Sub-Machine-Code GP: New Results and Extensions

Riccardo Poli

School of Computer Science
The University of Birmingham
Edgbaston
Birmingham, B15 2TT, UK
R.Poli@cs.bham.ac.uk
Phone: +44-121-414-3739

Abstract. Sub-machine-code GP (SMCGP) is a technique to speed up genetic programming (GP) and to extend its scope based on the idea of exploiting the internal parallelism of sequential CPUs. In previous work [20] we have shown examples of applications of this technique to the evolution of parallel programs and to the parallel evaluation of 32 or 64 fitness cases per program execution in Boolean classification problems. After recalling the basic features of SMCGP, in this paper we first apply this technique to the problem of evolving parallel binary multipliers. Then we describe how SMCGP can be extended to process multiple fitness cases per program execution in continuous symbolic regression problems where inputs and outputs are real-valued numbers, reporting experimental results on a quartic polynomial approximation task.

1 Introduction

Genetic Programming (GP) [8, 9, 2] is usually quite demanding from the computation load and memory use point of view. So, over the years a number of ideas on how to improve GP performance have been proposed in the literature.

For example, Singleton [22] proposed a GP implementation in C++ which was several times faster than equivalent Lisp implementations. Handley [6] proposed storing a population of trees as a single directed acyclic graph obtaining considerable savings of memory and computation. Nordin proposed evolving programs in machine code form [13, 16, 14, 15], a technique which is claimed to be at least one order of magnitude faster than GP system based on higher-level languages. Other researchers (see for example [4]) have proposed to compile at runtime standard GP trees into machine code before evaluation. Speed up strategies based on intelligently reducing the number of fitness cases have been proposed [5, 24, 11]. Finally, some research has been devoted to parallel and distributed implementations of GP (see for example [1, 23, 7]).

Some of these techniques are now used in many GP implementations. This and the increased power of modern workstations make it possible run 50 generations of a typical GP benchmark problem with a population of 500 individuals in perhaps ten seconds on a normal workstation. Nonetheless, the demand for more and more efficient implementations has not stopped.

Most computer users consider their machines as sequential computers. However, at a lower level of abstraction CPUs are really made up of parallel components. In this sense CPUs can be seen as Single Instruction Multiple Data (SIMD) processors. *Sub-machine-code GP* (SMCGP) is a method to exploit this form of parallelism to improve the efficiency and the range of applications of genetic programming [20]. SMCGP extends the scope of GP to the evolution of *parallel programs running on sequential computers*. These programs are faster since, thanks to the parallelism of the CPU, they perform multiple calculations in parallel during a single program evaluation. SMCGP is more efficient since, in some domains, it allows the evaluation of multiple fitness cases per program execution. In [20] we showed that on Boolean classification problems nearly 2 orders of magnitude speedups can be achieved. In this paper we apply SMCGP to the problem of evolving parallel binary multipliers and extend SMCGP to process multiple fitness cases per program execution in continuous symbolic regression problems where inputs and outputs are real-valued numbers.

The paper is organised as follows. In Section 2 we describe the basic principles behind SMCGP and we provide examples on how the technique can be used to evaluate multiple fitness cases per program execution in Boolean classification problems and to evolve parallel programs which exploit the CPU's internal parallelism. In Section 3 we describe a new application of SMCGP to the evolution of parallel 2-bit multipliers. In Section 4, we describe how sub-machine-code GP can be extended to allow the parallel evaluation of multiple fitness cases in continuous symbolic regression problems. In the same section we report the results of the application of this SMCGP extension to a quartic polynomial approximation task. We discuss the advantages and disadvantage of SMCGP and draw some conclusions in Section 5.

2 Sub-machine-code GP

In a CPU some instructions are performed in parallel and independently for all the bits in the operands. For example, if the operands are two words containing the integers 193 (binary 00000000000000000000000011000001, assuming a 32-bit CPU) and 252 (binary 00000000000000000000000011111100), a bitwise AND operation on a modern CPU will produce a word containing the integer 192 (binary 00000000000000000000000011000000) in a single clock tick. The CPU will perform the operation by concurrently activating 32 different AND gates within as many slices of the arithmetic logic unit. So, logically in this operation the CPU can be seen as a SIMD processor made up of 32 1-bit processors. In other instructions the CPU's 1-bit processors can be imagined to interact through communication channels; in shift operation, for example, where each processor sends data to one of its neighbours. Some operations in the CPU are executed simultaneously by all 1-bit processors. Those requiring interaction between the 1-bit processors may require multiple clock ticks. Nonetheless, as far as the user is concerned the CPU's 1-bit processors run in parallel, since the results of the operation of all processors become available at the same time.

