Generating Functional Programs with Parallel Genetic Programming

Graeme McHale and Greg Michaelson

Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS
Email: `ceegsm@cee.hw.ac.uk` `greg@cee.hw.ac.uk`

Abstract

Genetic Programming is a programming paradigm where program development models biological evolutionary processes. Here we present a Genetic Programming system that solves simple tasks by generating solutions in a functional programming language. Parallel programming techniques are employed.

1 INTRODUCTION

1.1 Motivation

In our project to develop a parallelising compiler for SML [MSBK01], we exploit parallelism in higher order function (HOF) use through skeletons. Where a program lacks HOFs, we employ the λ-CLAM proof planner [CIM01] to try and synthesise HOFs using higher order unification and middle out reasoning. In effect, we attempt to generate appropriate HOF arguments from parts of function and then prove that the instantiated HOF is equivalent to the original function. However, the time taken to find HOFs grows very rapidly with HOF arity and with the complexity of the source program. For example, to find the single *scan* in a 2 line function to produce successive integer sums of list segments takes 199 seconds. In contrast, finding the 14 combinations of map and $foldr$ in a 7 line matrix multiplication program takes 10899 seconds.

Experience with genetic programming, to generate polynomial equations to fit tables of economic data[McH00], suggested that this might be an alternative approach to synthesising functions. However, generating functions is considerably more complex than generating equations. The use of a parallel island model both for table fitting and for finding coefficients for profiling equations in our compiler [MS00], suggested that this could offer useful time savings in search space traversal for function synthesis.

This paper presents preliminary findings from an investigation of parallel island model genetic programming for the synthesis of simple recursive functions in the strict, ad-hoc plyomorphic, pure functional language Navel[Mic93]. Subsequent sections discuss genetic algorithms and programming, Navel, the parallel architecture, results and conclusions.

1.2 Genetic Algorithms and Genetic Programming

Genetic Algorithms [Gol98] are a technique for searching large solutions spaces, taking inspiration from the genetic processes seen in the natural world. The first step in a GA is to randomly generate a population of *chromosomes*, each of which encodes a possible solution to the problem at hand. The mapping of the chromosome structure to the problem solution is highly problem specific. Each chromosome has a *fitness value*, a a measure of the distance between the required solution and the possible solution represented by the chromosome. This fitness value is evaluated by some *fitness function*.

A GA iterates through successive generations of the population. Each time the population is evaluated by the fitness function. Should any solutions prove sufficient then the algorithm will terminate. Otherwise the candidate set evolves into its next generation via a set of pseudo *genetic operations*. A fundamental principle here is that a specific chromosome's chance of forming part of this next generation is dependent on its fitness value. Thus GA mimic the "survival of the fittest" principles observed in nature.

The most basic genetic operation used is *reproduction*. Here a fit chromosome is copied directly from the current generation to the next. However if this were the only operation available, although the average fitness of the population would increase, there would be no progress towards the required solution. The first and most dominant operation which ensures this progress is *crossover*, which takes two different chromosomes at random from the population and splits and recombines them to produce two new chromosome. If two chromosomes give separate partially acceptable solutions to a problem, then the combination of these two chromosome may produce solutions which combine the best features of both parents.

With both reproduction and crossover, populations can converge over several generations to a single group of particularly fit candidates. Good as these candidate solutions may be, their over representation will prevent the determination of even better candidates. The genetic operation of *mutation* changes the value of chromosomes selected to make up part of the next generation at random in the hope of finding fitter solutions.

In a simple GA, a single population is evolved. Another approach is to populate independent *islands* that evolve independently against the same fitness function. Every so often, islands may exchange chromosomes. The intention here is that each island follows a distinct evolutionary path but may develop a uniform population which cannot evolve further. The exchange of extraneous chromosomes may provide new genetic material in the local population which enables further evolution take take place.

The island model is well suited to parallelisation as each island may be placed on a separate processor for autonomous evolution. The processing/communication balance for the whole model may be fine tuned by adjusting the timing and rate of inter-island communication.

Genetic Programming (GP) is essentially an extension of the field of (GA). In

GP, the chromosomes encode computer programs whose fitness is usually determined by executing them on known data and comparing the results with those expected. The principles behind GP are covered extensively by Koza's text [Koz92]. Grant's PhD thesis [Gra00] also gives an in depth survey of the other literature available.

