Asynchronous parallelism in steady-state genetic algorithms

V. Scott Gordon

Department of Computer Science
Colorado State University

ABSTRACT

Genetic algorithms (GA's) are becoming increasingly popular for signal detection, often in conjunction with neural networks. The time-intensive nature of these techniques has fostered an interest in parallel implementations. Genitor is a widely-used algorithm belonging to the class of *steady-state* GA's which are generally believed to contain little exploitable parallelism. Parallel versions have involved fundamental changes to the algorithm by introducing islands. This paper describes how Genitor can be parallelized *virtually as is*, with nearly linear speedup, by re-arranging the order of some of the genetic operations. An analytical method is derived which can be used for determining the amount of parallelism that can be achieved. An implementation for a shared-memory machine is described, and the resulting execution is shown to support the analysis.

Keywords: genetic algorithms, parallel algorithms, dataflow computing, Genitor.

1 INTRODUCTION

Genetic algorithms are search algorithms loosely based on principles of natural evolution, particularly genetic evolution. They are becoming popular for signal processing and detection, often in conjunction with neural networks. Genetic algorithms have been applied to finite impulse filters [10], infinite impulse filters [4], stack filters [3], binary phase-only filters [2], sonar data [9], and other domains [1, 8]. The time-intensive nature of these techniques has fostered an interest in parallel implementations.

Most implementations of genetic algorithms can be categorized as either *generational* or *steady-state*. Generational versions (such as the SGA [6]) produce an entirely new population at each generation. In a generational model, the use of tournaments to select strings ensures that no inter-dependencies exist, and thus each string in the new population can be created in parallel. By contrast, steady-state versions (such as Genitor [11]) produce offspring one at a time, inserting them back into the original population. Each string replacement in a steady-state model introduces a dependency, since there is some probability that the newly-inserted string will be selected for recombination at the very next trial. Therefore, parallel versions of steady-state genetic algorithms typically involve either isolating sub-regions (*islands*) to remove the dependency [12], or distributing strings across a network of processors and running in a generational manner (often called *massively parallel* or *cellular* [13]).

However, it is possible to execute steady-state models with a fairly high degree of parallelism (e.g., 50 or less processors), with nearly linear speedup on a shared-memory machine, *without changing the behavior of the algorithm*. Three similar methods for doing this are presented. We give an analytical method for determining the degree of parallelism which can be reasonably achieved. Finally, we describe an actual implementation on a shared memory Sequent Balance, and show how the resulting execution is consistent with the analysis.

2 PARALLELIZING GENITOR: PREGENITOR

Figure 1 shows pseudocode for a simple Genitor algorithm (with tournament selection), along with a parallelism profile produced by a simulator of the Manchester Dataflow Machine [7]. Critical path is shown on the x-axis, and parallelism is shown on the y-axis. Each repeating pattern is the creation and insertion of a new string into the population. The simulator recognizes the dependency and therefore executes each string replacement serially. This example is for 10 string replacements, with a population size of 10 and a string length of 10. A trivial fitness function, $f(x) = x$, is used so that the profile emphasizes the effects of genetic operations rather than effects due to the fitness function.

