Investigation of Genetic Algorithms and Cluster Computing

CAP 5937 – Cluster Computing
Summer Semester 2001

By

Hal Stringer

July 30, 1001

Executive Summary

This paper summarizes results from an investigation into the various methods for implementing parallel Genetic Algorithms (GAs). Experiments were conduct on a Beowulf Cluster using three different parallel methods and topologies. The results of these experiments show that parallelization can improve GA performance both in execution time (speedup) and quality of solutions found.

Table of Contents

List of Acronyms

GA	Genetic Algorithm

List of Figures

List of Tables

1 Introduction

The primary source for this section were my class notes.

Genetic Algorithms ("GAs") are used to search for solutions to a wide variety of problems. This search begins by encoding values for problem variables as genes represented by bit strings. Each chromosome contains a set of genes which together represent a possible solution (though not necessarily the best solution) to the problem being solved.

An individual's chromosome is used as input to the GA's fitness function which calculates a raw fitness for the solution represented by that chromosome. The fitness function may also compute a scaled or proportional fitness value which can be used later in the selection process to determine parents for mating (crossover).

A GA executes for a given number of generations or until the optimal solution is found. The following pseudo-code shows the steps and number of repetitions of each function that must be performed by a GA:

```
procedure GA
    initialize population;
    for (i=1 to number of generations){
        for (j=1 to population size)
            evaluate fitness of each individual;
        for (j=1 to population size / 2)
            select two parents for crossover;
            crossover to produce two children;
        for (j=1 to population size)
            perform mutation on children;
        for (j=1 to population size)
            insert children in next generation;
    }
end procedure GA
```

The pseudo-code clearly illustrates the repetitious nature of a genetic algorithm. In the majority of GAs, the fitness function is the most time consuming operation. Given that fact, a GA is of the order O(G*P) as it relates to fitness function evaluation where G is the number of generations and P is the population size. The speedup experiments described in this paper were performed for 500 generations with populations of 100 individuals. This required 50,000 fitness evaluations per GA run.

Genetic algorithms are pseudo-random in their processing. Functions such as selection, crossover and mutation use of a processor's random number generator to make probabilistic decisions. Since no random number generator is truly random, the same GA can be run with the same seeded generator and always produce the same result. Therefore, it is often necessary to run a GA multiple times (e.g., 20-50) using a different random number seed with each run to compensate for any pseudo-random behavior. The best solution is the most optimal solution found during all runs.

The more times a GA is run, the better the chances are of finding a more optimal solution. However, this must be done at the cost of additional processing time. For the speedup experiments described herein, each GA was run 32 times. As a result, each GA required a grand total of 1,600,000 fitness evaluations to search for a solution.

Give the number of fitness evaluations, GAs would seem to benefit from some form of parallelization. The goal of this project is to investigate those methods using a Beowulf Cluster and to determine whether such methods will reduce the required processing time and/or improve the quality of the solutions found.

In the next section, we describe some common methods for parallelizing a GA. In Section 3 we provide details about the problem, GA parameters, and parallel methods used in our experiments. Sections 4 and 5 show the performance improvements obtained as a result of parallelization, both speedup and improved solution quality. Finally, we conclude with some general observations and suggestions for follow up research.

2 Parallel Methods and Background Research

The primary source for this section was (Cantu-Pas, 2000).

A number of different techniques have been published which parallelize a genetic algorithm. Four major classifications were described in (Cantu-Pas, 2000) and include 1) Single-Population Master-Slave GAs, 2) Multiple-Population GAs, 3) Fine Grained GAs, and 4) Hierarchical Hybrids. We want to also mention Multiple-Run GAs and Functionally Parallel GAs as two additional parallelization methods. Each of these six methods is described briefly below:

- **Multiple-Run GAs** – An *n*-population GA is run on *p* processors (*p* [*n*, *n* usually divisible by *p*). Basically, this is a stand-alone GA run on multiple processors simultaneously. Each population initialized with a different random number seed. Speedup is achieved by performing multiple runs in parallel rather than serially on a single processor. This is the simplest form of a parallel GA and lowers execution time but does not effect the quality of solutions found.
- **Single-Population Master-Slave GAs** – A single-population GA that offloads only fitness function evaluation to other processors. Effective only if fitness evaluation time significantly exceeds communication time. May require multiple runs to compensate for pseudo-random behaviors. More useful for GAs with extremely complex fitness functions.
- **Functionally Parallel GAs** –A single-population GA run on multiple processors where each processor performs a specific GA function (e.g., fitness evaluation, selection, crossover, mutation). An extension of the Single-Population Master-Slave GA. Speedup of execution time is limited since at most four processors (one per

function) can be assigned to execute the GA. May need to be run multiple times to compensate for pseudo-random behaviors.

