Using Parallel Genetic Programming to Evolve Compression Preprocessors

Johan J.M.G. Parent
Vrije Universiteit Brussel
Brussels, 1150, Belgium,
parentjo@vub.ac.be

August 10, 2001

Abstract

We present a new approach for applying genetic programming to lossless data compression. Unlike programmatic compression the evolved programs are preprocessors. The fitness function is based on the entropy concept, thereby making it independent of the type information being processed. The obtained results are encouraging in that significant improvements can be achieved. Furthermore the computation time required is much smaller than in the case of programmatic compression. This combined with the use of parallel genetic programming kernel makes this a viable approach. We used a strongly typed GP kernel. The parallel kernel runs in parallel using the island model.

1 Introduction

1.1 Genetic programming

John Koza [8] has introduced genetic programming which is an extension of the ideas of John Holland. In genetic programming the bit strings are replaced by parse trees. These trees represent expressions consisting of functions and terminals defined by the user. The trees are then evaluated and subjected to genetic operations thereby introducing potentially better solutions.

1.2 Parallel genetic programming

Parallel genetic programming is not a new research domain. Genetic algorithms and genetic programming are well suited for parallelization when using the so called island model. In this model several populations (demes) evolve simultaneously and periodically exchange individuals. The easiest parallelization distributes the populations onto different computing nodes. Thereby allowing the evolution of the populations to proceed in parallel. The infrequent communication between populations and the usually computationally expensive evaluation of individuals, makes parallel genetic programming an example of coarse grain parallelism.

The island model however introduces new parameters in the genetic programming scheme. These parameters are the exchange frequency, number of individual exchanged, number of populations and the exchange topologies. The implications of this model have been studied by many researcher : Punch[14], Koza and Andre [1], ...

2 Parallel kernel

The package that has been used for this experiment, is a modified version of the Lil-gp [13] that can run in parallel. The software is freely available on-line under the terms of the GPL (see [12]).

The parallel kernel is based on PVM message passing library [10]. The implementation is fully distributed as far as the genetic operations are concerned. Meaning that the evaluation of the individuals and the genetic operations are done in parallel on the different computing nodes that host the populations. A client-server model is used as well in order to collect statistics about the experiments and to configure the computing nodes. The parallel kernel transparently handles several tasks:

- the distribution of populations onto PVM nodes
- configuring the computing nodes
- collection of statistics
- exchange of individuals between populations
- peer-to-peer communication for the exchanges

The kernel supports both synchronous and asynchronous exchanges of individuals. In synchronous mode at any given point in time all the computing nodes evolve the same generation. Using barier synchronization the kernel ensures that faster nodes effectively *wait* for slower nodes before starting to evolve the next generation. This is not the case in asynchronous mode. In asynchronous mode individuals are exported but the nodes do not wait if there are no new individuals that need to be imported. Furthermore when individuals are imported, the generation of the individual needs not to match the generation being evolved at that moment on that node.

In order to match the granularity of the parallel hardware the kernel is capable of grouping several populations on one computing node. In that case, the kernel distinguishes between internal and external interpopulation exchanges to minimize the use of the message passing library.

The main advantage of the parallel kernel is the transparant handling of all the parallel operations. Yet the computational speedups one achieves using the island model are significant. Thanks to the design of the kernel the user needs not to worry about this during the implementation of the problem!

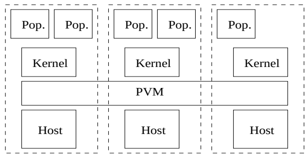

Figure 1: The parallel kernel completely shields the user from the parallel operations. Handling the population mapping, exchange of individuals and the statistics.