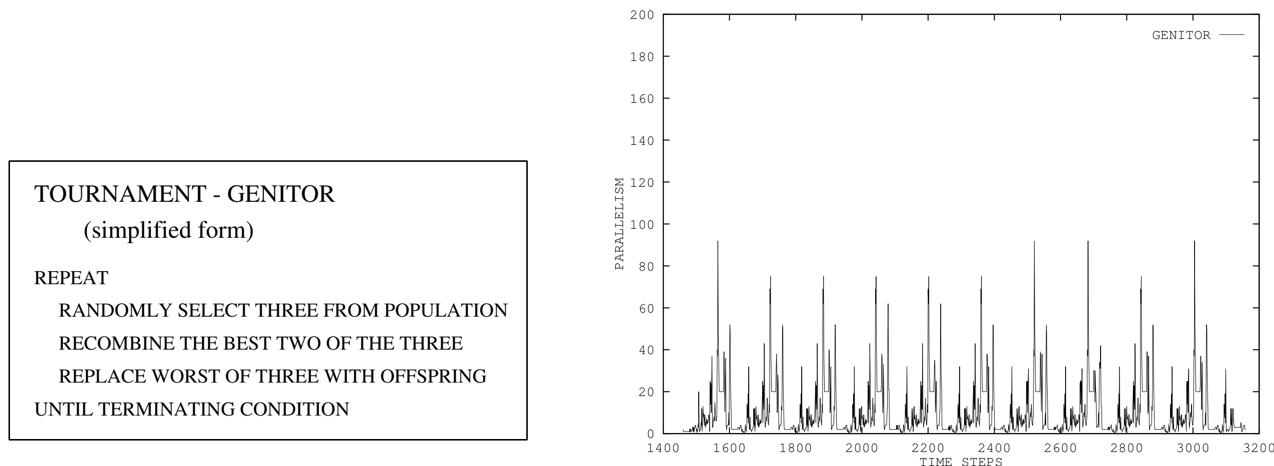

Figure 1: Genitor algorithm and parallel profile (10 string replacements). popsize = 10, string length = 10.

The probability that a newly generated string is selected next is $3/popsize$ (where *popsize* is the population size), since tournaments consist of three individuals (see Figure 1). If a population size of 100 were used, and two tournaments were performed in parallel, there would be only a 3% chance that such a *collision* would occur. It is therefore reasonable to assume that two tournaments could be performed in parallel most of the time. It follows that as the population size increases, more and more tournaments could be executed in parallel with a small likelihood of collision. Since testing for collision does not involve any genetic operations (it merely requires testing an integer for set membership), the actual recombinations could be deferred *until a collision occurs* and performed in parallel.

This gives rise to a re-ordering of the operations shown in Figure 1. Rather than generating the tournament and then immediately performing recombination, one could instead generate triples of integers (serially) in the range (1,*popsize*) until a collision occurs, placing them on a *ToDo* list, and then perform the recombinations afterwards during a separate parallel phase. This algorithm, which shall hereafter be referred to as *PreGenitor*, is *functionally identical to Genitor*, and the degree of parallelism is dependent on population size. The larger the population size, the less chance of collision, and the greater the resulting parallelism. Figure 2 shows the revised PreGenitor algorithm, along with a parallelism profile produced by the dataflow simulator, assuming that collisions result after exactly two recombinations.

Note that parallelism is roughly doubled (the "spikes" are twice as high) and that the critical path has been cut in half (the graph is half as wide). The only additional serially processed steps in PreGenitor are those involving the maintenance of the "ToDo" list. Speedup approaches linear as the complexity of the evaluation function increases, because lengthy evaluation functions will dominate the critical path for string generation.

```
PRE GENITOR

REPEAT
    REPEAT {serial phase}
        RANDOMLY GENERATE 3 STRING INDICES
        ADD TO ToDo-LIST
    UNTIL COLLISION
    REPEAT {parallel phase}
        SELECT ITEMS FROM ToDo-LIST
        RECOMBINE THE BEST TWO OF THE THREE
        REPLACE WORST OF THREE WITH OFFSPRING
    UNTIL ToDo-LIST EMPTY
UNTIL TERMINATING CONDITION
```

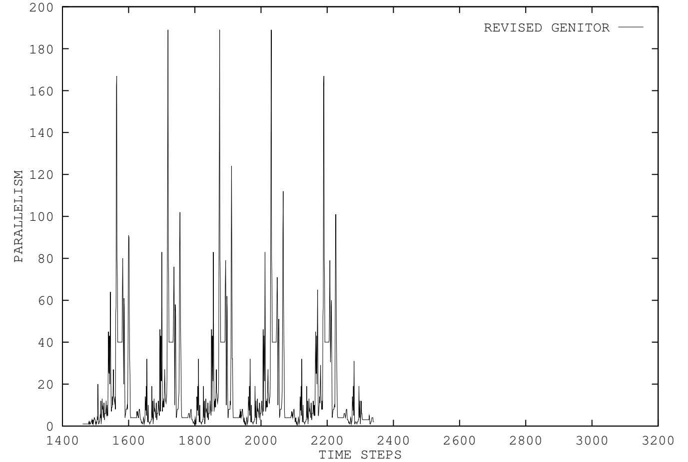

Figure 2: PreGenitor: algorithm and profile (parallelism=2).