1.3 Navel & Navel Light

Navel is a strict, ad-hoc polymorphic, run-time typed functional programming language which was originally developed as the meta-language for an interpreter-interpreter. Navel is based on Turner's original SASL[Tur73] augmented with first class parse rules for coercing strings to concrete syntax trees. Core Navel provides:

- integer, boolean, character and list values and operations;
- recursive and non-recursive functions;
- local definitions;
- curried and higher order functions.

To date, most GP has been aimed at synthesising LISP programs as LISP notation greatly eases syntax tree manipulation. Here, Navel seemed an appropriate choice for GP as it is very simple and is supported locally. The full Navel system is only 172032 KB in size, leaving most memory on a typical processing element for interprocessor communication and the local chromosome population. On a list processing benchmark, the Navel system runs at twice the speed of interpreted SASL and at comparable speed to interpreted Franz LISP.

For GP, a cut down version, termed *Navel-Light* was produced:

- All interactive features were removed.
- Grammar parsing features were removed.
- Protected division by zero was added, as in common in most GP applications.

As discussed in the next section one of the major issues in allowing the system to generate recursive functions is that of *gene termination*. Termination is undecidable and there is no way to ensure that terminating genes will be generated. It was decided to add a recursion limit within the interpreter itself. If a gene calls itself more than a set number of times it is no longer allowed to recurse, and the interpreter passes back an arbitrary but context specific value.

2 SYSTEM ARCHITECTURE

The system under consideration consists at the highest level of a master process and a number of worker processes. Each worker process is responsible for the

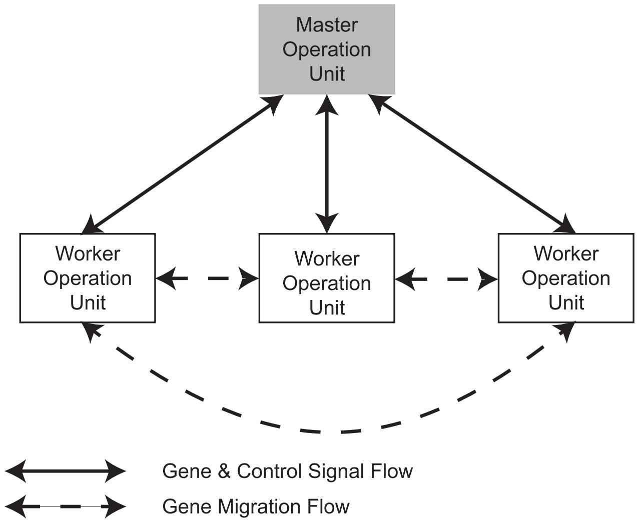

FIGURE 1. Genetic Processing World

evolution of a local set of candidate solutions or genes in order to try and globally solve the problem at hand. The master process is responsible for the distribution of genes in-between periods of local evolution.

The overall architecture of the system, or the *Genetic Processing World* is displayed in Figure 1.

Each operation unit executes on a separate physical processor concurrently thus achieving parallelism. Communication between the operation units is either in the form of genetic information (either distribution of candidates or the migration of candidates between islands) or control signals.

The two main functional units described in whole system architecture given in Figure 1 are the *Master Operation Unit* and the *Worker Operation Unit*. The internal structure of the Master Operation Unit (MOU) is shown in Figure 2.

The MOU has overall control of all islands, as well as acting as a worker island itself, essentially having it's own Worker Operation Unit (WOU).

Work on the islands is carried out by the WOU, illustrated in Figure 3. Each island (through the WOU) initially generates its own starting population. The first population is then locally evolved for a given number of generations and is then sent back to the MOU for globalisation. Upon a command from the MOU a new generation is sent and the WOU should evolve this as required. The WOU's are also responsible for managing the migration of the fittest genes between islands periodically.

The system was implemented in approximately 5,000 lines of C, not including the *Navel-Light* interpreter. Parallelisation was achieved using the message-passing paradigm, with the MPI libraries. Their use in relation to parallelising a standard GA is discussed in [McH00].