All this powerful parallelism inside our CPUs has been ignored by most of the GP community so far, perhaps because many of us are not used to think in term of bits, nibbles, carries, registers, etc. The most notable exception to this is the work of Peter Nordin [13, 16, 14, 15] which exploits the CPU is its entirety in some problems (for example, in [13] complete Swedish words were coded into 32 bit integers which were processed in parallel for classification purposes).

Sub-machine-code GP exploits the CPU's internal parallelism to obtain faster and more expressive forms of GP. This is obtained as follows:

- The function set includes operations which exploit the parallelism of the CPU, e.g. bitwise Boolean operations, shift operations, etc.
- The terminal set includes integer input variables and constants. These are interpreted as bit vectors where each bit represents the input to a different 1-bit processor. For example, the integer constant 192 (binary 00000000000000000000000011000000) is seen as a 1 by the 1-bit processors acting on bits 7 and 8. It is seen as a 0 by all other 1-bit processors.
- The integer result produced by the evaluation of a program is interpreted as a bit vector. Each bit of this vector represents the result of a different 1-bit processor.

Since most programming languages include operations which use directly the corresponding machine code operations, it is possible to exploit the parallel nature of the CPU in most languages. This means that any GP system can potentially be used to do sub-machine code GP.

An ideal application for SMCGP is to evaluate multiple fitness cases in parallel. Boolean induction problems lend themselves naturally to this use of sub-machine-code GP, leading to 1.5 to 1.8 orders of magnitude speedups (for 32 and 64 bit machines, respectively). We explored this idea in detail in [20]. In the following subsection we recall the main points.

2.1 Simultaneous Evaluation of Multiple Boolean Fitness Cases

In Boolean classification problems sub-machine-code GP can be used to evaluate multiple fitness cases in parallel. The approach is as follows: a) bitwise Boolean functions are used, b) before each program execution the input variables need to initialised so as to pass a *different fitness case* to each of the different 1-bit processors of the CPU, c) the output integers produced by a program are converted into binary form and each of their bits is interpreted as the output for a different fitness case.

For example, let us assume that we want to evolve a program implementing the XOR function. The truth table of this function is:

x1 x2	x1 XOR x2
0 0	0
1 0	1
0 1	1
1 1	0

Let us further assume that we are using a function set including bitwise AND, OR, and NOT, a terminal set {x1,x2}, and a fitness function which measures the number of bits in the rightmost column of the truth table correctly classified by each program. Then we can evaluate the fitness of a program in one program execution by loading the values 5 (binary 0101) into x1 and 3 (binary 0011) into x2 and then executing the program. The values assigned to the terminals are integers obtained by converting into decimal the corresponding bit columns of the truth table. The value returned by the program represents the program's attempt to fit the rightmost bit column in the truth table. So, by measuring the Hamming distance between the two, we obtain a program's fitness. For example, if we run the program `(AND (NOT x1) (OR x1 x2))`, it first performs the bitwise negation of x1, obtaining 10 (binary 1010), then it computes the bitwise disjunction of x1 and x2, obtaining 7 (binary 0111). Finally the two results are conjoined obtaining 3 (binary 0010). If we compare (bitwise) this number with the desired output, 6, representing the binary number in the rightmost column in the truth table (0110), the comparison indicates that the program `(AND (NOT x1) (OR x1 x2))` has a fitness of 3. This is obtained with only one execution of the program.

In general using SMCGP all the fitness cases associated to the problem of inducing a Boolean function of n arguments can be evaluated with a *single program execution* for $n \leq 5$ on 32 bit machines, and $n \leq 6$ on 64 bit machines. For bigger values of n the 2^n fitness cases can be evaluated in blocks of 32 or 64 [20, appendix]. Since this can be done with almost any programming language, this technique can lead to 1.5 to 1.8 orders of magnitude speedups.