3 EA's and compression

Evolutionary algorithms have been used in the past for compression purposes. Two approaches can be distinguised. In a first group, genetic algorithms were used to find parameters for a compression algorithm in order to maximize compression Driesen[4]. Feiel and Ramakrishnan [6] have used genetic algorithms to optimize the compression of color images using vector quantization. Other approaches used genetic programming for what is called *programmatic compression*. De Falco and al.[5] have used genetic programming for string compression. Fukunaga and Stechert[7] have used genetic programming for lossless compression of grayscale images. Nordin and Banzhaf[11] achieved lossy programmatic compression of images and sound. Noteworthy is the fact that [11][7] both used a genocompiler for their experiments. This software eliminates the function call overhead incurred by other systems during the evaluation of the individuals. Luke[9] reports a relative improvement in speed up of 2000 times compared to LISP code and of 100 times compared to interpreted C (the latter has been used for this experiment).[1]

3.1 Preprocessing

Our attention here goes toward lossless compression algorithms. Examples of popular lossless compression programs are gzip and Winzip [3]. One of the more recent contributions in generic lossless compression is the Burrows-Wheeler transform (1993) [2].

[1] Notwithstanding the increase in computation speed, [7][11] report runs lasting several tens of hours on powerful workstations.

Instead of aiming for a program that recodes the data, we seek for a transformation. We investigate whether a program can be evolved to transform given data in order to enhance its compressibility. The exact transformation is not explicitly known but one can formulate certain conditions it should possess.

Such a preprocessing program could be formalized as a function P that works on a string S over an alphabet A. The length of a string S is denoted as $|S|$. C is represents a data compression algorithm. The result we are looking for is a transformation P, such that the following condition holds:

$$S' = P(S) \tag{1}$$
$$|C(S')| < |C(S)| \tag{2}$$

Stating the problem in these terms makes it easy to fold it into the genetic programming framework since both the program we are looking for, P here, and the fitness function are easily separated.

The conditions has two serious disadvantages though. First, computing the result of Eq. (2) is computationally expensive. Second, the transformation depends on the compression algorithm used.

To avoid this problem we reformulated it using a metric from information theory, the entropy.

The entropy gives the average *information content* of a symbol in a given message.[2] The formula for the entropy is given below, note that P_i represent the probability of symbol i in the message[3].

$$H = -\sum P_i . \log P_i \tag{3}$$

Using the entropy (Eq (3)) we have a means to determine how much information is actually present in a given message. One can use the entropy as an objective criterion for the transformation that we seek to evolve. The purpose of this transformation is : lowering the entropy of a message (data). By lowering the entropy we reduce the information content.

[2]Entropy has a much more rigorous mathematical foundation but the description given here suffices for the purpose of this text.

[3]The model used here is a first order model (marginal probability). Higher order models are based on conditional probabilities.

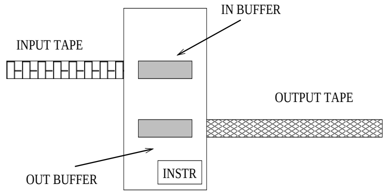

Figure 2: A simple virtual machine has been defined which bears some resemblance with an automaton.

As a result it is possible to recode the message even more shortly (this is the domain of compression algorithms. . .).

The terminal and function set provided to the genetic programming software is designed to reduce the entropy, albeit under the good circumstances. It is up to the evolutionary pressure to bring forth those combinations of transformations that will reduce the entropy the most. Using the entropy we now can define a new condition for the transformation we wish to evolve:

$$\frac{H_{out} \times L_{out}}{H_{in} \times L_{in}} < 1 \tag{4}$$

Eq. (4) provides us with a computationally cheaper fitness function and is independent of any data compression algorithm. The formula requires the total information after transformation to be lower than before transformation. Note that we impose no limit on the length of the transformed data. Since we do not immediately compress the data, L_{out} can either be greater or equal to L_{in}.

4 Setup

The setup used for this experiments will now be presented. A simple virtual machine has been defined which bears some resemblance with an automaton since it uses an input and an output tape. Data can be read from the input and new data can be put on the output (see figure 2).

