

# Characterizing the Genetic Programming Environment for FIFTH (GPE5) on a High Performance Computing Cluster

Kenneth Holladay

Southwest Research Institute

San Antonio, Texas

kholladay@swri.org

## ABSTRACT

Solving complex, real-world problems with genetic programming (GP) can require extensive computing resources. However, the highly parallel nature of GP facilitates using a large number of resources simultaneously, which can significantly reduce the elapsed wall clock time per GP run. This paper explores the performance characteristics of an MPI version of the Genetic Programming Environment for FIFTH (GPE5) on a high performance computing cluster. The implementation is based on the island model with each node running the GP algorithm asynchronously. In particular, we examine the effect of several configurable properties of the system including the ratio of migration to crossover, the migration cycle of programs between nodes, and the number of processors used. The problems employed in the study were selected from the fields of symbolic regression, finite algebra, and digital signal processing.

## Categories and Subject Descriptors

I.2.2 [Artificial Intelligence]: Automatic Programming – *Genetic Programming, High Performance Computing*

## General Terms

Performance, Experimentation, Languages

## Keywords

Genetic Programming, High Performance Computing, Digital Signal Processing

## 1. INTRODUCTION

Introduced in 2007 [7], the Genetic Programming Environment for FIFTH (GPE5) incorporates several important features that set it apart from other genetic programming (GP) systems. GPE5 programs are evolved using a stack-based language (FIFTH) whose syntax and interpreter is similar to FORTH. The single parameter stack for FIFTH holds containers that support multiple data types with intrinsic handling of vectors and matrices. Other

stack-based GP languages such as PUSH3 [14] use separate stacks for each data type with no provision for manipulating a vector as an object.

FIFTH is expressive and easily extensible. In addition to the typical math and logic functions, the operators include high level vector transforms (e.g., windowing and Fourier transform) as well as vector reductions (e.g. mean, standard deviation, and variance). GPE5 includes a FIFTH interpreter, allowing generated programs to be embedded directly into applications.

A second distinguishing feature is that GPE5 uses a linear program representation (instead of a tree) and performs all evolutionary manipulations using stack, branch, and flow control tracing to ensure both syntactic and operational correctness. While there have been other published efforts using stack tracing [15], GPE5 is the first publicly available code set to do so. These features enable GPE5 to solve classes of problems in areas such as digital signal processing that were previously intractable to GP techniques [8].

The initial design of GPE5 recognized the benefit of utilizing multiple processors to solve practical problems within reasonable time periods. The implementation was based on a distributed design with the Common Object Request Broker Architecture (CORBA) as the inter-process communication middleware. A single master node performed all of the genetic manipulation functions while the more time consuming fitness evaluations were distributed over available computing resources. The code base proved to be robust for heterogeneous groups of computers (mixtures of Windows, Linux and Sun operating systems) connected in a local area network, and it exhibited reasonable scaling efficiency for small numbers of computers. However, vector based signal processing problems often required days of wall clock time to complete a single run even when using up to 30 processors.

It was clear that achieving a meaningful reduction in the wall clock time consumed to solve complex problems would require a significant increase in computing resources, such as those available on a High Performance Computing (HPC) cluster. This paper describes the process of creating a Message Passing Interface (MPI) version of GPE5 to run on the Lonestar system at the Texas Advanced Computing Center (TACC). TACC is part of the National Science Foundation's TeraGrid project. Section 2 gives a summary of the technical approach used to port the software, as well as a description of the configuration parameters unique to the parallel implementation of GPE5. Section 3 introduces the experimental problems used to measure performance and the results for each problem. The final section discusses the practical implications of the work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO '09, July 8–12, 2009, Montréal Québec, Canada.

Copyright 2009 ACM 978-1-60558-325-9/09/07...\$5.00.

## 2. TECHNICAL APPROACH

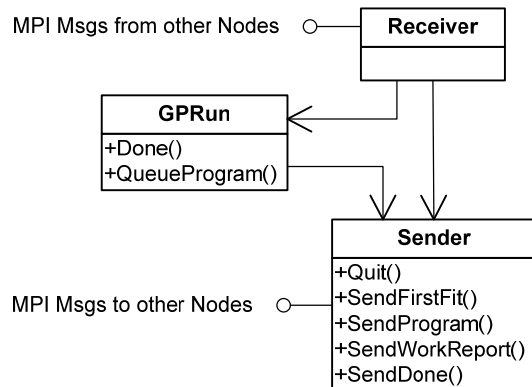
The distributed architecture of the CORBA version of GPE5 is not well suited to parallelization, as many of the processors are idle during the genetic manipulation process. When creating parallel versions of serial programs, using more resources is only effective if the computational work can be divided evenly among the available processors.