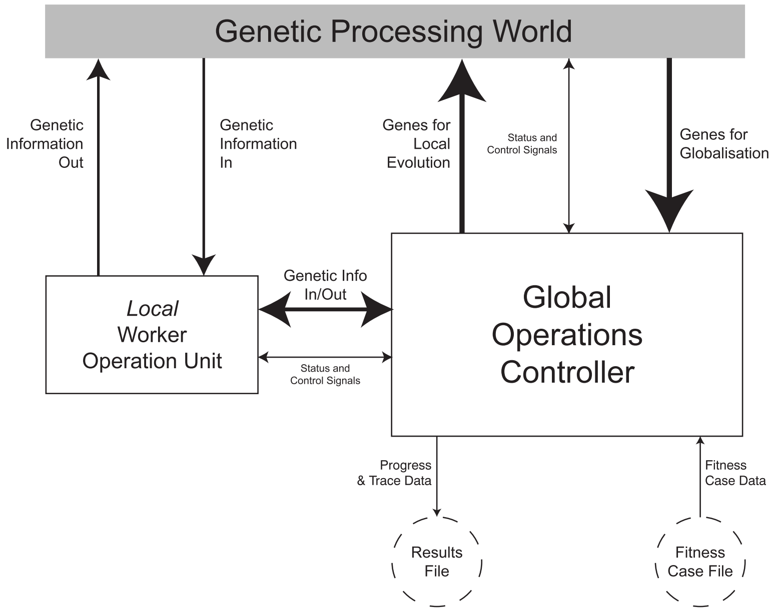

FIGURE 2. Master Operation Unit

3 RESULTS

Non-Recursive Function Generation

The initial operational requirement of the system was that it be able to find solutions to simple problems which involved the use of both arithmetic and conditional expressions, for example:

```
def func a b c = if (a>=b & a<=c)
                 then a*(a*a)
                 else 0;
```

As well as the expected function, the system also found the following functions:

```
def func a b c = (if (((a>c)|(b>a)))
                 then (10/0)
                 else ((a*a)*a));

def func a b c = (if (((a>=b)&(c>=a)))
                 then ((a*a)*a)
                 else (c*(c*0)));
```

Determination of these functions using 8 islands took an average of 4.0 development cycles, and and took an average of 102.93 seconds.

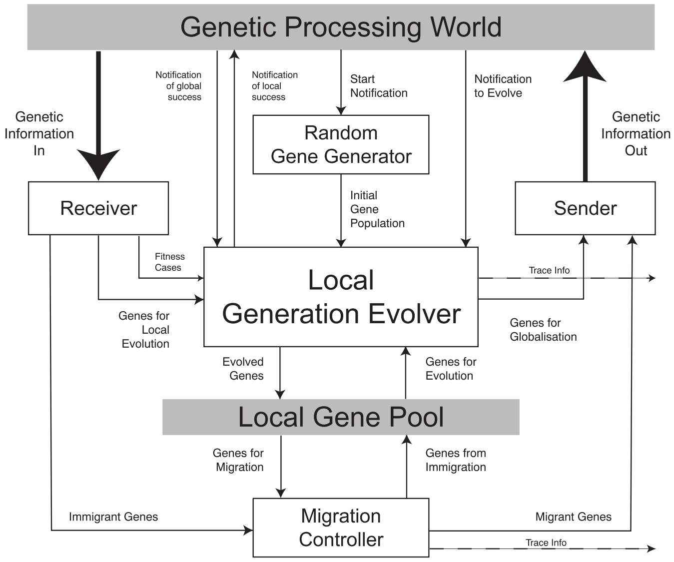

FIGURE 3. Worker Operation Unit

Recursive Function Generation

Trying to genetically develop functions to solve recursive problems is extremely difficult not least for the following reasons:

- Most functions generated will be non-terminating.
- A sequential approximation which gives the same results over a finite set of values is usually preferred by genetic development as it is easier to find.

Never-the-less it was still a major goal to be able to genetically develop solutions to functions such as that given below, which sums all integers between 0 and `a`.

```
def func a = if (a=1)
             then 1
             else a+(func (a-1));
```

The only way that it was thought that a solution to this function could be found was to limit the system to developing within a specific term structure:

```
def func a = if ({small conditional})
             then {variable or value}
             else {arithmetic expression} {op} (func(a-{value}));
```

Limiting the development structure in such a way did allowed the system to determine the above function. Amongst the expected results, the system also found the following functions:

```
def func a = (if ((a<>0))
              then (a+(func (a-1)))
              else a);

def func a = (if ((0<a))
              then ((func (a-1))-((0-a)*1))
              else 0);
```

Finding these solutions took 500 genes each on 4 islands, taking an average across 5 runs. Solutions were found in an average of 21.46 seconds and never took more than 1 development cycle.

