Evolving Parallel SISAL Programs Using GP

V. Rao Vemuri and Patrick Miller
University of California
Lawrence Livermore National Laboratory

Introduction

The genetic programming (GP) paradigm is an off shoot of genetic algorithms (GA). GP is envisaged as an automatic method for generating computer programs. In GP we use populations of data structures (or, programs) that are evaluated by some problem specific criterion. More fit structures are propagated to future generations of populations through genetic operations that are similar to those used in GAs [Koza, 92, 94].

In the early implementations of the GP paradigm, Lisp was the language of choice. The program structures of GP are stored as Lisp symbolic expressions (S-expressions) and are manipulated such that better and better program structures are evolved, preserved and propagated until a desired structure capable of solving a given problem is generated. That Lisp has been a successful vehicle for the implementation of GP is a proven fact [Koza, 92, 94].

The Case Against C

Lisp, although a common programming language, is not well known outside the computer science community. Even among computer scientists, its popularity pales in front of other languages like C. Perhaps a larger group would try GP if it can be implemented in a more popular language like C or FORTRAN. Spurred by this type of motivation there have been attempts to implement GP in C (Andre and Hondl, 1994; Zongker and Punch, 1995).

Our first desire was to try to implement GP in C to gain some first hand experience. After some initial thinking we soon abandoned that idea in favor of SISAL. The reasons for our initial desire to try C and subsequent switch from C to SISAL are many. Suffice it to say that we felt that C, in spite of its popularity, is not a good language for GP implementations. "To be 'using C,' an implementation must use as its chromosomal material , C source code. It must store these programs, be able to successfully perform the operations of mating crossover, mutation (while preserving each individual as a functioning C program). The compositions of the sets of terminals and the sets of operators must be determined and specified. There are a great many details to specify." (Alme, 1995).

"The C language, in source code form, has a very rigid structure that must be obeyed. There are many details that must be addressed: variables must be declared and their data types specified, function calls must have the correct number of arguments, unary operators must operate on single objects, binary operators on two, (*sic)* and so on. The placement of parentheses and semicolons is important. Unlike a Lisp expression, which *is* a program, a C source file is simply a list of instructions for the computer to generate a program." Finally we felt that C suffers from, for the lack of a better term, "syntactic fragility." That is performing GP operations in C source code, without abstraction is unlikely to produce another source file that will pass through the compiler without error. Lisp expressions, on the other hand, look like trees and there is a one-to-one correspondence between the expressions that the investigator sees and what the computer sees. This is not the case for the C language."

We looked at two implementation attempts (Andre and Hondl, 1994; Zongker and Punch, 1995) to understand what others are doing. We felt that these two implementations did not address, to the extent we understand them, the issues that were raised here. Both implementations form a tree where the terminal nodes and operators are functions that are written in C, but they do not directly manipulate the C source code. "Once the source code for the pieces of the tree have been written and successfully compiled, the program will manipulate individuals whose smallest components are the precompiled functions specified by the investigator. While they are certainly effective implementations that use the GP technique, they do not conform to the specification of 'using C.' They are written in C, but so is the Lisp interpreter that manipulates the Lisp S-expressions" (Alme, 1995).

Motivation for SISAL

This initial attempt and our subsequent lack of conviction that C would be a good language prodded us to look at other alternatives. One immediate candidate happened to be SISAL, a language developed by a team that included a group from Lawrence Livermore National Laboratory. The motivations for the selection of SISAL are different from those behind the initial trials with C. SISAL definitely is not a language that is in widespread usage; it is still an experimental language. The first version was designed in 1983. At the time of this writing SISAL has been installed at well over 70 sites internationally.

That we abandoned our effort with C and embarked on an exploration of SISAL is no absolute verdict against C or any other language. Granted that SISAL is lot more obscure than Lisp. Our desire to experiment with SISAL is motivated because it is a parallel language. Although it can run in a uniprocessor environment, its speed advantage is fully realized when used on parallel architectures. Finally, we wanted to learn the fallacies and pitfalls of evolving parallel programs and for this objective SISAL is as good as any other parallel language.

SISAL (Streams and Iterations in a Single Assignment Language) is a functional language that takes advantage of parallel architectures (McGraw, et al. 1985). It is a proven high performance, parallel language system (Cann, 1992). The SISAL 90 language (Feo, et. al. 1995) has added higher order objects making the language capable of defining and manipulating run-time structures. SISAL defines and uses pure functions that map input data to results. These would form the basis for functional chromosomes much as with LISP. Recursive composition of functions is straightforward. SISAL has important capabilities that may be important in GP. These include iterative constructs that may be more effective than recursion, automatic parallelization and vectorization, a static typing system that makes compilation more effective, and the tree based intermediate form IF1 (Skedzielewski, et. al, 1985).

Unlike GP strategies in C where only small atomic functions may be easily manipulated at the source level, SISAL may be manipulated using its hierarchical intermediate. This IF1 sub language is conceptually identical to S-expressions in lisp. Variable declarations are unnecessary because they are implicit in its structure. Programs can be built up from the language's atomic operations and user functions (the genes) into complex chromosomal structures within the intermediate's hierarchical operations (iteration, parallel do-across, and alternation). GP generated programs in the intermediate form can also be "uncompiled" into SISAL source. We feel that the SISAL/ IF1 combination will be easier to manipulate than C and will yield higher-performance, parallel programs that are not obtainable using Lisp.

Acknowledgment:

The authors were benefited by discussions with Tom DeBoni of Lawrence Livermore National Laboratory. They thank Prof. John Koza for inspiring us to undertake this work.

References

Alme, H. J. (1995) Computerized Darwinism: Applying genetic programming to C source code, Term paper submitted in partial fulfillment of the requirements of AEL216G taught by Prof. V. Vemuri in Winter 1995.

Andre, D. and Hondl, C. (1994) Dave's Genetic Programming Code in C, Informal Report, Stanford University, April 28, 1994.

Cann, D. C. "Retire FORTRAN: A Debate Rekindled", CACM, August 1992

Feo, J. T. Miller, P. J. and Skedzielewski, S. K. (1995) "SISAL90", Proc. High Performance Parallel Computing '95, Denver, Colorado. April 1995.

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

Koza, J. R. (1994) Genetic Programming II: *Automatic Discovery of Reusable Programs,* MIT Press, 1994.

McGraw, J. Skedzielewski, S., Allan, S. Oldehoeft, R., Glauert, J., Kirkham, C., Noyce, B. and Thomas, T. (1985) SISAL: Streams and Iterations in a Single Assignment Language: Reference Manual Version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

Skedzielewski, S. and Glauert, J. (1985) IF1 - An intermediate form for applicative languages, Version 1.0, Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.