Fortunately, the GP literature describes multiple techniques for creating parallel versions of genetic programming algorithms, typically achieving linear or super-linear performance[1]. While recent publications have shown interesting results using the fine-grained model [6] and cooperative co-evolution model [11], we chose to implement a variation of the course-grained or island model [5] to minimize the necessary code changes. In the island model, every processing node runs the same GP algorithm performing both genetic manipulation and fitness evaluation on its own private population known as a deme.

Since CORBA is not typically supported for parallel applications on HPC platforms, the more common Message Passing Interface (MPI) [12] was selected for interprocess communication. An important goal of the software conversion work was to achieve a code base that could be compiled on multiple operating systems (Windows, Linux, Sun) and produce three GPE5 applications: a distributed version using CORBA, a stand-alone version for a single computer, and a parallel MPI version.

The first step in accomplishing these objectives was to restructure the code base to abstract the CORBA communication, thus decoupling the communication requirements from the genetic programming logic. This resulted in a FIFTH library containing all of the core GP components including the FIFTH interpreter, data set management, population management, fitness calculation, and genetic manipulation. The library consists of approximately 20,000 lines of code in 45 files. The reconstructed CORBA version required 11 additional files containing about 3,000 lines of code. After completing the code refactoring and abstraction, a stand-alone version of GPE5 was created by adding a single file with less than 800 lines of code.

As illustrated in Figure 1, the architecture for the MPI version (MGP5) centers around three new classes: GPRun (control logic for the GP algorithm), Receiver (accepts all incoming MPI messages), and Sender (handles all outgoing MPI messages).



**Figure 1 - Class diagram of the MPI logic and control portion of MGP5.**

Two key design choices resulted in a clean and easily extensible implementation comprised of only 1100 lines of code in 7 files. First, the multi-threaded version of the MPI library allows the Sender and Receiver to operate asynchronously while using the synchronous MPI\_Send() and MPI\_Recv() calls. Second, all messages between nodes are formatted in XML.

Our version of the island model required only one new configuration parameter. The serial version of GPE5 already had a parameter to specify how many of the top ranking programs were copied (migrated) into the next generation. For the parallel version, this becomes the number of programs to send to neighboring nodes. A new “Cycle” parameter controls how often programs are sent. A Cycle value of 0 means that no programs are exchanged, a value of 1 sends programs at the end of every fitness evaluation cycle, a value of 2 every other cycle, and so forth. The computing nodes are considered logically connected in a toroidal grid so that each node has four neighbors.

After genetically manipulating one generation to produce the next generation, the GPRun class checks the received program queue to determine whether any new programs have arrived from one or more of its neighbors. If so, all programs are removed from the queue and are added to the next generation before executing the fitness evaluation cycle. This method allows each node to progress through generations at an independent rate.

Node 0 has two unique responsibilities. First, it writes three summary reports at various stages in response to messages from the other nodes. These include a record of when each node found its first fit entry, an intermediate indication of the number of evaluations performed, and a brief summary generated during the shutdown process. Second, node 0 ensures a clean termination for all nodes. When any node meets the termination criteria (fitness, minimum generations, maximum generations, etc.) it sends a Done message to node 0. Node 0 then sends a Quit message to all other nodes, causing them to send their Done messages. Once node 0 receives a Done message from each of the other nodes, it sends out a Shutdown message causing all nodes to terminate.

All development and testing of MGP5 occurred on the Lonestar Dual-Core Linux Cluster which is configured with 5,200 compute-node processors, 10.4 TB of total memory and 95 TB of local disk space. The peak performance is rated at 55 TFLOPS. Nodes are interconnected with InfiniBand technology in a fat-tree topology with a 1GB/sec point-to-point bandwidth.

Jobs are submitted using the LSF batch system. Turn around time for a job ranges from a few minutes to a few days, depending on the resources requested (number of processors and estimated wall clock time) and system utilization.

The complete source code for all versions of GPE5 is available at [fifth.swri.org](http://fifth.swri.org).

## 3. EXPERIMENTS AND RESULTS

Characterizing the performance of a GP system is a difficult task. While there are many test problems that have been published by GP researchers, there is no well-recognized test suite. Many researchers choose one or more of the problems introduced by Koza [9, 10] such as Boolean multiplexers, Boolean parity, Lawnmower, Bumblebee, or others. A previous empirical study of the parallel performance of an island model, tree-based GP system [4], used even parity 4, artificial ant, and symbolic

regression. As noted by Daida [3], a general theme is to include one problem each from the categories of Boolean logic, symbolic regression, and finite state machine.

Rather than follow this trend for our initial characterization of MGP5, we decided instead to choose problems that differed significantly in three key areas that affect performance: data set size and type, operator selection and complexity, and expected problem difficulty. Problem difficulty is a comparison based on measurements from trial runs with the serial version of GPE5 and includes both an estimated probability of success (POS) and the average wall clock time consumed. The three selected problems are taken from symbolic regression, finite algebra, and digital signal processing.

