Caching for Parallel Linear Genetic Programming

Carlton Downey
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
Carlton.Downey@ecs.vuw.ac.nz

Mengjie Zhang
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
Mengjie.Zhang@ecs.vuw.ac.nz

ABSTRACT

Parallel Linear Genetic Programming (PLGP) is an exciting new approach to Linear Genetic Programming (LGP) which decreases building block disruption and significantly improves performance by the introduction of a parallel architecture. We introduce a caching algorithm for PLGP which exploits this parallel architecture to avoid the majority of instruction executions. This allows PLGP programs to be executed an order of magnitude faster than LGP programs with an equal number of instructions.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Design

Keywords

Genetic Programming, Linear Genetic Programming, Caching

1. INTRODUCTION

Population based search algorithms such as Genetic Programming (GP) are hamstrung by large run times. While there has been significant improvement in the performance of GP algorithms, little progress has been made in reducing GP program execution times. In fact many GP algorithm improvements actually increase algorithm run times [3], somewhat offsetting the advertised performance benefits.

Caching is one technique commonly used to decrease algorithm run time. Unfortunately conventional forms of GP are not well suited to caching. The execution of GP program components is highly contextual: Two identical instruction subsequences can have completely different executions due to different positioning within their respective GP programs.

Parallel LGP (PLGP) is an exciting new form of GP introduced by Downey and Zhang in 2011 [2]. The key concept of PLGP is the adoption of a parallel program architecture, where each PLGP program consists of several independently executed *factors*. Thus PLGP lacks the contextual information which prevents effective caching in LGP, presenting opportunities for caching not available in a conventional LGP system.

GECCO'11, July 12–16, 2011, Dublin, Ireland.
ACM 978-1-4503-0557-0/11/07.

2. BACKGROUND

In LGP the individuals in the population are programs in some imperative programming language. Each program consists of a number of lines of code, to be executed in sequence. The LGP used in this paper follows the ideas of register machine LGP [1]. In register machine LGP each individual program is represented by a sequence of register machine instructions, typically expressed in human-readable form as C-style code.

PLGP is an extension of LGP where each program consists of n dissociated factors. Each factor is a short sequence of instructions identical to a conventional LGP program. To execute a PLGP program we execute all of its factors in parallel to produce n result vectors. These result vectors are then summed to produce the program output.

3. CACHING ALGORITHM

The output of any PLGP program in generation $n+1$ is identical to its output in generation n, with the exception of a single factor. When program evolution occurs, precisely one factor is modified. All program factors which were not modified during evolution will produce identical output to the previous generation.

We can calculate program output at generation $n+1$ based on program output at generation n. Let V_i^m be the output of the i'th *factor* at the m'th generation and let S^m be the *program output* at the m'th generation. Then:

$$S^m = \sum_{i=1}^{n} V_i^m = \sum_{i=1}^{n-1} V_i^m + V_n^m \quad (1)$$

Without loss of generality let V_n be the factor that is modified in generation $m+1$. Then we know factors $V_1, ..., V_{n-1}$ will evaluate to the same vectors in generation $m+1$ as they did in generation m. Hence:

$$\begin{aligned} S^{m+1} &= \sum_{i=1}^{n-1} V_i^m + V_n^{m+1} \\ S^{m+1} &= S^m - V_n^m + V_n^{m+1} \end{aligned} \quad (2)$$

In short, we can calculate program output in generation $n+1$ by subtracting off the old result vector, and adding the

new result vector. We refer to this technique as difference caching.

To execute a PGLP program on a single training example we require knowledge of three vectors: program output at generation n, the output of factor f_i at generation n, and the output of factor f_i at generation $n+1$. The output of factor f_i at generation $n+1$ is calculated directly, and the other two vectors are kept in the cache.

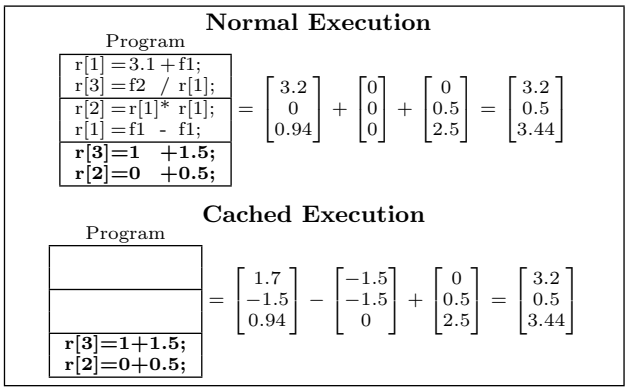

Figure 1: Contrasting normal PLGP program execution to cached PLGP program execution. Assume that during the most recent iteration factor three was selected for evolution.

4. THEORETICAL ANALYSIS

Executing a PLGP program without caching causes every instruction in every factor to be executed. Therefore if each program has m factors of length n, then execution without caching requires $O(nm)$ instruction executions. Executing a PLGP program with difference caching requires one factor to be executed. Therefore program execution with caching requires n instruction executions. Hence *difference caching will save $mn-n$ instruction executions*, allowing PLGP programs to be execute m times faster than LGP programs.

The most important benefit of difference caching is that the cost of caching is *independent* of the number of program factors. Hence programs with a greater number of factors will receive a greater benefit from caching. This is particularly important because it has been shown that PLGP programs give optimal performance when they have a large number of factors.

5. RESULTS

Fig. 2 shows that LGP programs execute more rapidly than PLGP programs of equivalent size, regardless of the number of factors. Furthermore PLGP program execution time is proportional to the number of factors: Programs with more factors execute more slowly.

Fig. 3 shows that cached PLGP programs can be executed far more rapidly LGP programs of equivalent size. Furthermore the execution time of cached PLGP programs is directly related to the number of program factors: Programs with more factors execute more rapidly, while programs with fewer factors execute more slowly.

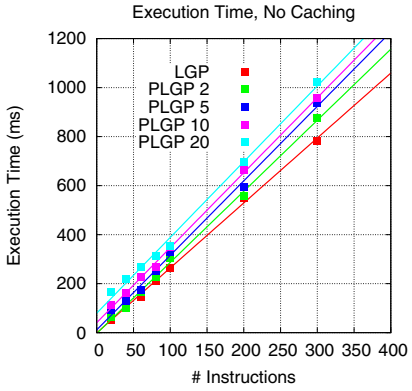

Figure 2: LGP vs. PLGP for various program lengths

Figure 3: LGP vs. PLGP with difference caching for various program lengths

6. CONCLUSIONS

By exploiting the parallel architecture of PLGP programs it is possible to decrease program execution times by more than an order of magnitude. PLGP programs are composed of n independent factors. PLGP program execution consists of executing these factors and summing the result vectors. By caching program output and factor output from generation n it is possible to calculate program output in generation $n+1$ as the sum of three vectors. This allows us to avoid executing the majority of the program factors and hence allows PLGP programs to be executed an order of magnitude more rapidly than LGP programs with an equal number of instructions.

7. REFERENCES

[1] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*. Morgan Kaufmann Publishers, 1998.

[2] C. Downey and M. Zhang. Parallel linear genetic programming. In *EuroGP*, volume 6621, pages 178–189, 2011.

[3] B. E. Eskridge and D. F. Hougen. Memetic crossover for genetic programming: Evolution through imitation. In *Genetic and Evolutionary Computation – GECCO, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 459–470. Springer-Verlag, 2004.