3 ANALYSIS OF PREGENITOR

The effectiveness of PreGenitor as a parallel algorithm depends on the probability of collisions. Assume that there are k processors. We start by computing the probability of at least one collision in a list of k random integers each within the range (1,p), where p is the population size. The number of ways of selecting k random integers without replacement is p *choose* k. The number of ways of selecting k random integers with replacement is p^k. Therefore, the likelihood of randomly selecting an individual that has not yet been selected (i.e., no collision) is:

$$\frac{\binom{p}{k}}{p^k} = \frac{p!}{(p-k)! \cdot p^k} \tag{1}$$

It follows that the probability of a collision is:

$$C_k = 1 - \frac{p!}{(p-k)! \cdot p^k} \tag{2}$$

Table 1 shows C_k for various values of p and k. The meaning of the values in Table 1 depend on how PreGenitor is implemented. We will consider three versions of PreGenitor, denoted Type A, Type B, and Type C.

3.1 Type A PreGenitor

The pseudocode for Type A PreGenitor was shown previously in Figure 2. Parallelism varies because collisions do not necessarily occur at regular intervals (although in Figure 2 we assumed, for clarity, that they did). Tournament selection and recombination are both performed during the parallel phase, since any of the three strings in a given tournament might be selected for replacement. A variable number of parallel processes must

PreGenitor collision table			
k	C_k		
	p=50	p=100	p=500
2	.02	.01	.002
3	.06	.03	.005
4	.12	.06	.01
5	.19	.10	.02
6	.27	.14	.03
7	.36	.19	.04
8	.45	.25	.05
9	.53	.31	.07
10	.62	.37	.09
11	.69	.43	.10
12	.76	.50	.12
13			.15
14			.17
15			.19
16			.21
17			.24
18			.27
19			.29
20			.32
21			.35
22			.37
23			.40
24			.43
25			.46
26			.48
27			.51

Table 1: Probability of collision in a stream of integers in (1,p).

be allocated depending on how large the *ToDo* list grows before a collision occurs. One measure of the degree of parallelism is the expected time E for the first collision. Determining E requires calculating the probability of a collision occurring precisely at each value of k, and then using these values as a probability distribution. The probability of a collision occurring precisely at the kth iteration is equal to $C_k - C_{k-1}$, thus the expected value of the time of collision is:

$$E(p) = \sum_{k=2}^{p-1} \left[\left(1 - \frac{p!}{(p-k)! \cdot p^k} \right) - \left(1 - \frac{p!}{(p-(k-1))! \cdot p^{(k-1)}} \right) \right]$$

$$= \sum_{k=2}^{p-1} \frac{p!(k-1)}{p^k (p-k+1)!} \qquad (3)$$

We have computed $E(p)$ for small values of p (i.e, less than 50), and found that in these cases, E is very close to values of k (Table 1) where $C_k = .50$. Thus it appears that E can be approximated by using Table 1 in this manner. The number of parallel processes is then computed by dividing k by the tournament size (i.e., by 3).

Thus one would expect to use on average $(k|C_k = .50)/3 = 3$ processors when $p = 50$, 4 processors when $p = 100$, and 9 processors when $p = 500$.

3.2 Type B PreGenitor

Type B PreGenitor extracts additional parallelism by performing tournament selection during the serial phase. The advantage of this approach is in the observation that collisions are only bad *when they involve the losers of the tournaments*. Thus if it were known which individuals would lose, it would be possible to concentrate only on the collisions that mattered, and not worry about the others. Type B PreGenitor is also functionally equivalent to Genitor.