Because of the overheads associated to the packing of the bits to be assigned to the input variables and the unpacking of the result the speedup factors achieved in practice are to be expected to be slightly lower than 32 or 64. However, these overheads can be very small. Indeed in [20] we reported speedups of 31 times for a 32-bit machine. More recently running a C SMCGP implementation on a 64-bit 400MHz machine we have measured an average evaluation time per fitness case of $1.37\mu s$, corresponding to 1.3 primitives per CPU cycle!

2.2 Evolution of Parallel Programs for the CPU

The CPU 1-bit processors must always execute the same program. Nonetheless it is still possible for them to perform different computations. This can be achieved by passing constants with different bits set to different processors.

For example, let us consider a 2-bit CPU which is computing a bitwise AND between a variable `x` and the constant 2 (binary `10`), and suppose that `x` is either 0 (binary 00) or 3 (binary 11). In these conditions, the first 1-bit processor will perform `(AND x 1)` and will therefore return a copy the first bit in `x`, i.e. it will compute the identity function. The second 1-bit processor will instead compute `(AND x 0)` which is always false, so it will return the constant 0 whatever the value of the second bit in `x`.

This idea can be exploited to evolve parallel programs for the CPU using SMCGP. To do so it is sufficient to: a) use bitwise operations in the function set, b) use a range of constants in the terminal set which can excite differently different 1-bit processors in the CPU, c) use variables in the terminal set whose bits are either *all* 0 or *all* 1, d) unpack the output. Note that this approach is the exact dual of the one used in the parallel evaluation of fitness cases described in the previous subsection. Here we use different bit configurations for the constants and identical bit configurations for the variables (i.e. variables can only take the binary values 000....0 or 111....1). There we did the opposite.

In [20] we applied sub-machine-code GP to the problems of evolving parallel programs implementing 1-bit and 2-bit adders with and without carry. We also used SMCGP to evolve parallel character recognition programs. In the next section we will demonstrate the idea on another problem: the evolution of parallel 2-bit multipliers.

3 Evolution of Parallel 2-bit Multipliers

A 2-bit multiplier has four inputs: `x1` and `x2` which represent the least significant bit and the most significant bit of the first operand, respectively, and `x3` and `x4` which represent the second operand. The multiplier has four outputs, `r1`, `r2`, `r3` and `r4`, which represent the result of the multiplication. The truth table for a 2-bit multiplier is the following:

x1	x2	x3	x4	r1	r2	r3	r4
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
1	0	1	0	1	0	0	0
0	1	1	0	0	1	0	0
1	1	1	0	1	1	0	0
0	0	0	1	0	0	0	0
1	0	0	1	0	1	0	0
0	1	0	1	0	0	1	0
1	1	0	1	0	1	1	0
0	0	1	1	0	0	0	0
1	0	1	1	1	1	0	0
0	1	1	1	0	1	1	0
1	1	1	1	1	0	0	1

By exploiting the parallelism in the CPU it is possible to compute `r1`, `r2`, `r3` and `r4` simultaneously. This can be done by slicing the output of a *single* program into its component bits and interpreting the first four of them as `r1`, `r2`, `r3` and `r4`. Of course to do this it is necessary to include in the program appropriate constants which can excite differently different parallel components of the CPU.

Solving the multiplier problem by hand would already be difficult if one could use four separate program trees to compute `r1`, `r2`, `r3` and `r4`. However, designing a parallel program to do that would be really challenging. Fortunately, this can easily be done by evolution using sub-machine-code GP.

To do that we used the function set {`AND, OR, NOT, XOR`}, where all functions were implemented using bitwise operations on integers. This problem has four outputs and so requires using four 1-bit processors within the CPU. The terminal set was {`x1, x2, x3, x4, R`}, where `x1` and `x2` represent the first operand, and `x3` and `x4` represent the second operand. The terminal `R` is a random constant generator which creates integer constants in the set {0,1,...,15}. These constants allow GP to modify selectively the behaviour of different parts of the program for different 1-bit processors. The variables `x1`, `x2`, `x3` and `x4` took the values 0 (binary 0000) and 15 (binary 1111) depending on whether the corresponding bits of the 2-bit multiplier truth table were 0 or 1. This was done to provide the same inputs to the first four 1-bit processors of the CPU. The fitness function was the sum (over all the 16 possible combinations of inputs) of the normalised weighted sum (over all the 4 outputs) of the absolute difference between the actual output and the desired output, i.e.

