High-Performance, Parallel, Stack-Based Genetic Programming
by Kilian Stoffel and Lee Spector

Full citation:
Stoffel, K., and L. Spector. 1996. High-Performance, Parallel, Stack-Based Genetic Programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors) *Genetic Programming 1996: Proceedings of the First Annual Conference*, 224-229. Cambridge, MA: The MIT Press.

High-Performance, Parallel, Stack-Based Genetic Programming

Kilian Stoffel*
stoffel@cs.umd.edu

Lee Spector* †
lspector@hampshire.edu

*Department of Computer Science
University of Maryland
College Park, MD 20742

†School of Cognitive Science and Cultural Studies
Hampshire College
Amherst, MA 01002

ABSTRACT

HiGP is a new high-performance genetic programming system. This system combines techniques from string-based genetic algorithms, S-expression-based genetic programming systems, and high-performance parallel computing. The result is a fast, flexible, and easily portable genetic programming engine with a clear and efficient parallel implementation. HiGP manipulates and produces linear programs for a stack-based virtual machine, rather than the tree-structured S-expressions used in traditional genetic programming. In this paper we describe the HiGP virtual machine and genetic programming algorithms. We demonstrate the system's performance on a symbolic regression problem and show that HiGP can solve this problem with substantially less computational effort than can a traditional genetic programming system. We also show that HiGP's time performance is significantly better than that of a well-written S-expression-based system, also written in C. We further show that our parallel version of HiGP achieves a speedup that is nearly linear in the number of processors, without mandating the use of localized breeding strategies.

1 Performance of Genetic Programming Systems

Genetic programming is a technique for the automatic generation of computer programs by means of natural selection [Koza 1992]. The genetic programming process starts by creating a large initial population of programs that are random combinations of elements from problem-specific function and terminal sets. Each program in the initial population is then assessed for fitness, and the fitness values are used in producing the next generation of programs via a variety of genetic operations including reproduction, crossover, and mutation. After a preestablished number of generations, or after the fitness improves to some preestablished level, the best-of-run individual is designated as the result and is produced as the output from the genetic programming system.

The performance impact of alternative approaches to genetic programming can only be assessed by measuring performance over a large number of runs. This is because the algorithm includes random choices at several steps; in any particular run the effects of the random choices may easily obscure the effects of the alternative approaches.

To analyze the performance of a genetic programming system over a large number of runs one can first calculate *P*(*M*,*i*), the cumulative probability of success by generation *i* using a population of size *M*. For each generation *i* this is simply the total number of runs that succeeded on or before the *i*th generation, divided by the total number of runs conducted. Given *P*(*M*,*i*) one can calculate *I*(*M*,*i*,*z*), the number of individuals that must be processed to produce a solution by generation *i* with probability greater than *z*.[1] *I*(*M*,*i*,*z*) can be calculated using the following formula:

$$I(M,i,z) = M * (i+1) * \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil$$

Koza defines the minimum of *I*(*M*,*i*,*z*) as the "computational effort" required to solve the problem with the given system.[2]

2 High Performance Genetic Programming

Genetic programming is a computationally intensive enterprise, and recent work has focused on ways in which high performance computation techniques can increase the set of problems to which it can be applied. Since the underlying genetic algorithm appears to be parallelizable, and since several parallel implementations of string-based genetic algorithms have been described in the literature, there has been

[1] For the analyses in this paper a value of *z*=99% is always used.

[2] The *P*(*M*,*i*) and *I*(*M*,*i*,*z*) measures were developed by Koza and are discussed on pages 99 through 103 of [Koza 1994].

particularly strong interest in parallelizing genetic programming systems.

Most previous high performance genetic programming systems have either been wedded to particular computer architectures, or have been hampered by high overheads in distributing fitness evaluation and in coordinating reproduction across parallel machines. For example, Koza and Andre have implemented an S-expression-based genetic programming system on a network of transputers and have achieved considerable speed improvements over serial implementations [Koza and Andre 1995]. Their approach relies on a hardware base that is less expensive than parallel supercomputers but is nonetheless somewhat exotic. They used local breeding strategies with migration; they report that this reduces computational effort as compared with a panmictic (globally interbreeding) population on the problems they tried. Others have reported similar improvements from the use of localized breeding strategies, both in genetic programming and in string-based genetic algorithms (e.g., [Collins and Jefferson 1991]). It is not yet clear, however, that particular localized breeding strategies are always beneficial, and it is important that high-performance genetic programming systems not be swamped with communication overheads when less localized or global (panmictic) strategies are used.