### 3.1 Symbolic Regression

One of the most common real-world applications of GP is finding a mathematical model that fits a specific data set. In GP literature, this is often referred to as symbolic regression. We selected the binomial-3 problem for the first set of experiments. For a detailed analysis of the characteristics of this problem that make it popular for GP studies, see [2]. The model is expressed formally as:

$$f(x) = (1 + x)^3$$

#### 3.1.1 Data Set

The generated data set contains 500 points evenly spaced over the interval [-6, 6], which includes the inflection point (Figure 2). This is a moderate number of points that are easily kept entirely in memory. For each generation evaluation, the data set manager used 50 points, seven of which were common for all generations and 43 of which were randomly selected.

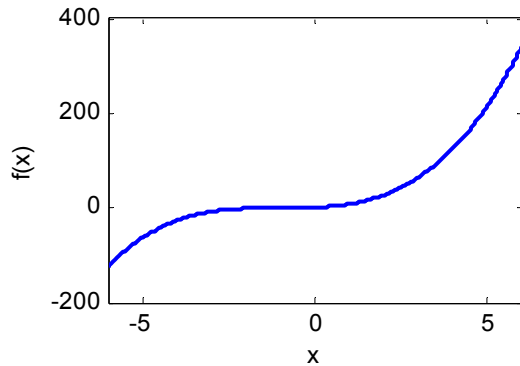


Figure 2 – Graph of the function  $f(x) = (1 + x)^3$ .

#### 3.1.2 Fitness

Program fitness ( $f_p$ ) was calculated as the mean of the difference between the program output ( $p_i$ ) and the known desired output ( $o_i$ ) over the number of data points used ( $n$ ). This provides a high resolution, continuous function with a value of zero being a perfect fit. The fitness target was set to 0.0001.

$$f_p = \frac{1}{n} \sum_{i=1}^n |p_i - o_i|$$

#### 3.1.3 Operators and Terminals

The set of operators was limited to primitive math ( $*/+-$ ). The only input terminal was the abscissa value ( $x$ ). GPE5 has a different approach to ephemeral random constants (ERCs). The configuration file specifies a set of values that may be randomly selected by the program builder and genetic operators. We have found that using a small set of prime numbers often works well as the programs evolve their own combinations to produce values in  $\mathbb{R}$ . For this problem, we included 1, 2, 3, 5, 7, 11, -1.

#### 3.1.4 Problem Difficulty

Test runs with this problem rank it as relatively easy. Using a population size of 1000 programs per generation, POS for a serial run is about 80% within 20 generations with an average time expenditure of about 7 seconds per generation.

#### 3.1.5 Experiments and Results

The first set of experiments used a migration cycle of zero and 128 processing nodes. The purpose was to characterize a baseline performance and determine the effect on POS when varying the probability of mutation (M) and crossover (X) during genetic manipulation. Figure 3 shows the results.

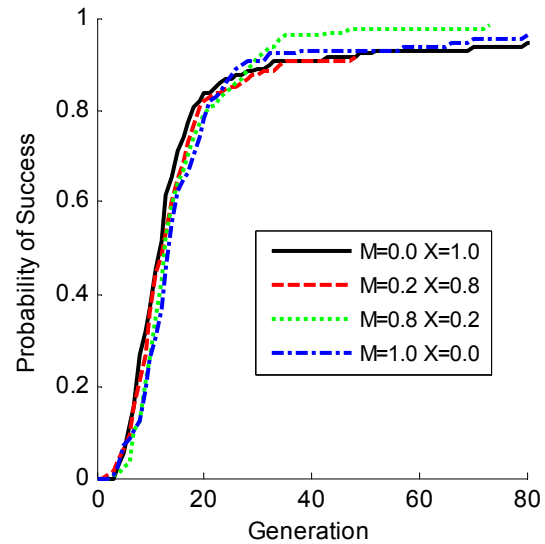


Figure 3 - Effect of mutation (M) and crossover (X) on the probability of success for the polynomial regression problem.

The highest POS is achieved using a mutation probability of 0.8 and a crossover probability of 0.2. Therefore, these probabilities were used in the second set of experiments to characterize the effect of the migration cycle. Still using 128 processing nodes per run, Figure 4 shows the improvement in the POS as the migration cycle is decreased from 4 to 1, effectively achieving a 100% probability of solution in less than 10 generations. We selected a migration cycle of 2 for the final experiment.

The final experiment was designed to indicate the effect of the number of processors on the wall clock time required to achieve a solution while performing approximately the same amount of work during each run. Using the total number of programs evaluated in a generation across all nodes as a rough estimate of

the work expended, we selected a baseline of 128,000 (product of the number of nodes and the population size per node). This approach has an obvious scalability problem since at some point the number of programs per node will drop to a critical level below which genetic manipulation will not be effective.