$$f = \sum_{fc=1}^{16} \sum_{i=1}^{4} \frac{2^{(i-1)}}{15} \cdot |\mathtt{y}i - \mathtt{r}i|$$

where `y1`, `y2`, `y3` and `y4` are the first four bits of the integer produced by the evaluation of a program and `r1`, `r2`, `r3` and `r4` are the desired outputs (for fitness case fc). To promote parsimonious solutions this fitness was decreased by 0.001 times the number of nodes in a program. Except for the fitness function, no change of our standard-GP system was necessary to run sub-machine-code GP.

In our runs we used a population of 1000 individuals, 500 generations, standard subtree crossover (with uniform probability of selection of the crossover points) with crossover probability 0.9, point mutation [18, 19] with a probability of 0.03 per node, and tournament selection with tournament size 7. The random initialisation of the population was performed so as to obtain a uniform distribution of program sizes between 1 and 40 nodes. Runs were not stopped when a solution was found, allowing GP to discover additional, more parsimonious solutions.

We performed 50 independent runs with these parameters. Of these, 32 were successful. The computational effort to obtain a solution with 99% probability in repeated runs was 2,088,000 fitness evaluations. The average generation at which the first solutions were found was 275. The average structural complexity of the first solutions found was 83.7 nodes. However, by generation 500 the average size of the solutions was 36.0, with structural complexities ranging between 21 and 55 nodes.

For example, in one run a solution was found by GP at generation 298. The solution included 105 nodes. By allowing the run to continue, by generation 477 GP had simplified this solutions down to 32 nodes. The simplified solution is shown in Figure 1. As far as the bit-1 processor of the CPU (the one which com-

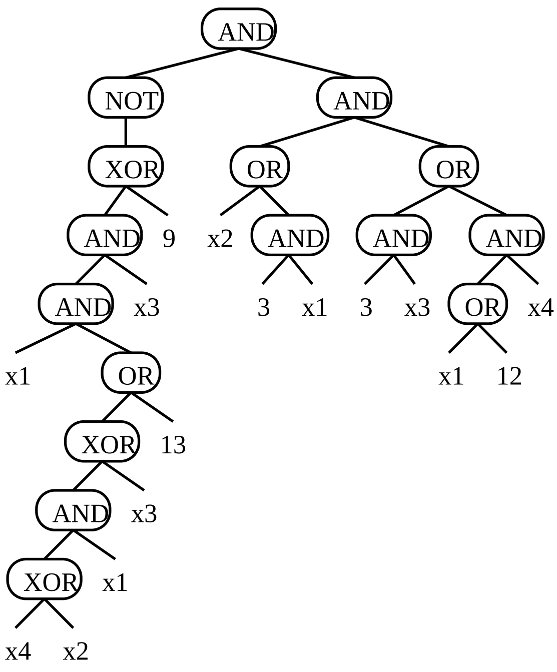

Fig. 1. A program evolved using sub-machine-code GP to solve the 2-bit multiplier problem.

putes r1) is concerned this program is equivalent to the one shown if Figure 2(a). This version of the program has been obtained by replacing all the constants in Figure 1 with bit 1 in the corresponding binary representation (e.g. 13 (binary 1101) is replaced with 1, 12 (binary 1100) with 0), and simplifying the resulting program tree. The programs "seen" by the other three 1-bit processors, shown in Figures 2(b)–(d), have been obtained in a similar fashion substituting constants with their 2nd, 3rd and 4th bits, respectively.