Solutions to other simple linear recursive functions were found using the same method.

3.1 Local Evolution Analysis

One of the major advantages of using the PIM over more conventional methods is it allows the concept of local evolution. In traditional methods, worker processes are only delegated part of the development cycle, but with LE, each worker is responsible for the whole process. It was felt necessary to both verify and demonstrate that with local evolution; development happens both more quickly, and more efficiently.

Due to the inherent non-deterministic nature of a genetic programming system, all values presented are an average taken across 10 system runs. The general parameter values used both for these tests, and those in the following section are given in Table 1.

The fitness function that the system was required to determine in each of the experiments was:

```
def func a = if (a>10)
             then a-1
             else a;
```

Table 2 shows the average results of trying to find an optimal gene using numbers of local evolutions between 1 and 20. A local population of 2000 genes was used in all cases. It can be observed that the efficiency of the process, (shown by the number of cycles taken) does indeed improve as the number of local evolutions used is increased.

The time taken to achieve an optimal gene also reduces to a point as the number of LEs used increases. However, as the minimum number of cycles possible is 1, if the number of LEs is too large then there is a lot of redundant processing should an optimal gene be found early in this cycle.

System Efficiency

In assuming that there is some physical limit to the number of genes each island in the system can process, one can establish how much more efficient and effective the system becomes when more islands are added.

Parameter	Value
No. of Islands	4
Maximum Permitted Integer	10
Initial Maximum Tree Depth	2
Overall Maximum Tree Depth	4
Genes Reproduced	90%
Genes crossed-over	30%
Genes Mutated	5%
Frequency of Migration	Every 3rd Evolution
Number of Migrants	5
Optimal Fitness Value	0.0

TABLE 1. General Parameter Values

No. of Local Evolutions	1	2	5	10	20
Average Time Taken (secs)	13.03	8.88	13.95	10.98	30.91
Average No. of Cycles Taken	7.0	3.1	2.4	1.0	1.0

TABLE 2. Local Evolution Number Analysis

The efficiency of the system is believed to be demonstrated by how quickly (in terms of a reduction in fitness) genetic development occurs. The most reliable measure of this will be the number of cycles taken by the system to find an optimal gene. As genetic processing is non-deterministic, this was measured as an average across a number of system runs.

Analysis was carried out in three separate experiments, using a varying local population size. The results are shown in Tables 3, 4 and 5 respectively.

There are a number of interesting observation that can be made from these results, all of which could be predicted:

- Generally, as the local population size of an island increases:
  - The fewer the number of cycles that are required to find a solution.
  - The longer a cycle lasts.

No. of Islands	1	2	4	8
Global Population Size	250	500	1000	2000
Average Time Taken	7.34	4.49	3.49	5.47
Average No. of Cycles Required	5.84	3.36	2.40	1.90

TABLE 3. Efficiency Analysis (250 Genes Locally)

No. of Islands	1	2	4	8
Global Population Size	500	1000	2000	4000
Average Time Taken	5.80	4.08	4.09	4.80
Average No. of Cycles Required	3.20	2.20	2.00	1.20

TABLE 4. Efficiency Analysis (500 Genes Locally)

No. of Islands	1	2	4	8
Global Population Size	1000	2000	4000	8000
Average Time Taken	17.06	10.07	8.85	15.48
Average No. of Cycles Required	3.00	1.80	1.60	1.30

TABLE 5. Efficiency Analysis (1000 Genes Locally)

- An increase in the number of islands within the system means:
  - A decrease in the number of cycles required and therefore a increase in overall efficiency.
  - A decrease in the time taken to find a solution and therefore an increase in performance. It is noted that this performance increase becomes less pronounced as the number of genes on each island increases. This is due to the fact gene processing time increases non-linearly as population size increases.