Table 1 shows the timing results for four representative runs. The Resource Seconds column is the time charged by the batch system for the run (the elapsed wall clock time multiplied by the number of processing nodes used) and is thus a real measure of the resources consumed. Note that increasing the number of processors from 16 to 128 results in a 32% reduction in resource consumption and an 84% reduction in wall clock time. Moving from 16 to 256 processors actually increases the resource consumption by 8%, but results in a 92% reduction in wall clock time. While 84% and 92% sound impressive, they equate to only 2.5 and 2.8 minutes respectively.

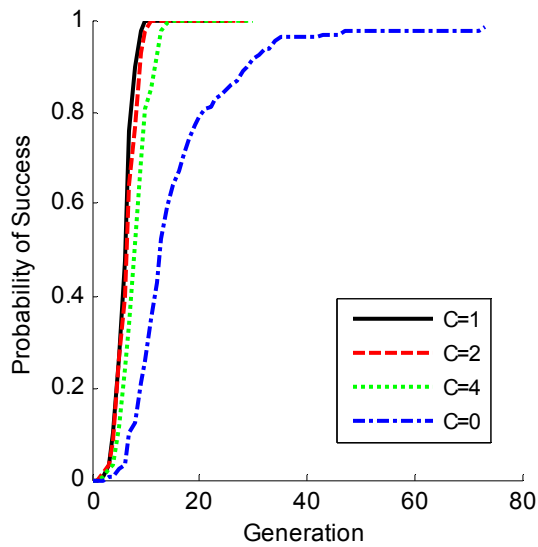


Figure 4 - Effect of migration cycle (C) on the probability of success for the polynomial regression problem.

Table 1 - Parameters and results for varying number of nodes applied to the polynomial regression problem.

Nodes Per Run	Programs Per Node	Seconds to Solve	Resource Seconds
16	8000	182.5	6,611.1
64	2000	42.4	5,636.1
128	1000	28.7	4,436.9
256	500	14.4	7,178.0

### 3.2 Finite Algebra

The second problem selected for study comes from the field of finite algebra. Spector [13] introduced this problem to GP researchers in 2008, demonstrating outstanding results where GP solutions were significantly better than human solutions. For this study, we selected the search for a discriminator operation using Spector's A1 grammar. The discriminator operation on algebra A is given by:

$$t^A(x, y, z) = \begin{cases} x & \text{if } x \neq y \\ z & \text{if } x = y \end{cases}$$

Using a dollar sign to represent the operator, Table 2 shows the single operator grammar for the three element algebra A1. The aspects of this problem that make it an interesting candidate for GP study are detailed in the following subsections.

Table 2 – Single operator results for three element algebra A1.

\$	0	1	2
0	2	1	2
1	1	0	0
2	0	0	1

#### 3.2.1 Data Set

The data set for this problem poses some unique challenges. Since it consists of all possible combinations of the three element grammar, it contains only 27 entries. This is a small number of test cases relative to the number of terms in the expected output. In addition, the result of a normal error calculation, such as that used in the polynomial regression problem, does not convey a distance meaning. That is, an output of 0 when it should be 2 (error = 2) is really no worse than an output of 1 when it should be 2 (error = 1).

#### 3.2.2 Fitness

This type of problem is well suited to the “percent incorrect” fitness evaluation available in GPE5. The calculated answer is deemed to be correct if it is within a specified tolerance ( $\tau$ ) of the actual output. We used  $\tau=0.01$ . The fitness function is:

$$f_p = \frac{100}{n} \sum_{i=1}^n (|p_i - o_i| > \tau)$$

While this eliminates the distance measure ambiguity, it exposes a dangerous limitation. With only  $n=27$  points to evaluate, the fitness result can only take on 27 discrete values. This is a very low resolution, effectively providing only 27 bins into which the programs for each generation would be sorted. However, the effective resolution for GPE5 will be somewhat better. When sorting programs, if the fitness of two programs is identical, the fitness of the shorter program is considered better.

#### 3.2.3 Operators and Terminals

The terminals for this problem are x, y, and z.