Nordin and Banzhaf have developed a "compiling" genetic programming system that directly manipulates SPARC machine code [Nordin and Banzhaf 1995]. Through a combination of compact fixed-length representations, simplified memory management, and non-interpreted program execution they have achieved dramatic speed improvements over traditional genetic programming systems. Their system depends on details of the SPARC architecture and would require significant redesign for other machines.

Juillé and Pollack have developed a SIMD (single instruction multiple data) parallel genetic programming system that represents programs as S-expressions but "precompiles" them into programs for a virtual stack machine prior to execution [Juillé and Pollack 1995]. They note that the variation in S-expression sizes across the population can introduce overhead, and they use local breeding strategies to reduce interprocessor communication costs.

S-expression-based program representations are responsible for several of the performance limitations of previous systems. These representations make strong demands on a system's memory allocation subsystems and they are expensive to manipulate and to move between processors. In addition, less prior research has been conducted on the optimization and parallelization of S-expression interpreters than on techniques for more common models of computation.

These limitations of previous systems lead us to consider the idea of genetic programming with small, linear programs that can be executed on stack-based virtual machines for which portable, high performance interpretation techniques have already been developed. We were encouraged by previous work that explored the performance benefits of linear program representations (e.g., [Keith and Martin 1994]). We were further encouraged by Perkis's work on stack-based genetic programming [Perkis 1994], in which he used linear programs that were executed on a stack-based virtual machine. In contrast to the work of Keith and Martin and of Juillé and Pollack, Perkis performed string-based genetic operations (e.g. string-based crossover) directly on the linear programs. The safety of the resulting programs was guaranteed by specifying that all functions take their arguments from the stack, and that function calls that occur with too few items on the stack simply do nothing. Using this scheme, Perkis reported lower computational efforts than were required using traditional S-expression-based genetic programming.

In HiGP we have combined elements of the stack-based program frameworks of Perkis and of Juillé and Pollack, the linear chromosomes of string-based genetic algorithms (also used in Nordin and Banzhaf's and Perkis's systems), the efficient, low level execution model of Nordin and Banzhaf, a machine independent virtual stack machine, and high-performance parallel programming techniques.

3 The HiGP Virtual Stack Machine

HiGP programs are executed on a virtual machine that is similar to a pushdown automaton. The virtual machine consists of three components: an input tape containing a linear program, a pushdown stack, and a finite-state control unit. The contents of the input tape are restricted to a small set of words that have been defined as HiGP operators. The contents of pushdown stack are restricted to double precision floating point numbers. The finite-state control unit reads the input tape and executes, for each word, the function call for the corresponding operator. The operators may perform arbitrary computations and manipulate the values on the stack. They may also reposition the read head on the input tape; this allows for the implementation of conditionals and loop structures. Return values are generally read from the top of the stack at the end of program execution.

The system includes two basic stack operators, `pop` and `dup`. The `pop` operator removes the topmost element from the stack, while the `dup` operator pushes a duplicate of the top element onto the stack. The system also includes a family of `push` operators that correspond to the terminal set in a traditional genetic programming system; each `push` operator pushes a single pre-determined value onto the stack.

The system also includes a `noop` operator that does nothing. This is necessary because all programs in the system have the same length, and because we do not wish to predetermine the number of actual problem-solving operators that should appear in solution programs. With the inclusion of the `noop` operator the fixed program size becomes a size

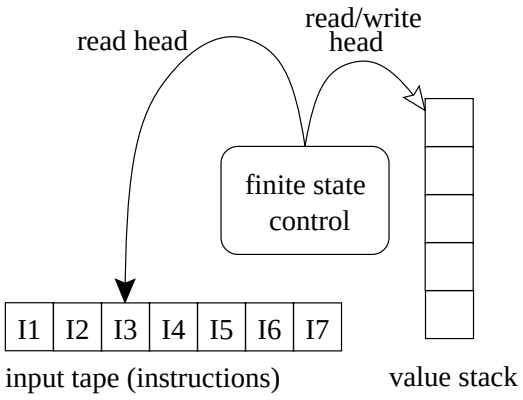

Figure 1: The virtual stack machine

limit, analogous to the depth limits used in S-expression-based genetic programming systems. "Shorter" programs are encoded by filling in extra program steps with `noop`s. Note that this provides the same flexibility with respect to program size as do S-expression-based genetic programming systems. If one suspects that a large program may be required, then one can set the program size to an arbitrarily large number. In such a case, however, the system may still produce efficient, parsimonious programs; it may do so by producing programs that consist mostly of `noop`s.