- **Multiple-Population GAs** – Most common parallel method. Also known as the "island" or "multiple-deme" method. An *n*-population GA is run on *n* processors. Each processor performs fitness evaluation and reproduction on its own population with periodic exchange of chromosomal material between processors. Although similar to a Multiple-Run GA where $n = p$, the Multiple-Population GA includes exchange of chromosomes between processors and the potential for improved solution quality.
- **Fine-Grained GAs –** A single population GA where each processor supports a single individual (chromosome). All fitness evaluations occur simultaneously. Reproduction occurs among individuals within defined local neighborhoods. Implementation of this parallel method requires high numbers of processors (e.g., 100+) and is the most difficult to code. May need to be run multiple times to remove pseudo-random behaviors.
- **Hierarchical Hybrids** – This catch-all category includes parallel GAs which combine the properties of the other five categories listed. For example, a Hierarchical Hybrid might combine a Functionally Parallel GA with a second group of processors dedicated to fitness evaluation (Master-Slave).

3 Experiments Performed

No primary source for this entire section. All original material.

A number of experiments were performed using some of the parallel methods described in Section 2. The purpose of these experiments was to determine if improvements in execution time (speedup) and quality of solutions could be found using parallel GAs. Three different parallel GAs were implemented in this project. The first was a Multiple-Run GA. The second and third parallel GAs were both Multiple-Population but using two different topologies: 1) a ring topology and 2) a fully connected network topology.

GA Design, Parameters and Coding

To make comparisons between methods easier, the same problem, GA structure and GA parameters were used for most of the experiments. The GA was designed to solve a 6-order symbolic regression problem for the following equation:

$$y = ax^6 + bx^5 + cx^4 + dx^3 + ex^2 + fx + g + h\cos(x)$$

A file of twenty-five (X,Y) pairs was read in at the start of the program. These would be used by the fitness function to determine the correctness of the solution represented by each chromosome. Y values were generated for each X value between 1 and 25 with the following coefficients:

$a = 2$	$e = -12$
$b = -144$	$f = 211$
$c = 2$	$g = -49$
$d = 121$	$h = 200$

The X value from the file was plugged into the above equation. Then each coefficient (*a*, *b*, *c*, *d*, *e*, *f*, *g* and *h*) obtained from an individual's chromosome (8 genes per chromosome) were also plugged into the equation. The equation was then evaluated and the result subtracted from the correct Y value obtained from the input file. The sum of the twenty-five absolute differences between these two numbers became the chromosome' raw fitness value.

The lower the raw fitness, the closer a chromosome's genes were to representing the coefficients listed above.

- Our GA incorporated the following features for every experiment:
- Population Size = 100 Individuals
- Representation Method = Bit String
- Number of Bits per Gene = 9
- Number of Genes per Individual = 8
- Crossover Rate = 0.7
- Mutation Rate = 0.01

Three important parameters, 1) number of runs 2) number of processors, and 3) swap option were varied during each experiment.

3.1 Swapping Options

The GA was designed to permit experimentation with various swapping options. The first option (SWAPTYPE = 0) has the parallel GA perform without swapping. No exchange of chromosomes occurs between populations. Instead, the program acts as a Multiple-Run GA. Each GA executes as a stand alone program on a single processor. However, all GAs are run simultaneously.

The second option (SWAPTYPE = 1) implements a Multiple-Population GA with a ring topology. Each GA shares chromosomal material with its nearest neighbor to the right (next higher rank). The last GA/processor completes the ring by sending its data to the very first non-control processor (rank = 1). The amount of data (SWAPCOUNT) and frequency of the exchange (SWAPINTERVAL) are determined by parameters that must be set prior to compilation. For all experiments, these parameters were set to 5 and 10 respectively. Random selection was used to determine the chromosomes to be sent to the next processor in the ring.