This problem requires only a single operator, but since it is not a standard math or logic operator, it would have to be coded into most GP systems. However, with GPE5, the configuration file allows the specification of pre-code that is executed prior to the program and which is neither included in the program length nor used in the genetic manipulation operations. That capability allowed us to define the \$ operator in FIFTH as follows:

```
<PreCode>
{ 2 1 2 | 1 0 0 | 0 0 1 } CONSTANT A1
VARIABLE aTable A1 aTable !
WORD $ SWAP HORZCAT aTable SWAP {}@ ENDWORD
</PreCode>
```

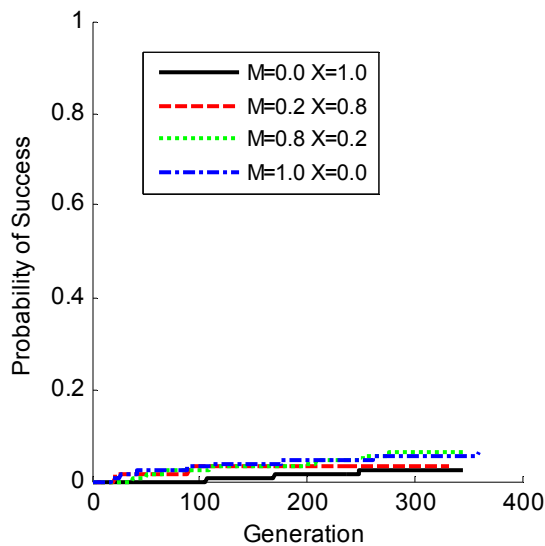
### 3.2.4 Problem Difficulty

With its small memory footprint and minimal number of operators and terminals, this problem might appear trivial. However, test runs rank it as moderately difficult. Using a population size of 1000 programs per generation, POS for a serial run is less than 5% within 400 generations with an average time expenditure of about 6 seconds per generation.

The best 100% correct discriminator term for algebra A1 reported by Spector required 3,210 generations and contained 39 terms. With that as a basis, we constrained the program size to be between 5 and 200 terms.

### 3.2.5 Experiments and Results

Figure 5 shows the effect on POS with constant migration cycle ( $C=0$ ) and processing nodes ( $n=128$ ) while varying the probability of mutation ( $M$ ) and crossover ( $X$ ). Although the lines are rather cramped, the  $M=0.8$  and  $X=0.2$  values perform marginally better.



**Figure 5 - Effect of mutation and crossover probabilities for the A1 grammar discriminator problem.**

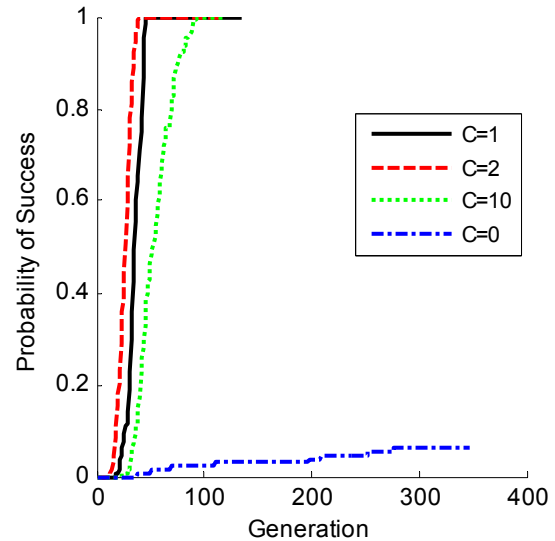
Figure 6 shows the dramatic effect on POS of varying the migration cycle. Note that migrating programs between nodes every generation ( $C=1$ ) did not perform as well as migrating every 2 generations ( $C=2$ ). In fact, several runs with  $C=1$  failed to produce any fit programs within 500 generations. With  $C \geq 2$ , the POS generally improved with lower values of  $C$ . The final experiment runs used  $C=2$ .

The final experiment to determine the effect of the number of processors on the wall clock time required to achieve a solution used the same constant work premise basis of 128,000 evaluations as the theoretical total per generation.

Table 3 shows the timing results for four runs. For this problem, the resource reduction decreased with every increase in the number of processors. Increasing from 16 to 256 processors reduced the wall clock time by 97% (35.7 minutes).

The best 100% correct discriminator program produced during these experiments contained 102 terms and was found in 72

generations. This does not represent the best solution of which GPE5 is capable. These studies were constructed to measure the time to the first correct solution. While the GPE5 configuration can be set to continue processing after reaching a solution (as long as fitness continues to improve), this potential improvement requires additional time and resource consumption.



**Figure 6 - Effect of migration cycle for the A1 grammar discriminator problem.**

**Table 3 - Parameters and results for varying number of nodes applied to the finite algebra problem.**

Nodes Per Run	Programs Per Node	Seconds to Solve	Resource Seconds
16	8000	2,196.4	38,562
64	2000	435.2	31,101
128	1000	107.8	17,666
256	500	53.6	17,545