The degree of parallelism for Type B PreGenitor can be calculated similarly as for Type A. Estimating the expected value for the first collision can be done using Table 1. Here it is not necessary to divide k by three, since only one of the strings in a prior tournament can cause a collision. However, any of the three strings in the most recent tournament must be considered (selecting a string that was previously replaced is a collision). Therefore we can use a "sliding window" (of size three) on Table one to count the number of tournaments that can occur until the window contains the point at which $C_k = .50$. A simple way to calculate this is $(k|C_k = .50) - 3$, which equals 6 when $p = 50$, 9 when $p = 100$, and 24 when $p = 500$.

There are difficulties in using Type A and Type B PreGenitor. It is not always efficient to allocate processors dynamically for each set of tournaments. Allocating a fixed number of processors for an entire run may be preferable. This is the motivation for Type C PreGenitor.

3.3 Type C PreGenitor

Type C PreGenitor uses a fixed number of parallel processes. The number of active tournament selection processes is preset and remains constant. Unlike Type A and Type B PreGenitor, Type C PreGenitor is not functionally identical to Genitor, because collisions are possible. However, by using Table 1, it is possible to select an appropriate number of processes so that the expected number of collisions is very low, and the behavior closely approximates that of Genitor.

Type C PreGenitor does not require a ToDo list. Each process generates tournaments independently, and collisions are ignored. The effect is as if a ToDo list were built by generating a series of n random numbers in the range (1,*popsize*), where n is the desired number of parallel processes times 3 (since we are assuming a tournament size of 3). Since any of the three strings might be selected for replacement, a conservative approach would be to choose n from Table 1 where $C_k = .50$, and thus we could use $(k|C_k = .50)/3 = 3$ processors when $p = 50$. Similarly, we could use 4 processors when $p = 100$, and 9 processors when $p = 500$. Since some of the collisions do not matter (such as those where the same string is selected for recombination twice), these may be overly conservative values.

The algorithm can be implemented either synchronously or asynchronously. That is, each process could access a shared population simultaneously, either in lockstep or independently. The previous discussion assumed lockstep (i.e., where each process performs one recombination). Asynchronous processing can be described as several "Genitors" each selecting and inserting from the same pool. Collisions result in wasted effort. For example, when two "Genitors" try to replace into the same string location, the first replacement is wasted effort because it is immediately overwritten by the second.

4 IMPLEMENTING PREGENITOR

We implemented an asynchronous version of Type C PreGenitor on a shared memory Sequent Balance, using up to 12 processors and a single shared population. Several key points from Section 3 can be illustrated.

4.1 Effect of population size on collisions

Figure 3 shows the execution of Type C PreGenitor on an 30-bit deceptive function. In this function, ten 3-bit deceptive problems are concatenated together, so that in order to solve the function the genetic algorithm has to solve all ten subfunctions. Deceptive functions were introduced by Goldberg [5]; the subproblems here are not interleaved as they are in the "ugly" deceptive functions often described in GA literature.

The graphs on the left and right are for population sizes of 100 and 500, respectively. The plots are for various numbers of processors. The x-axis indicates the number of recombinations done by each processor. Mutation rate is .04, tournament size is 3. The plots show the best string found so far, averaged over 30 runs.