The third option (SWAPTYPE = 2) also implements a Multiple-Population GA. However, this version uses a fully connected network topology. Every 10 generations, each GA sends chromosomes from randomly selected individuals to every other GA running on the cluster. The chromosomes to be transmitted are selected randomly from the GA's population.

In both Multiple-Population methods, chromosomal material received from other GAs was inserted into the receiving population by randomly selecting an existing individual for replacement.

3.2 Source Code Overview

The GA was coded as a single "C" program with an associated "make" file (see disk included with this report.) After making any necessary edits to set parameters, the program is compiled using the command `make –f pga.make.` To execute the program enter the following:

```
mpirun –np numprocessors pga numruns[1]
```

where *numprocessors* is the number of processors on which the program will run and *numruns* is the number of runs to be execued on each processor.

[1] Time did not permit the addition of SWAPTYPE as a command line argument. This parameter must be changed in the source code, then the source recompiled.

The program was designed to use a single processor (P0) as a control and data collection point. All statistical information from other processors was forwarded to P0 for handling. All other processors (P*x* where $x \geq 1$) ran the portion of code specific to the genetic algorithm. Exchange of chromosomes occurred only between P*x* processors in multiple-population GAs.

As a result of this design, it is necessary to set *numprocessors* equal to $x + 1$ when invoking the program. This ensures that there are *x* processors running the GA and one additional processor to collect statistics.

A listing of the complete source code with comments is provided in Appendix A as an attachment to this report. All MPI calls have been bolded to assist in review of the code.

4 Speedup Results

No primary source for this entire section. All original material.

GAs using the three different swapping options outline in Section 2 were run on the SEECS Beowulf Cluster. Statistics concerning execution time were gathered by P0 and output to a summary file.

For each swap option, multiple runs of the experiments were performed to test execution times for various combinations of GA-only processors and runs. In all cases, the number of GA-only processors multiplied by the number of runs was always equal to 32. The results of these experiments are described in the following sections.

4.1 Speedup for Multiple-Run GA

The Multiple-Run GA exhibited a linear speedup tied to the number of processors used. The speedup was equal to or greater than the number of processors. This is not surprising given that the same program is run in parallel on multiple machines rather than serially on a single machine. The results of these experiments are listed in Table 1 below.

Table 1: Speedup Results for GA Executed on From 1 to 32 Processors

# Proc GA-only	Number of Runs	Total GA Time	Average GA Time	GA Speedup	Total Time	Avg Tot Time
1	32	118.77	118.77	n/a	237.55	118.76
2	16	119.55	59.77	1.99	179.56	59.85
4	8	118.86	29.72	4.00	148.76	29.75
8	4	118.25	14.78	8.04	133.44	14.82
16	2	118.74	7.42	16.01	126.45	7.43
32	1	118.45	3.70	32.10	122.41	3.71

The first column of the chart shows the number of GA-only processors used in each experiment. The second column shows the number of GA runs per processor. For example, in the last experiment, 32 processors ran the GA only once. In second to the last row, two runs of the GA were performed on 16 processors. The combinations of processors and runs were chosen to ensure that each experiment resulted in the same number of total runs (32) and thereby the same number of fitness evaluations (1.6M) regardless of the number of processors.

Columns three and four show the total time for GA-only processors and average time per processor (Total GA Time / # Proc). The speedup in column 5 indicates how much faster the Multiple-Run GA performed on 2 or more processors compared to running the GA 32 times on a single processor (row 1).

The last two columns of the table are provided as information only. They represent the total processing time including the time required by P0 for gathering and analyzing statistics sent from the GA-only processors.

4.2 SpeedUp for Multiple-Population GAs with Ring Topology

The next set of experiments were performed to test speedup on a Multiple-Population GA. In this set of experiments, chromosomes were passed in a ring topology. Thus, every ten generations, each processor forwarded five chromosomes to its nearest neighbor to the right and received five chromosomes from its neighbor to the left. The results are shown in Table 2.

Table 2: Speedup Results for Ring Topology GA Executed on From 1 to 32 Processors

# Proc GA-only	Number of Runs	Total GA Time	Average GA Time	GA Speedup	Total Time	Avg Tot Time
1	32	118.77	118.77	n/a	237.55	118.76
2	16	119.24	59.62	1.99	178.86	59.62
4	8	121.54	30.38	3.91	151.93	30.39
8	4	121.49	15.19	7.82	136.72	15.19
16	2	124.81	7.80	15.23	132.65	7.80
32	1	122.93	3.84	30.93	126.84	3.84

The first row of this table is used as a baseline for speedup comparisons. Swapping is not possible with a single processor. The results from the single processor, 32 run Multiple-Run GA are used here instead.