## 3.3 Digital Signal Processing

The final problem addressed in this paper comes from the field of digital signal processing. This is a feature extraction task where the goal is to evolve an algorithm that highlights symbol transitions in a digital signal so that the symbol rate (baud) can be easily calculated. The output of the evolved program is expected to be a vector that if plotted would show significant peaks or valleys at the majority of the symbol transitions. The output vector is post processed using Fourier analysis to determine the symbol rate. To avoid having to write special code and compile it into GPE5, the post processing algorithm can be written in FIFTH and included in the problem configuration file as follows:

```
<TrainPostCode>
WND HAMMING FREQCOMP VMAXINDEX
x LENGTH SWAP DROP Fs SWAP / * . " , " .
symbolsPerSec .
</TrainPostCode>
```

This code expects an output vector on the stack. The symbol  $x$  is the signal vector, and  $F_s$  is the sampling frequency. The code extracts the frequency components, locates the maximum peak, and converts the peak index into a symbol rate. The result consists of two values required by the fitness calculation: the calculated symbol rate and the actual symbol rate. A complete description of the symbol rate problem may be found in reference [8].

### 3.3.1 Data Set

Symbol rate estimation typifies a large class of DSP problems where the raw data is best represented as a vector. This makes the data set management significantly more complicated than the previous problems. The first step is to create a collection of signals that adequately represents the domain of interest. Table 4 lists common signal, channel, and receiver properties that affect the performance of blind symbol rate estimation algorithms for surveillance systems working in the High Frequency (HF) range of 3-30 MHz. The last column indicates the range of values used in this experiment for each property.

**Table 4 - Properties of the training set signals used for the symbol rate experiments.**

Property	Typical values in HF	Values used
Modulation	FSK, MSK, PSK, DPSK, OQPSK, QAM, ASK	FSK, PSK
Pulse shape	None, raised cosine (RC), root RC (RRC), Gaussian	None, RC
Excess bandwidth (rolloff)	Limit: 0.00 to <1.00. Typical: 0.10 to 0.35	0, 0.1, 0.2, 0.35
Mod Index	0.5 to 3	0.1
Symbol rate	Typical: 10 to 2400 symbols per second	13 values in [50, 2400]
Symbol states	2, 4, 8	2, 4, 8
Signal to noise ratio (SNR)	Practical range: 0 to 60 dB	12 dB

We used signal simulation software written in MATLAB to produce five random variations of each combination of the above properties, resulting in 140 FSK signal files and 780 PSK signal files. Using a sample size of approximately two seconds and a sample rate of 8000 Hz, the signal vector in each file consisted of 16384 complex valued data points. Using the MATLAB V4 file format allowed all of the signal metadata to be stored in the same file. This required approximately 256 KB per file for a total disk storage of 235 MB.

It is not unusual for this type of problem to require thousands of signal files and multiple gigabytes of disk space, making it unlikely that the entire data set can be stored in RAM. GPE5 handles large data sets by using configuration parameters to specify directory locations, file extensions, and file types.

### 3.3.2 Fitness

The symbol rate problem is another example of where the typical error calculation does not represent the desired characteristics of the evolved algorithm. In this case, the relative error is more

significant than the absolute error. For example, a 10 baud error when the actual rate is 50 baud represents a much more significant inaccuracy than if the actual rate is 2400 baud. The relative error fitness function in GPE5 is defined as:

$$f_p = \frac{1}{n} \sum_{i=1}^n \frac{|p_i - o_i|}{o_i}$$

### 3.3.3 Operators and Terminals

Table 5 lists the terminals and operators used for this problem along with the probability of selection during initial random program generation and mutation. The last row includes a number of vector transformation operators, both general and domain specific.

**Table 5 – Terminals and operators used for the symbol rate problem.**

Terminals and Operators	Probability of selection
X	0.1
1, 2, 3, 5, -1, PI	0.1
+ - * /	0.1
REAL IMAGINARY ANGLE UNWRAP MAGSQRD MAGNITUDE DIFF FFT SQRT VRAMP ROUND REVERSE COS SIN WND_HAMMING FLOOR CEILING	0.7

### 3.3.4 Problem Difficulty

The symbol rate problem is several orders of magnitude more difficult than either symbolic regression or finite algebra. Runs on a serial machine often require days of wall clock time. A typical run requires about 40 minutes per generation for a population of 1000 programs. These runs exhibit a high failure rate with a POS of < 1%.

### 3.3.5 Experiments and Results

Following the same pattern as the previous experiments, we ran four trials with 0 migration cycle, 128 processors, 1000 programs per generation, and varying mutation and crossover rates. None of these runs produced a single fit program. Each run lasted 24 hours and processed an average of 30 generations per node.

Since earlier experiments obtained reasonable results with  $C=2$ ,  $M=0.8$ , and  $X=0.2$ , we decided to use these parameters with a population size of 500 programs and 128 compute nodes. Figure 7 shows the results. The first fit program appeared in generation 51, and by generation 87, all nodes had produced fit programs.

For this problem, the final experiment was the most important. Anticipating an extended run with 16 processors, we reduced the constant work load to a theoretical 64,000 evaluations per generation, as shown in Table 6.