(LOAD *int data***)** ⇒ *tape*	load data
(FWD *data***)** ⇒ *tape*	reload data
(REW *data***)** ⇒ *tape*	load output as data
(SEQ *tape tape***)** ⇒ *tape*	make sequences
(DIV *int num***)** ⇒ *int*	protected division
(DPCM *num***)** ⇒ *data*	difference code
(MIN *num***)** ⇒ *data*	average
(RAW) ⇒ *data*	raw copy
(INV *num***)** ⇒ *data*	inversion
(PEC) ⇒ *data*	pseudo exponent
(MTF) ⇒ *data*	move to front
(SUB *num***)** ⇒ *data*	substitution
(ERC_int) ⇒ *int*	random constant
(END) ⇒ *int*	length input
(ERC_num) ⇒ *num*	random constant

Table 1: Short description of the different functions and terminals as well as their associated type(s).

4.1 Strongly typed

The approach presented here uses a strongly typed genetic programming kernel written in C that produces LISP-like programs. The use of a typed genetic programming kernel is considered a must here since it allows to structure the programs, thus speeding up the search process. The different types and the instruction and terminal set used for this experiment will be presented shortly.

The instruction set can be divided into two categories. Instructions that control the input tape and instructions that process the data read from tape. Typing is used to structure the programs that can be evolved. The first two types are integer types, ***int*** and ***num***. The ***num*** type represents small integers in the range [0,20]. In order to integrate the instruction of the machine two types were defined for the functions : ***tape*** and ***data***.

4.2 Settings

The experiments presented here were done using 2 populations of 500 individuals. The selection probabilities of the different genetic operations were: crossover 75%, mutation 20% and reproduction 5%. The subpopulations exchanged 3 individuals every 10 generations. The exchanged individuals (the ones being exported) were selected using tournament selection. The worst individuals in the destination population were replaced by the imported ones. The exchange topology was a ring.

4.3 Platform

In this experiment our attention went toward the feasibility of this new preprocessing oriented approach. The hardware used for this experiment a 8 Pentium II 350 Mhz cluster running Linux, the network used a 3Com 100Mbit/s non-blocking switch.

4.4 Input data

During these experiments the Canterbury Corpus has been used as well as various files ranging from bitmaps to word processor files. Meaning that the size of the input tape usually exceeded the order of several kilobytes and went even up to more than 1 megabyte in size. Some interesting results were observed.

5 Results

To validate the results a set of software tools has been implemented. Meaning that next to the functions and terminals needed by the gp software, an encoding format has been designed and a decoder has been implemented. One can thus decode the preprocessed files and compare them with the original data with a program like *diff*.

The obtained reduction of the entropy is given in table 2. To demonstrate the usefulness of this work the preprocessed files have been compressed (see table 3).

5.1 Speedup

In order to compare the computational speedup the parallel and sequential runs have been timed. In figure 4 one can see that the computational speedup is linear (the parallel run time is half the sequentail

Filename	original H (bits)	new H (bits)
kennedy.xls	3.57	0.77
laptop.bmp	7.76	3.37
lena_std.ppm	7.75	5.29
mosaic.pnm	7.78	4.15
peppers.ppm	7.66	5.35

Table 2: Comparison between the initial entropy and the entropy after preprocessing.

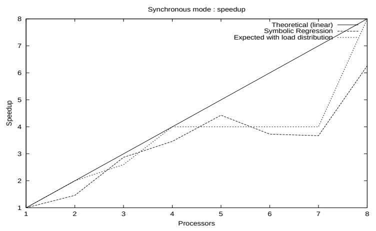

Figure 3: Speedup achieved on the standard symbolic regression problem in synchronous mode. The graph also shows the ideal linear speedup as well as the maximum speedup based on the population distribution.

time). This clearly illustrates the efficiency of the parallel kernel. One should keep in mind the fact that the user doesn't write a single line of *parallel code*! The solutions found in parallel and sequential mode are equivalent. This means that the search space exploration was the same.

5.2 Filters

Genetic programs tend to improve more slowly as their size increases. This is due to the increase of points candidate for genetic change. In their experiments Nordin and Banzhaf[11] also had problems evolving programs that *cover* all the data. Therefore they have chosen to evolve separate programs for segments of fixed size, "chunking". This approach was not chosen although large amounts of data were used. The solution that emerged here was filtering.