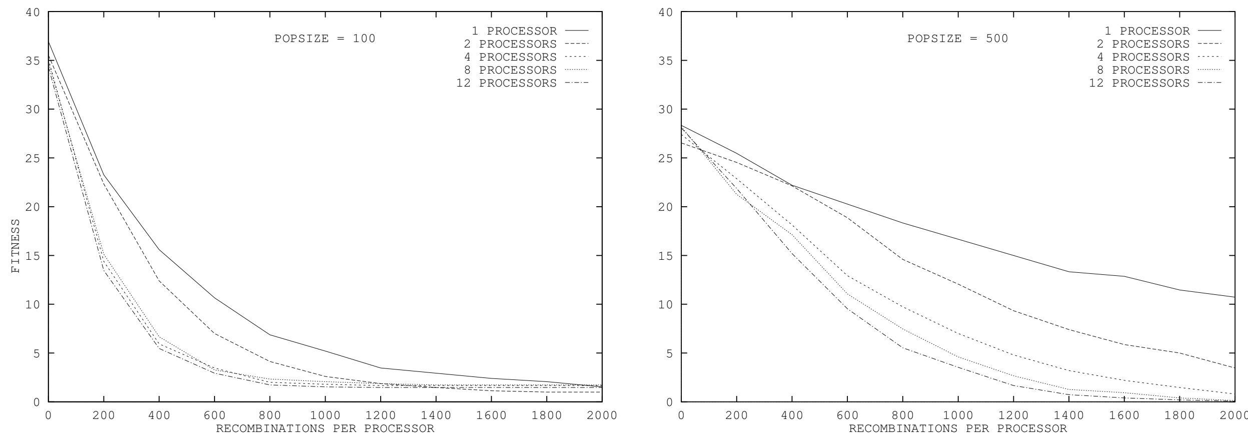

Figure 3: PreGenitor Type C, applied to a 30-bit "easy" deceptive function, for various numbers of processors. For $p = 100$ (left), only 4 processors can be effectively exploited – adding more is futile because of collisions. In the chart on the right, all parameters are identical except for population size ($p = 500$), and the additional processors are not wasted.

When there are no collisions, adding processors should result in more recombinations, and correspondingly fewer generations to find the optimum. This is noted in the graph on the right (Figure 3), but not in the graph on the left, because of the difference in population sizes. As indicated in Section 3, at $p = 100$ collisions become probable ($C_k \geq .50$) at only 4 processors, whereas at $p = 500$ this does not occur until 9 processors. These values are strikingly illustrated in Figure 3; the graphs corroborate the model very closely. It is possible that the *even slightly poorer* performance for $k > 4$, $p = 100$ (Figure 3, at the left) represents some form of degraded behavior due to frequent collisions, but this is not clear.

4.2 Actual speedup

Although an algorithm may in theory exhibit a certain speedup, sometimes overhead associated with processor allocation and communication can reduce the speedup. Although these effects vary from machine to machine, it is fairly easy to measure the time speedup for the Sequent implementation of Type C PreGenitor.

Speedup is measured by running PreGenitor for 120000 *total* recombinations using various numbers of processors. For instance, each processor in an 8-processor configuration is run for half as many recombinations as for a 4-processor configuration. The measurements are given in Table 2 below, and plotted in Figure 3.

#proc	recomb/proc	run time	speedup
1	120000	836.9 sec	—
2	60000	420.6 sec	1.99
4	30000	215.5 sec	3.88
8	15000	112.0 sec	7.47
12	10000	77.7 sec	10.77

Table 2: Measured time speedups for PreGenitor.