The two smallest solutions found by SMCGP were

```
(AND (OR x4 3) (XOR (AND (OR x2 1) (XOR 4 (AND (OR 2 x1) x3)))
                    (AND x4 (AND x1 2))))
```

and

```
(AND (XOR (AND (OR 1 x2) (XOR 4 (AND x3 (OR x1 2))))
          (AND (AND 2 x1) x4)) (OR 3 x4))
```

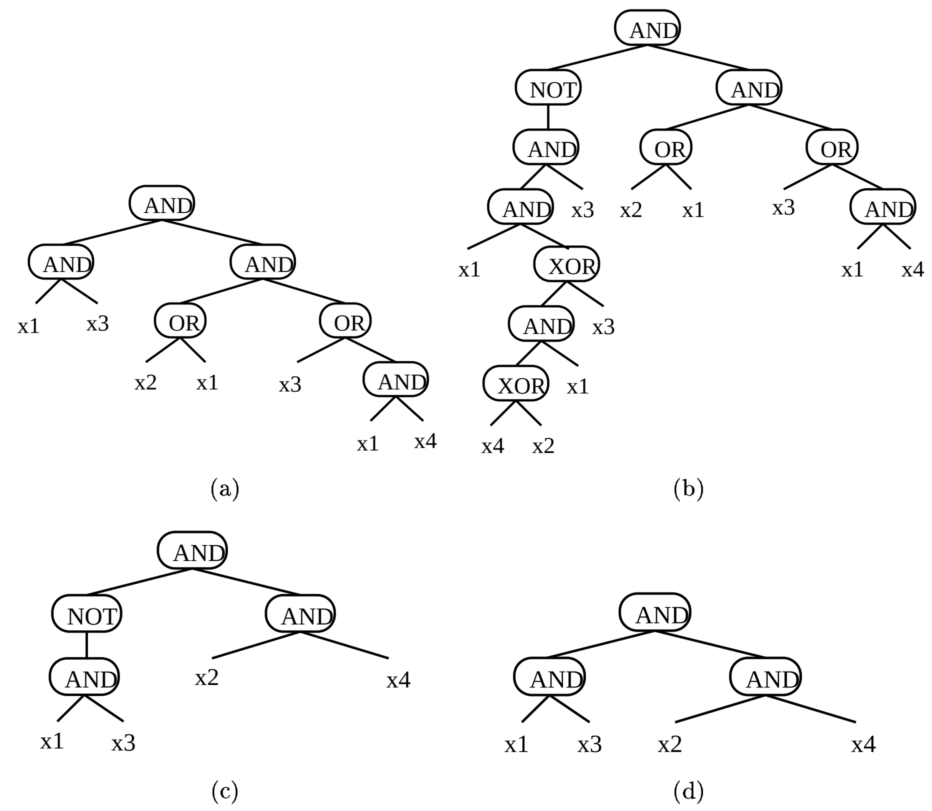

Fig. 2. Programs seen by different 1-bit processors in the CPU when executing the program in Figure 1.

These include 21 nodes, but only 10 gates each, which compares favourably with results reported in recent evolvable hardware literature [12,3].

4 Evaluation of multiple fitness cases in continuous symbolic regression problems

In Section 2 we have seen how SMCGP can be used to evaluate multiple fitness cases in parallel in Boolean classification problems, by feeding each fitness case into a different 1-bit processor of the CPU. This is possible because in such problems the input variables can only take two values, either 0 or 1, and each operation in the function set does not require more than one 1-bit processor to be performed. In continuous symbolic regression problems, where the input variables and the program output can take a wide range of values, this is certainly not the case. If, for example, we wanted to evolve a function that fits the data points (-1.0,1.0), (-0.9,0.81), ...(1.0,1.0), obtained by sampling the polynomial $y = x^2$, using a function set {*,+,/,-}, we would normally use floating point operators and floating point representations for the terminals (e.g. for the variable x and some random constants). Since such operations and representation require the use of all slices of the CPU, it seems quite difficult to use a form of SMCGP based on a floating point representation to speed up the evaluation of programs in continuous symbolic regression problems. However, this can easily be done using a different representation for reals, the fixed point representation, which allows SMCGP to parallelise the evaluation of fitness cases in continuous domains too, although to a more limited extent than in discrete domains and with some compromises.