With the introduction of the special **END** terminal in the set this unexpected result showed up. The entire data undergoes several transformation before

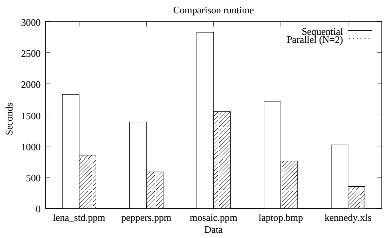

Figure 4: A comparison between the sequential and parallel execution time (2 nodes). 8 populations of 1000 individuals over 50 generations.

being sent to the output. This filtering solution circumvents the *data coverage* problem that was initially encountered.

5.3 Introns

Furthermore during the experiments it was observed that by defining a more flexible virtual machine which can cope with introns, the obtained programs were much better. The same observation was made by [9] but from another point of view. The introns combined with a relatively high mutation rate offer more opportunity for useful changes (i.e. new stage in the filters) to occur.

6 Conclusion

This experiment presents a new approach for combining genetic programming and lossless data compression. The chosen approach develops preprocessing programs which are tailored to the data one wishes to compress. The obtained results are encouraging both in term of gain in compression as for the computation time required to evolve the programs. Surprisingly the evolved programs were mostly filters although the provided functions and terminals allowed for more complex preprocessors to evolve.

7 Future work

When exchanges between populations are rather infrequent the load balancing seems to deteriorate.

Filename	Bytes	bzip2 -9 (GP)	bzip2 -9
kennedy.gp	1029744	24991	130280
laptop.gp	304182	131144	162270
lena_std.gp	786466	538238	584538
mosaic.gp	910534	471795	658750
peppers.gp	786447	544211	635479

Table 3: Difference in compression rate after preprocessing.

This occurs when the size of the programs being evolved by the different populations differs a lot. Further investigation is required to find out which parameters of the island model play a significant role for balancing the computational load.

References

[1] David Andre and John Koza. Parallel genetic programming: A scalabel implementation using the transputer network architecture. In Peter J. Angeline and Jr. Kenneth E. Kinnear, editors, *Advances in Genetic Programming 2*. The Mit Press.

[2] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. *Digital System Research Center research report 124*, 1994.

[3] Winzip Corporation. www.winzip.com.

[4] Karel Driesen. Compressing sparse tables using a genetic algorithm. In *In Proceedings of the GRONICS'94 Student Conference, Groningen*, February 1994.

[5] I. De Falco et al. A kolmogorov complexity-base genetic programming tool for string compression. In *Proceedings Seventh International Conference on Genetic Algorithms (ICGA97)*, 1997.

[6] H. Feiel and S. Ramakrishnan. A genetic approach to color image compression. In *ACM 0-89791-850-9*, 1997.

[7] Alex Fukunaga and Andre Stechert. Evolving nonlinear models for lossless image compression with genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 92–102, 1998.

[8] John R. Koza. *Genetic Programming II*. The MIT Press, 1994.

[9] Sean Luke. Code growth is not caused by introns. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Conference*, 2000.

[10] Parallel Virtual Machine. http://www.netlib.org/pvm3/.

[11] Peter Nordin and Wolfgang Banzhaf. Programmatic compression of images and sound. In David B. Fogel John R. Koza, Davis E. Goldberg and Satnford University Rick L. Riolo ed., editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 354–350. CA, USA, Mit Press.

[12] J. Parent. Parallel lil-gp technical report. Vrije Universiteit Brussel, http://parallel.vub.ac.be./~johan/Projects/.

[13] B. Punch and D. Zonker. lil-gp genetic programming version 1.1 beta. michigan state university, http://garage.cps.msu.edu/software/lil-gp/index.html.

[14] William F. Punch. How effective are multiple populations in genetic programming. In K. Chellapilla K. Deb M. Dorigo D.B. Fogela M. Garzon D. Goldberg H. Iba J.R. Koza, W. Banzhaf and R.L. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual conference*. Morgan Kaufmann.