For this problem, the time savings is impressive from any perspective. Increasing the number of processors from 16 to 256 reduces the time to solve by 92%, or in more meaningful units, from 37.5 hours (~1.5 days) to 3 hours.

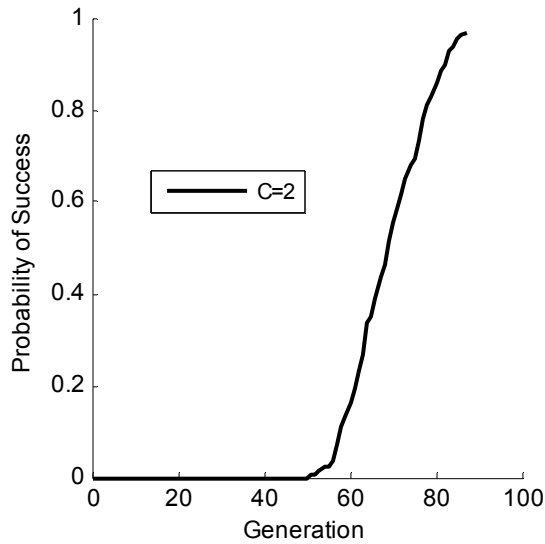


Figure 7 - Effect of migration cycle for the symbol rate problem.

Table 6 - Parameters and results for varying number of nodes applied to the symbol rate problem.

Nodes Per Run	Programs Per Node	Seconds to Solve	Resource Seconds
16	4000	135,195	2,449,130
64	1000	59,502	4,116,137
128	500	22,472	3,234,650
256	250	10,781	3,030,568

#### 4. DISCUSSION

Not unexpectedly, the migration cycle had a dramatic positive effect on both the probability of success and the wall clock time required to achieve a solution for all three test problems. Since the island model can be easily simulated on a single processor, the next release of the standalone version of GPE5 will include that option.

Similar to the results in [4], there was a poisoning effect with a cycle of 1 on the primal algebra problem. It is possible that this early sharing occasionally caused premature convergence to non-optimal solutions which would then propagate rapidly and prevent any node from producing a program that met the fitness criterion. Some papers that use demes assume a sharing cycle of one as standard. An interesting future research topic would be to determine whether this phenomenon affects other types of problems.

The principal motivation behind this work was the need to significantly reduce the wall clock time required to solve complex, vector-oriented problems. Figure 8 is a plot of the time to first solution as a function of the number of processing nodes for all three of the problems covered in this paper. Note the logarithmic time scale. While both the symbolic regression and finite algebra problems exhibit large percentage reductions in run time, the actual savings amount to only a few minutes. For real-

world problems of equivalent difficulty, this level of savings is unlikely to justify the annoying details and potential expense associated with obtaining time on an HPC system.

However, the time savings for the symbol rate problem is much more dramatic and meaningful, reducing the expenditure per run from days to a few hours. If fast turn around is a high priority, then using an HPC system could easily be justified.

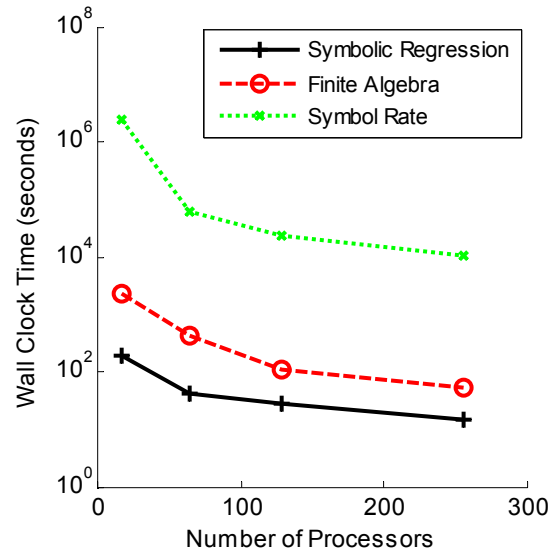


Figure 8 - Effect of the number of processors on the time to achieve first solution.

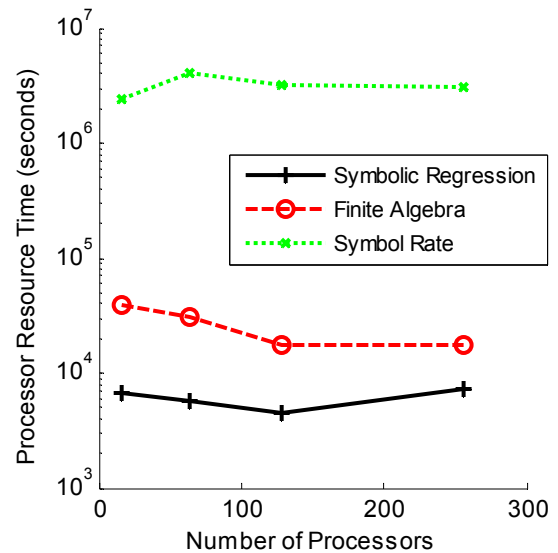


Figure 9 - Effect of the number of processors on the resource time required to achieve first solution.