Any additional, problem-specific operators must take their input values from the stack and must push their results back onto the stack. When there are not enough values on the stack for an operator it is skipped by the finite-control unit and the stack remains untouched (as in [Perkis 1994]).

A simple example may help to clarify the operation of the virtual stack machine. Consider the following program:

```
push-x noop push-y * push-x push-z noop
- + noop noop
```

The `noop`s in this program have no effect and the remainder is equivalent to the Lisp expression:

```
(+ (* x y) (- x z))
```

and to the C expression:

```
(x * y) + (x - z)
```

For the test examples presented in this paper we used only the four basic arithmetic operators: `+` (addition), `-` (subtraction), `*` (multiplication) and `%` (protected division [Koza 1992]). Although conditionals, loops, and other control structures were not used for the examples in this paper, several have been implemented for HiGP. For example, one implemented control structure reads a value from the stack and executes the next instruction if the value, when truncated, is zero; otherwise it jumps over the next instruction. Another control structure jumps over a number of instructions; the number is obtained by truncating the value on the top of the stack. These control structures, along with several other features of our virtual stack machine, are similar to those of the FORTH programming language [Brodie 1981]. The extension of the virtual stack machine with additional operators and control structures is trivial; the FORTH language provides examples of what such extensions might look like.

4 Genetic Programming in HiGP

One nice feature of the HiGP program representation is that one can use the same genetic operators that are used in string-based genetic algorithms. Each gene in the chromosome represents an operator for the virtual machine. Since the operators communicate with one another only by means of the stack, and since stack-underflow is handled gracefully (by ignoring offending operators), every possible ordering of operators represents a syntactically correct program. Arbitrary string-based manipulations of the chromosomes are therefore acceptable. In fact, it would be possible to use many existing, off-the-shelf genetic algorithm packages to perform the genetic operations on HiGP programs. In the current implementation we use three standard genetic operators: reproduction, single-point crossover, and point mutation. The population is initialized by setting each gene in each chromosome to a random member of the operator set.

An additional consideration for genetic programming is the implementation of ephemeral random constants [Koza 1992]. In HiGP constants are not included directly in programs (chromosomes) because a small, fixed gene size is important for optimized, high-performance execution and manipulation of HiGP programs. Because some applications call for constants with representations that are larger than the optimal gene size (for example, double precision floating-point constants in an application with a small number of operators and constants) HiGP maintains a table in which constants are stored. Inside the programs the constants are represented by their positions in the table. The "execution" of a constant causes the corresponding value from the table to be pushed onto the stack. The ephemeral random constants in the table are initialized at the beginning of a run and remain unchanged during the entire run; this differs from Koza's implementation, in which new constants may be generated by the mutation operator at any point in a run. Our point mutation operator can transform one constant into another, or into an operator, but the set of available constants does not change during a run. The size of the constant table is limited only by the size of the alphabet for the virtual stack machine, which is in turn determined by the operator word-size. Because the word-size is a variable parameter in the current implementation, any number of ephemeral random constants can be supported.

5 Advantages of the HiGP approach

HiGP has several advantages over traditional S-expression-based genetic programming systems. HiGP programs are of fixed length and our experience is that they can be quite small. We have solved several problems with programs only 32 operators in length, with less than a byte required for each operator. This makes it easy to optimize memory allocation, to create populations with millions of individuals, and to efficiently pass programs between processors in multiprocessor architectures. Load-balancing during fitness evaluation is also eased by the fixed program length. These factors combine to allow for nearly linear speedups with added processors, even when a global breeding strategy is used. Of course many problems will require longer programs, but all components of the system should scale at worst linearly with program length.

In addition, the evolutionary dynamics of systems of linear programs appear in some cases to be superior to those of S-expression-based genetic programming (e.g., in [Perkis 1994] and below). We do not yet have a theoretical explanation of why this should be so.

6 Example: Symbolic Regression

The goal of the symbolic regression problem, as described in [Koza 1992], is to produce a function that fits a provided set of data points. For each element of a set of (x, y) points, the program should return the correct y value when provided only with the x value. For our experiments we obtained our data points from the equation $y = x^9$. One can view the task of the genetic programming system as that of "rediscovering" this formula from the data points used as fitness cases. We used 20 fitness cases, with randomly selected x values between -1 and 1.