System Performance

Speedup is a measure of how much faster a system using *n* processors (or islands in this case) is than a system using 1 processor. It is calculated according to the following formulae:

$$Speedup = \frac{Time\, with\, 1\, island}{Time\, with\, n\, islands}$$

To gauge the speedup obtainable using this system a number of experiments were run fixing the global population size across a varying island number. Figure 4 shows the time taken to find an optimal gene, varying the number of islands within the system. Given these times, Figure 5 shows that for 2 and 4 islands, the speedup given by the system is linear, as expected. For 8 islands, speedup approaches being super-linear: As the number of genes on an island decreases, the time taken to process each gene decreases super-linearly as each *Navel-Light* interpreter has to perform less *garbage collection*. Each generation was subjected to 10 local evolutions in a cycle.

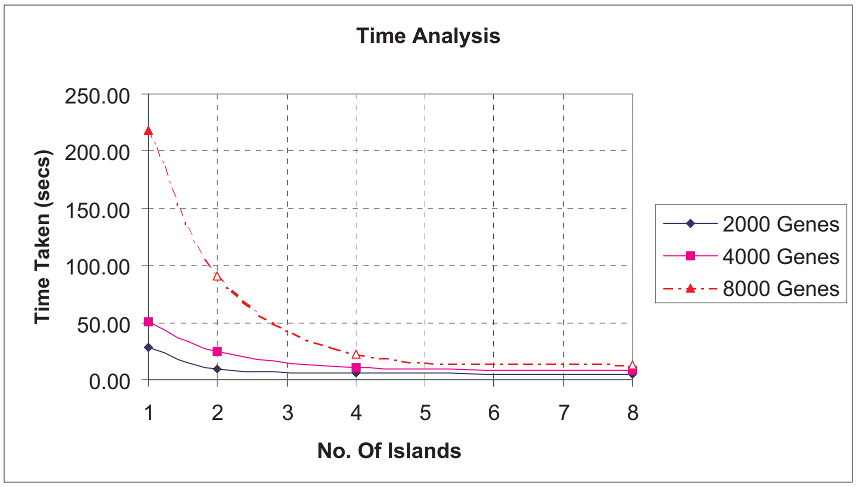

FIGURE 4. Island Number Analysis - Time Taken

Therefore this system shows that the performance of a genetic programming system can be at least linearly increased through parallelisation via the model presented.

All tests were run using a 32 machine Beowulf cluster. Each node had 533MHz Celeron processor, 128MB RAM and ran a Linux operating system.

3.2 Migration Frequency Analysis

A key feature of the PIM is *Migration*, or the swapping of genes between WOUs mid-cycle. It is thought that the best way to test migration has a positive effect on the development process was to measure the fitness of the fittest gene produced after each cycle over a fixed number of cycles, in this case eight. The frequency of the migration of a fixed number of genes was be varied and results observed. A function needed to be selected that would normally be expected to require 8 cycles or more to be found by system. The function chosen is given below. General system parameters were set according to the values in Table 6.

```
def func a b c = if (a>=b & a<=c)
                 then a*(a*a)
                 else 0;
```

The graph, Figure 6, shows that migration appears, to a point, to have the following effects:

- It increases the rate of fitness improvement within the system.
- It increases the amount of genetic diversity within the system, resulting in a reduced likelihood that the system will converge too early.

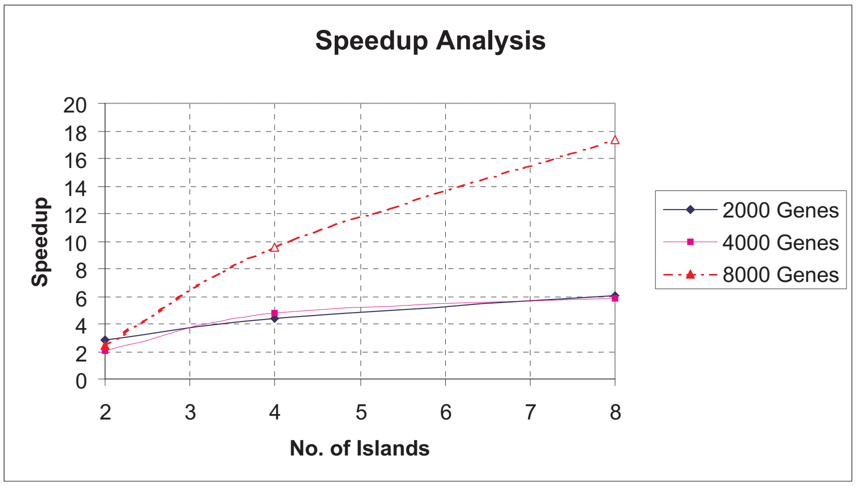

FIGURE 5. Island Number Analysis - Speedup

The limit of both of these observed effects, is thought to be due to the implications of the second: If there is more genetic diversity within the system, then there are more unfit genes.