Another interesting result of this work for the symbol rate problem was the relative invariance of the processor resource requirements as a function of the number of processors, as shown in Figure 9. This suggests that the most efficient use of resources involves maximizing the number of processors and minimizing the number of programs per generation. We found that a few test

runs usually provide reasonable bracket values for these parameters.

Finally, if minimizing the turn around time for a run is important, the anticipated wall clock time may not be the only factor to consider. For a shared, batch submission HPC system, there is also the time a job spends in a queue waiting to be scheduled for execution. When a system is busy, a job that requests fewer processors for a longer period of time usually has a higher scheduling priority than a job requesting a higher number of processors for a shorter period of time. For several runs of the symbol rate problem, when requesting 512 processors (the maximum normal queue request allowed on Lonestar), the queue time exceeded 30 hours, essentially negating the run time savings.

## 5. ACKNOWLEDGEMENTS

This work was sponsored by the Southwest Research Institute Advisory Committee for Research (ACR).

High Performance Computing resources were provided by the Texas Advanced Computing Center (TACC) at The University of Texas at Austin (<http://www.tacc.utexas.edu>).

## 6. REFERENCES

1. Andre, D. and Koza, J.R. A parallel implementation of genetic programming that achieves super-linear performance. Arabnia, H.R. ed. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, CSREA, Sunnyvale, 1996, 1163--1174.
2. Daida, J.M., Bertram, R.R., A, J. and Stanhope, S.A. Analysis of Single-Node (Building) Blocks in Genetic Programming. Spector, L., Langdon, n.W.B., O'Reilly, n.U.-M. and Angeline, n.P.J. eds. *Advances in Genetic Programming 3*, MIT Press, Cambridge, MA, USA, 1999, 217--241.
3. Daida, J.M., Bertram, R.R., Stanhope, S.A., Khoo, J.C., Chaudhary, S.A., Chaudhri, O.A. and A, J. What Makes a Problem GP-Hard? Analysis of a Tunably Difficult Problem in Genetic Programming *Genetic Programming and Evolvable Machines*, 2001, 165--191.
4. Fernandez, F., Tomassini, M. and Vanneschi, L. An Empirical Study of Multipopulation Genetic Programming *Genetic Programming and Evolvable Machines*, 2003, 21--51.
5. Fernandez, F., Tomassini, M., Vanneschi, L. and Bucher, L. A Distributed Computing Environment for Genetic Programming using MPI. Dongarra, J.J., Kacsuk, n.P. and Podhorszki, n.N. eds. *Recent advances in parallel virtual machine and message passing interface: 7th European PVM/Slash MPI Users' Group Meeting*, Springer-Verlag, Balatonfured, Hungary, 2000, 322--329.
6. Folino, G., Pizzuti, C. and Spezzano, G. A Scalable Cellular Implementation of Parallel Genetic Programming *IEEE Transactions on Evolutionary Computation*, 2003, 37--53.
7. Holladay, K., Robbins, K. and Ronne, J.v. FIFTH™: A stack based GP language for vector processing. Ebner, O.N., Ekart, Vanneschi and Esparcia Alcazar ed. *EuroGP 2007*, Springer, Valencia, Spain, 2007.
8. Holladay, K.L. and Robbins, K.A., Evolution of Signal Processing Algorithms using Vector Based Genetic Programming. in *Digital Signal Processing, 2007 15th International Conference on*, (Cardiff, Wales, 2007), 503-506.
9. Koza, J.R. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, 1994.
10. Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, 1992.
11. Potter, M.A. and Jong, K.A.D. Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents. *Evolutionary Computation*, 8 (1). 1-29.
12. Snir, M. *MPI--The Complete Reference*. MIT Press, Cambridge, Mass., 1998.
13. Spector, L., Clark, D.M., Lindsay, I., Barr, B. and Klein, J. Genetic programming for finite algebras *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, ACM, Atlanta, GA, USA, 2008.
14. Spector, L., Klein, J. and Keijzer, M., The Push3 Execution Stack and the Evolution of Control. in *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, (Washington DC, USA, 2005), ACM SIGEVO (formerly ISGEC) ACM Press New York, NY, 10286-1405, USA, 1689--1696.
15. Tchernev, E. Stack-Correct Crossover Methods in Genetic Programming. Cant'u-Paz, E. ed. *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)*, AAAI 445 Burgess Drive, Menlo Park, CA 94025, New York, NY, 2002, 443--449.