We used a function set consisting of the two-argument addition function `+`, the two-argument subtraction function `-`, the two-argument multiplication function `*`, and the two-argument protected division function `%`. We used a single "terminal" operator `push-x` that pushes the x value onto the stack. We did not use ephemeral random constants or any other constants. We ran HiGP with tournament selection (tournament size = 5), a 90% crossover rate, a 10% reproduction rate, and no mutation. We conducted 100 runs, each with a population size of 500, and each for a maximum of 30 generations. The length of each individual program was set to 32 operators.

We also conducted 100 runs of the lil-gp S-expression-based genetic programming system [Zongker 1995] on this problem. We used parameters that were as similar as possible to those used for HiGP: function set={`+, -, *, %`}, terminal set={`X`}, tournament selection (size=5), 90% crossover, 10% reproduction, no mutation, 100 runs, population=500, generations=30. The depth limit presented a more difficult problem since the HiGP concept of *length* limit and the lil-gp concept of *depth* limit are quite different. Lowering the depth limit in lil-gp improves its execution speed considerably, but in some cases this may make it more difficult or impossible to find correct programs. It is not clear, in general, how the depth limit influences computational effort. For this reason we ran lil-gp with a range of depth limits. We used a depth limit of 4 for the low end of the range; this is the lowest limit for which lil-gp was able to find any correct programs at all. We used 17 for the high end of the range; this is the default for lil-gp and, according to our data, much larger than optimal.

Table 1: Results for HiGP and for lil-gp using different depth limits

		time/gen.	*comp. effort*
HiGP		0.08	32,729
	depth		
lil-gp	4	*0.16*	212,521
	5	0.18	135,933
	6	0.21	71,481
	7	0.23	93,013
	8	0.25	82,210
	9	0.28	66,323
	10	0.30	*65,216*
	11	0.30	70,116
	12	0.34	77,030
	13	0.36	76,404
	14	0.35	71,481
	15	0.38	92,436
	16	0.39	90,379
	17	0.41	79,737

We believe that lil-gp, which is implemented in C, is a well-written program that provides a reasonable comparative benchmark for S-expression-based genetic programming. One way to establish the relative time performance of the two systems would be to compare the times that each system takes for a complete set of 100 runs. However, the number of generations computed by the two systems are different, as indicated by the computational effort results reported below. To remove this factor from the time performance comparisons we computed the time used by each system to evaluate one generation. For HiGP the execution time per generation is nearly constant, because of the fixed length of the programs and because of the absence of conditionals or looping operators in the test problem. For lil-gp the time for the execution of one generation can change significantly as the structures of the S-expressions evolve. We therefore used the mean value over 100 runs for all evaluated generations.

Our results, obtained on a Sun SPARC 5, are summarized in Table 1. HiGP was clearly superior both with respect to

computational effort and with respect to time performance. The best computational effort for lil-gp was obtained with a depth limit of 10, but even this effort (65,216) was nearly double that required by HiGP (32,729). In addition, the time performance of lil-gp with a depth limit of 10 was quite poor; the time required per generation was 0.30 seconds, while HiGP required only 0.08 seconds per generation. The best time performance for lil-gp was obtained with a depth limit of 4, but even this time (0.16 seconds) was double that of HiGP (0.08 seconds). In addition, the computational effort required by lil-gp with a depth limit of 4 was quite poor; it was 212,521, which is over six times greater than the effort required by HiGP (32,729).

7 Parallel HiGP

The parallel version of HiGP was developed to run on MIMD (multiple instruction multiple data) supercomputers, but it was not tailored for any one particular system. We used the MPI (message passing interface) communication library, which is available for many different parallel computer systems. This greatly simplifies the process of porting HiGP to various different computer systems. The system was written within the SPMD (single program multiple data) paradigm; all nodes of the parallel system execute identical programs but on different data.

The population is distributed evenly across the processors, with each processor maintaining its own local copy of a fitness table for the entire population. This allows much of the breeding process (including selection) to occur locally, along with fitness evaluation.

During the fitness evaluation phase, all processors evaluate the fitness of their individuals in parallel. Because each processor hosts the same number of programs, and because all of the evolving programs are the same length, all of the processors complete fitness evaluation at nearly the same time. Load balancing may be more complicated for applications with function sets that include functions of widely varying runtimes, and for applications that include conditional and looping operators in their function sets. At the end of the fitness evaluation phase all computed fitness values are exchanged between processors and all fitness tables are updated.