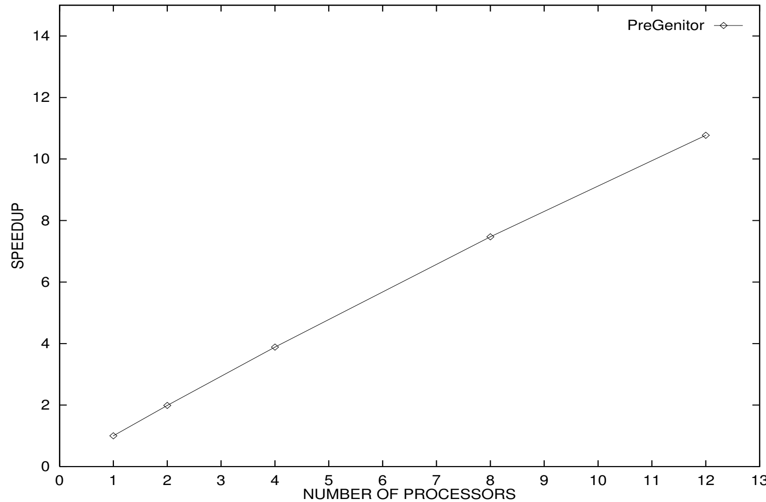

Figure 4: Observed speedup as processors are added to PreGenitor

5 CONCLUSIONS

We have described methods of exploiting parallelism in steady-state genetic algorithms, with minimal effect on their original functionality. The resulting algorithm, PreGenitor, has been introduced and an actual implementation on parallel hardware has been described. Speedup is obtained by re-ordering the Genitor operations, and the effect on critical path is directly observable on a parallelism profile produced by a dataflow simulator. A useful feature of PreGenitor is that there exists an accompanying analytical method of pre-determining the degree of parallelism which can be achieved. Preliminary verification of the analytical method has been presented, and the results are in close agreement.

PreGenitor's utility is its ability to run asynchronously, with little inherent overhead. Our shared-memory implementation displays nearly linear speedup for a simple evaluation function. For more complex functions, speedup should be even more linear, because lengthy function evaluations would dominate the critical path.

A disadvantage of using parallel steady-state algorithms such as PreGenitor is that they can only utilize a relatively small number of processors. However, the number of processors which can be utilized increases as the population size increases. Since large population sizes are believed to be beneficial in many cases (especially for

Genitor), PreGenitor may prove to be a powerful way of exploiting MIMD shared memory machines with less than (on the order of) 50 processors.

The ultimate usefulness of PreGenitor depends on whether or not steady-state algorithms such as Genitor are better function optimizers than generational, island, cellular, or other genetic algorithm methods. However, even though the degree of exploitable parallelism in PreGenitor is fairly limited, the claim that steady-state models are inferior because they are not parallelizable is invalid, at least for a significant class of parallel machines.

6 ACKNOWLEDGMENTS

The author wishes to thank Raja Das for his help and insight on this problem.

References

[1] S. Barnard and A. Bergman. Adaptation in signal spaces. *Parallel Problem Solving from Nature*, Springer Verlag, 1991.

[2] D. Calloway. Using a Genetic algorithm to design binary phase-only filters for pattern recognition. *Proceedings of the 4th International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1991.

[3] C. Chu. A genetic algorithm approach to the configuration of stack files. *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1989.

[4] S. Flockton and M. White. Pole-zero system identification using genetic algorithms. *Proceedings of the 5th International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1993.

[5] D. Goldberg. Simple genetic algorithms and the minimal, deceptive problem. in *Genetic Algorithms and Simulated Annealing*, L. Davis, ed. Morgan Kaufmann, 1987.

[6] D. Goldberg. *Genetic algorithms in search, optimization and machine learning* Addison-Wesley, ©1989

[7] J. Gurd, C. Kirkham, and W. Böhm. The Manchester dataflow computing system. in J. Dongarra, *Experimental Parallel Computing Architectures*, Special Topics in Supercomputing 1, North Holland, 1987.

[8] K. Mathias and D. Whitley. Noisy function evaluation and the delta-coding algorithm. *Neural and Stochastic Methods in Image and Signal Processing III (SPIE)*, 1994.

[9] D. Montana. Automated parameter tuning for interpretation of synthetic images, in *Handbook of Genetic Algorithms*, ed. L. Davis, 1991.

[10] J. Schaeffer, R. Caruana, L. Eshelman, and R. Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1989.

[11] D. Whitley. The GENITOR algorithm and selective pressure: why rank-based allocation of reproductive trials is best. *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1989.

[12] D. Whitley and T. Starkweather. Genitor II: a distributed genetic algorithm. *Journal of Experimental and Theoretical Artificial Intelligence*, 1990.

[13] D. Whitley. Cellular genetic algorithms. *Proceedings of the 5th International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1993.