4.1 SMCGP with fixed point representations

Fixed point representations are widely used in digital signal processing and have been used in the GA literature to represent reals within binary chromosomes (see for example [21]). The basic idea is to discretise the range in which the variables are allowed to vary and then to represent each possible discrete value with an integer, by properly offsetting, scaling and truncating/rounding such a value. For example, if a variable x is allowed to vary in the range [1,10], one could discretise such a range with a resolution 0.1 transforming it into the set $\{1.0, 1.1, 1.2 ..., 10.0\}$. Then it would be possible to represent each discrete value x_d with the integer $(x_d - 1)/0.1$. Normally the resolution is selected on the basis of the number of bits one wants to allocate to represent the integers resulting from the discretisation/rescaling process. For example, if one wants to represent the range $[a, b]$ using l-bit integers, the resolution will be set to $(b - a)/(2^l - 1)$.

In GAs fixed point numbers are manipulated directly at the binary level by crossover and mutation. So the only real-valued operation to be performed on fixed point numbers is their decoding from binary form. However, in signal processing a whole range of other operations like multiplication, addition, thresholding, etc. are normally performed. All these operations are basically defined as operations on integers, except that the results are properly scaled so as to represent the correct fixed point value. For example, if one wants to implement a multiplication between fixed point numbers in the range [0,1], stored as 8-bit integers, one can simply compute the product between the two integers and then divide the result by 256. (In reality things are slightly more complicated since overflows need to be properly handled.)

Standard GP could use fixed point numbers and operations on them. However, this would not lead to any performance advantage, since modern CPUs are able to perform floating point operations at virtually the same speed as integer ones. (This could still be a useful thing to do if one wanted to evolve digital signal processing algorithms to be later run on a different architecture.) On the contrary, a significant performance advantage can be achieved if one uses SMCGP in conjunction with a fixed point representation. This is because in most applications a relatively low resolution is sufficient. Therefore, it is possible to partition the 1-bit processors in the CPU into several groups, where each group processes a separate fitness case, in essentially the same way as in Section 2. For example, if one used 8-bit fixed point numbers on a 64 bit CPU, 8 fitness cases could be processed in parallel during each program execution.

Naturally there are limitations on what can be done with SMCGP in continuous domains. In fact, while there are operations, like addition on 32/64 bit integers, that are (nearly) decomposable into many smaller-scale (say 8-bit) operations performed in parallel, other operations, like multiplication, are not. The former kind of operations can be used freely in SMCGP with fixed point numbers, since their effect is nearly equivalent to performing several fixed-point operations sequentially and then assembling the result into a 32/64-bit word. The latter kind of operations cannot instead be used. This does not mean that one cannot use multiplications in SMCGP, but simply that multiplications will have to be performed sequentially. If other operations in the function set can be parallelised, the net effect of having some sequential operations can be relatively small.[1]

Other limitations are indirectly due to the features of fixed point representations. Fixed point representation with a small number of bits may be very efficient in terms of memory and effort to perform calculations. This is even more true in SMCGP with fixed point representations, since the number of fixed point calculations per node evaluation is inversely proportional to the number of bits used to represent fixed point numbers (when multiple calculations can be performed in parallel). However, using a small number of bits leads to inaccuracies of various kinds (limited resolution, underflow) which can only be reduced by increasing the number of bits used. So, in SMCGP with fixed point representations, there is a tradeoff between accuracy and speed.

We have explored these ideas in a set of experiments with a continuous symbolic regression problem: the problem of inducing a function that best fits a set of datapoints obtained by sampling a quartic polynomial. The experimental setup is described in the next section.

4.2 Quartic Polynomial Problem

The problem used in our experiments consisted in evolving a function from a set of (x_i, y_i) pairs obtained by sampling the polynomial $y(x) = x^4 + x^3 + x^2 + x + 6$ for $x \in [-1, 1]$. In the experiments we used 48 equally spaced fitness cases with $x_i = -1 + \frac{2}{47} \cdot i$ and $y_i = y(x_i)$ for $i = 0, ..., 47$.

We used a resolution of 8 bits to represent fixed point numbers and fixed point operations. This allowed us to divide the 64-bit CPU's 1-bit processors into 8 groups of 8, and, therefore, to process 8 fitness cases per program evaluation. The range in which the fixed point numbers were allowed to vary was

[1] Some CPUs have instructions which allow the execution of multiple integer operations on multiple data in parallel which eliminate the problem just mentioned. For example, the MMX technology of Intel CPUs allows one to partition 64-bit registers into multiple blocks of 8, 16 or 32 bits and to perform integer arithmetics on all the blocks in parallel and independently. MMX instructions can also take care of the overflows by using saturating arithmetics. However, since these features are architecture dependent, we have decided to ignore them in this work.