The speedup gained using the Multiple-Population method with 2 or more processors is almost as good as that obtained in the Multiple-Run GAs. Speedup is slightly lower due to the addition of communications costs in sending and receiving chromosomes. Informal experiments show that speedup decreases when the number of chromosomes exchange or the frequency of the exchange increases.

4.3 Speedup for Multiple-Population GAs with Fully Connected Network Topology

The last set of experiments related to speedup using a Multiple-Population GA with a fully connected network. Every 10 generations, each processor sent/received five chromosomes to/from every other processor in the cluster. Results from these experiments are provided in Table 3.

Table 3: Speedup Results for Network Topology GA Executed on From 1 to 32 Processors

# Proc GA-only	Number of Runs	Total GA Time	Average GA Time	GA Speedup	Total Time	Avg Tot Time
1**	32	118.77	118.77	n/a	237.55	118.76
2	16	119.63	59.81	1.99	179.45	59.82
4	8	124.47	31.12	3.82	155.59	31.12
8	4	137.37	17.17	6.92	154.55	17.17
16	2	549.24	34.33	3.46	583.67	34.33
32	1	1873.90	58.56	2.03	1932.73	58.57

Speedup was comparable to previous methods for experiments performed with lower numbers of processors (2, 4 and 8). However, speedup dropped dramatically for the experiments performed on 16 and 32 processors.

Execution times shown in this and other tables includes communication time as well as processing time. This makes it reasonable to assume that the amount of communication time required for the last two experiments negatively affected the overall execution time and hence the reduction in speedup.

For example, the 16-processor experiment required 5 * 16 or 80 chromosomes to be sent and received by every processor. This number multiplied by 16 processors and 72 bits per chromosome yields a total of 92,160 bits transmitted every 10 generations. The amount transmitted grows to 368,640 bits every 10 generations for the 32-processor experiment. The high volume of data coupled with additional MPI calls results in a loss of speedup as more processors are added to the fully connected network.

5 Qualitative Results

No primary source for this entire section. All original material.

Some parallel methods can improve the quality of solutions found by a GA as well as improve overall speedup of GA execution. A second series of experiments was performed to determine if this was the case with either of the two Multiple-Population topologies described previously.

No direct improvement in solution quality can be obtained from a Multiple-Run GA since no data is shared between populations. Any improvement in solution quality can only be gained indirectly by the ability to process more GA runs in a parallel environment.

Two methods were developed to compare solution quality. The first was to test the range of values for Best_of_Run found by each parallel GA. The second requires that we count the distribution of all Best_of_Run solutions found. For these sets of experiments, we used the same GA developed for the speedup experiments. However, each GA was performed for 25 runs on 10 processors three different times using different starting random number seeds. This gave us a total of 750 individuals representing the best of each run.

5.1 Range of *Best_of_Run* Values

The range of solutions found by parallel GAs can provide some evidence to prove improved quality of solutions. The highest and lowest Best_of_Run fitness values were found for the three experiments described previously. Table 4 summarizes these results.

Table 4: Highest (best) and Lowest (worst) results of all experiment runs

	Multiple Run (no swap)	Multiple Pop. (Ring)	Multiple Pop. (Network)
Lowest	122,392	243	126
Highest	45,941,294	115,054	1,033

The Multiple-Run GA had the worst performance in terms of fitness solution range. Despite 750 runs, it could do no better than 122,392 (remember that this GA minimizes fitness – the lower the better). And at least one run ended after 500 generations with fitness value greater than 45M.

The Multiple-Population GAs performed much better with significantly lower fitness values. And for the GA with network topology, the upper bound on the fitness was only 1033. This last parallel method/topology combination was the best at finding the smallest optimal solution range.

As information, the following chart lists the actual coefficients found by the fully connected network Multiple-Population GA for the lowest and highest Best_of_Run values. The first row ("A") contains the correct solution – the answer. The second and third rows contain the genotype for the individuals with fitness' of 126 and 1033 respectively.