During the breeding phase each processor independently produces its own segment of the next generation. The choice of which genetic operators to apply is determined locally, using the percentages specified for the whole population. For example, if it has been specified that 60% of the next generation should be produced by crossover, 30% by reproduction, and 10% by mutation, each processor can choose operators according to these percentages locally and the correct global percentages will result. Selection is also performed locally, using the local fitness tables to perform either fitness-proportionate or tournament selection. Interprocessor communication is only required to exchange the actual individuals that have been selected for use in genetic operations (and then only when they are not local).

Notice that the breeding strategy is global (panmictic) even though selection occurs locally on individual processors. The need for interprocessor communication is minimized, and there are low communication overheads both for the distribution of fitness values and for the transportation of the small linear programs. Notice also that it would be trivial to modify HiGP to make use of localized breeding strategies; although little would be saved with respect to communication costs, improvements in computational effort might result for certain problems.

nodes	time	Efficiency
1	91.41	—
2	47.12	96.9
4	27.32	83.6
8	16.82	67.9
16	10.89	53.4

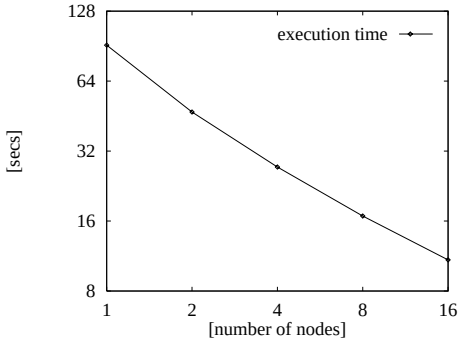

Figure 2: Parallel execution times for one run on up to 16 processors on an IBM SP2.

The HiGP strategy leads to a very efficient implementation, as Figure 2 shows. We were able to execute a program which took 91.41 seconds on one single node of the IBM SP2 nearly 9 times faster on 16 nodes in 10.89 seconds. This leads to an efficiency (number of times faster divided by number of nodes) of more than 50%, which is good for a communication-intensive program. Note also that the graph in Figure 2 is nearly linear.

8 Conclusions

HiGP is a new high-performance genetic programming system that combines techniques from string-based genetic algorithms, S-expression-based genetic programming systems, and high-performance parallel computing. HiGP is fast, flexible, and easily portable. It also has a clear and efficient parallel implementation that is not tied to any particular

parallel computer architecture.

HiGP manipulates and produces linear programs for a stack-based virtual machine, rather than the tree-structured S-expressions used in traditional genetic programming. This appears to have several benefits; along with the optimizations that it allows for memory management and parallelization, it appears to decrease the computational effort required to produce a correct program, at least for the regression problem that we presented.

We showed that HiGP's time performance is significantly better than that of lil-gp, a well-written S-expression-based system which is also written in C. We further showed that our parallel version of HiGP achieves a speedup that is nearly linear in the number of processors, without mandating the use of localized breeding strategies.

Acknowledgments

The research described in this paper was supported in part by grants from ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), and ARPA contract DAST-95-C0037.

Bibliography

Brodie, L. 1981. *Starting FORTH*. Prentice Hall.

Collins, R.J. and D.R. Jefferson. 1991. Selection in Massively Parallel Genetic Algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by R.K. Belew and L.B. Booker. pp. 249–256. San Mateo, CA: Morgan Kaufmann Publishers.

Juillé, H., and J.B. Pollack. 1995. Parallel Genetic Programming on Fine-Grained SIMD Architectures. In *Working Notes of the AAAI-95 Fall Symposium on Genetic Programming*. Eric V. Siegel and John R. Koza, editors. AAAI Press.

Keith, M.J., and M.C. Martin. 1994. Genetic Programming in C++: Implementation Issues. In *Advances in Genetic Programming*, edited by Kenneth E. Kinnear, Jr. pp. 285–310. Cambridge, MA: The MIT Press.

Koza, J.R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Koza, J.R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: The MIT Press.

Koza, J.R., and D. Andre. 1995. Parallel Genetic Programming on a Network of Transputers. Stanford University, Department of Computer Science Technical Report CS-TR-95-1542.

Nordin, P., and W. Banzhaf. 1995. Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code. In *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA-95)*. Larry J. Eshelman, editor. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

Perkis, T. 1994. Stack-Based Genetic Programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pp. 148–153. IEEE Press.

Zongker, D., and B. Punch. 1995. *lil-gp 1.0 User's Manual*. WWW: http://isl.cps.msu.edu/GA/software/lil-gp, or via anonymous FTP to isl.cps.msu.edu, in the directory "/pub/GA/lilgp".