4 CONCLUSIONS

The experiments described here provide initial assurance that GP is a plausible approach to generating recursive functions and that parallel island model GP offers useful speedup.

There is considerable scope for future work in synthesising functions in Navel. In particular the system should be extended to polymorphic functions and lists. Addressing either of these is implementationally complex because most *Navel-Light* base operators are monomorphic. An early and important design decision was that the interpreter should never be passed expressions which could cause an error. Providing support for anything other than integer types would involve the development of a type checker within the GP system.

As seen from examples above, a GP system tends to generate convoluted expressions which are unecessarily costly to evaluate. It would be useful to incorporate expression simplification before fitness evaluation.

It would be interesting to investigate the use of a theorem prover in measuring the fitness of synthesised functions. This would require some progressive measure of proof failure as most candidates would not meet problem criteria.

Parameter	Value
No. of Islands	4
Local Population Size	400
Maximum Permitted Integer	10
Initial Maximum Tree Depth	2
Overall Maximum Tree Depth	4
Genes Reproduced	90%
Genes crossed-over	30%
Genes Mutated	5%
Number of Migrants	5
Optimal Fitness Value	0.0

TABLE 6. Migration Analysis Parameter Values

REFERENCES

[CIM01] A. Cook, A. Ireland, and G. Michaelson. Higher Order Function Synthesis Through Proof Planning. *Formal Aspects of Computing*, submitted February 2001.

[Gol98] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley, 1998.

[Gra00] Michael Sean Grant. *An Investigation into the Suitability of Genetic Programming for Computing Visibility Areas for Sensor Planning*. PhD thesis, Heriot-Watt University, May 2000.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers of Means of Natural Selection*. MIT Press, 1992.

[McH00] Graeme S. McHale. Parallel Programming on Linux Networks Using MPI & PVM. Technical Report RM/00/10, Heriot-Watt University, Department of Comp. & Elect. Eng, 2000.

[Mic93] Greg Michaelson. *Interpreter Prototypes from Formal Language Definitions*. PhD thesis, Heriot-Watt University, April 1993.

[MS00] G. Michaelson and N. Scaife. Parallel Functional Island Model Genetic Algorithms through Nested Algorithmic Skeletons. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 307–313. Aachener Informatik-Berichte, September 2000.

[MSBK01] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications*, 16:181–206, August 2001.

[Tur73] D. A. Turner. SASL Language Manual. Technical report, Dept of Computational Science, University of St Andrews, 1973.

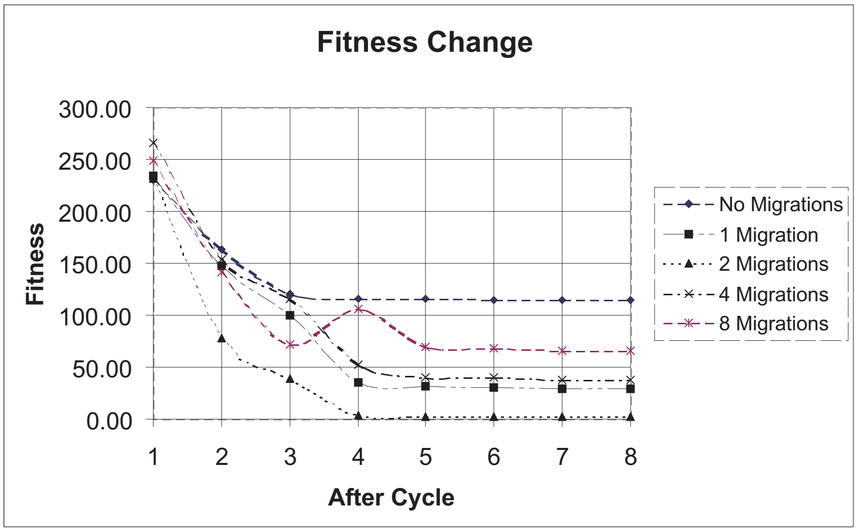

FIGURE 6. Effects of Migration on Fitness