Table 5: Coefficients foudn for highest (best) and lowest (worst) best of runs

Fit.	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
A	2	-144	2	121	-12	211	-49	200
126	2	-144	2	121	-12	211	-51	200
1033	2	-144	2	121	-13	238	-199	226

Both of these solutions are extremely close to actual coefficient values sought by the GA.

5.2 Distribution of *Best_of_Run* Values

Using the same data from the experiments performed in Section 5.1, we can prepare a table (see Table 6) and graph (see Figure 1) showing distribution of the 750 Best_of_Run solutions found by the three parallel methods. To simplify grouping, each fitness value was converted to an integer which equaled the truncated base 10 log of the fitness. For example, solutions 122,292 and 957,251 would both fall into the "5" group.

Table 6: Distribution of the raw fitness values from 0 to 10

	Count of Best Of Run Raw Fitness		
LOG_{10}	Multiple Run (no swap)	Multiple Pop. (Ring)	Multiple Pop. (Network)
0	0	0	0
1	0	0	0
2	0	6	35
3	0	18	18
4	0	24	24
5	9	289	185
6	44	149	311
7	164	264	177
8	521	0	0
9	12	0	0
10	0	0	0
Total Runs	750	750	750

The table shows us that overall, the solutions get better for Multiple Population GAs. Further, the fully connected network seems to provide better solutions than the ring topology. A chart graphically illustrating the above solution distribution is provided below:

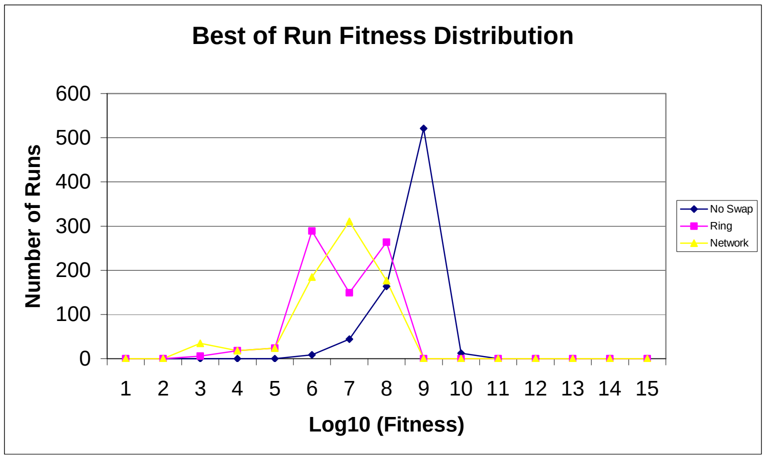

Figure 1: Graph of raw fitness distributions for values from 0 to 10

The distribution data described in this section corroborates the conclusions draw in Section 5.1 from the range information. We therefore can conclude that for some problems (i.e., symbolic regression) a parallel GA can improve the quality of the solution found.

6 Conclusions

No primary source for this section.

Genetic algorithms and cluster computing are a good match. The use of a cluster to parallelize a GA is simple, straightforward, and given the cost of clusters, relatively affordable. In its simplest form a cluster allows the user to cut processing time linearly by the number of processors using just a Multiple-Run GA. Dramatic speedup can be achieved by adding only a few lines of code to an existing GA.

The use of Multiple-Population GAs provides the added benefit of *potentially* better solutions. Experiments were limited to a specific problem (symbolic regression), therefore we cannot state conclusively that solution improvement will occur in all instances.

In the future, we hope to test for improvements in solution quality with other toy problem classes (e.g., number matching, traveling salesman problem) and with larger more complex GAs. The results should be interesting.

Our particular area of interest at the present time is the effect of gene-specific selection pressure on GA performance (Stringer, 2001). We hope to extend our investigation of parallel GAs to include gene-specific selection pressure, following up on neighborhood-wide research already conducted by other individuals (Sarma and De Jong, 1996).

List of References

Cantu-Paz, E. (2000). *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Press.

Sarma, J. and DeJong, K. (1996). *An Analysis of the Effects of Neighborhood Size and Shape on Local Selection Algorithms*. From Proceedings of the 4th Parallel Processing from Nature Conference.

Stringer, H. (2001). *Gene Values and Sub-Genotypes as an Indicator of Gene-Specific Selection Pressure within a Genetic Algorithm.* Class Project for CAP 5937, Evolutionary Computing, UCF, Spring 2001.

Appendix A

Source Code Listings