converge at the global maximum for different population sizes. As is the case with many genetic algorithms, our algorithm tended to experience premature convergence at a local maximum or would not converge before reaching a generous upper bound on generations. For this reason, as mentioned above, we decided to exploit our knowledge of the functions to focus on the speedup from parallelism since we are, after all, studying parallelism.

If we were to continue researching genetic algorithms and methods to parallelize them, we would likely attempt a few different things. First, we would research the existing literature for fully specified and implemented genetic algorithms (the papers we read left the actual parameters and methods of population generation used unspecified) and attempt to parallelize them the same way we did for our naive serial implementation. Second, we would experiment with changing the termination conditions to check for non-improvement in the overall fitness over a set number of generations or some other appropriate condition. Third, we would look into other methods of parallelism (e.g., CUDA), and determine whether or not they apply to the problem. Fourth, we would try expanding the input space to all real numbers in a rectangular grid. This would be tricky as the floating point representations would need to be converted into some genetic representation such that similar genes encode close floating point numbers, but it would allow our approach to be effective on a much broader class of problems.

Bibliography

Alba, E., Dorronsoro, B. (2008). Introduction to Cellular Genetic Algorithms. In: Cellular Genetic Algorithms. Operations Research/Computer Science Interfaces Series, vol 42. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-77610-1_1.

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis, 10*(2), 141-171.