[0,10] which gives a resolution of approximately 0.04. The terminal set was {`X, C1, C2, C3, C4, C5, C6, C7, C8`}, where `X` is the fixed point representation of the input variable x (to which an appropriate constant was added to transform it into a positive number), and `Ci`'s are fixed point constants represented by the following binary numbers: 00000001 (`C1`), 00000010 (`C2`), 00000100 (`C3`), 00001000 (`C4`), 00010000 (`C5`), 00100000 (`C6`), 01000000 (`C7`) and 10000000 (`C8`). The function set was {`MUL, DIV, PLUS, MINUS`}, where each operation is implemented to *approximate* multiplication, division, addition and subtraction of fixed point numbers, respectively. A C implementation of these operations for a 64 bit CPU is shown in Figure 3 (see [20, appendix] for a larger fragment of SMCGP code).

In the code the function `run()` is a simple recursive interpreter capable of handling terminals and four algebraic operations. The interpreter executes the program stored in prefix notation as a vector of bytes in the global variable `program`. The interpreter returns an unsigned long integer which is used for fitness evaluation. The terminal set is stored in an array of 64-bit integers (`t[]`). The first element of the array represents the variable `X`, which is set appropriately by the fitness function before calling `run()`. The other elements of `t[]` represent the fixed point constants `Ci`'s.

The `PLUS` and `MINUS` operations are only approximately correct for two reasons. Firstly, the operations are performed without any overflow/underflow error checking, assuming arguments and results within the range [0,10]. Since the operations are performed as unsigned integer operations, this means that if one subtracts a positive number (say 5) from a slightly smaller positive number (e.g. 4.9), the result may be a large positive number (9.9) instead of the expected small negative number (-0.1). Also, if one adds two relatively large numbers, like 6 and 5, the result may be a small number (1), instead of the expected large one (11). Secondly, the operations are approximate because they are implemented so that 8 of them can be performed in parallel by different groups 1-bit processors. This requires ignoring (zeroing) the possible carries coming from other groups of 1-bit processors.

As indicated in the previous section multiplication and division operations cannot be performed in parallel by all the groups of 1-bit processors. For this reason, `MUL` and `DIV` are implemented as several fixed-point operations performed sequentially.

In the quartic polynomial problem, the interpreter (the function `run()`) is invoked by the fitness evaluator 6 times, i.e. once to evaluate fitness cases $i = 0, ...7$, once for fitness cases $i = 8, ..., 15$, etc. Before `run()` is invoked, the 8-bit fixed point representations of the x coordinates of 8 fitness cases are properly packed into a 64 bit integer which is stored in `t[0]`. Also, the program counter (`program`) is properly reset. The value returned by `run()` is decoded/unpacked into 8 8-bit fixed point numbers. The absolute differences between them and the y coordinates of the corresponding fitness cases are accumulated over the 6 calls of the interpreter and then divided by the number of fitness cases (48) to obtain the average program error. This is then subtracted from 10 (the maximum possible error) to obtain a fitness value.

In our runs we used the following parameters: population size 1,000 and 100,000 individuals, 50 generations, steady state GA, 50% of the individuals underwent subtree crossover, 50% underwent point mutation (with a mutation probability of 0.05 per node) [18, 19], tournament selection with tournament size 20, "grow" initialisation method with maximum initial depth of 3. We performed 50 independent runs with each population size. Runs were not stopped until generation 50.

During the runs whenever a new best-of-run individual was discovered, this was tested on a larger set of cases to measure its generalisation power. This was assessed by measuring the average error on 256 equally spaced values for the variable x in the interval [-1,1]. Since we use 8-bit fixed point numbers, this is the biggest set of different fitness cases that can possibly be generated.

4.3 Results

The results of the experiments with populations of 1,000 individuals were mixed. Runs were extremely fast lasting 4.1 seconds on average, but the quality of the resulting programs was relatively poor. Indeed, the mean absolute error over the 48 fitness cases of the best of run programs (averaged over 50 runs) was above 0.3. The behaviour of an average quality best-of-run program is shown in Figure 4. Nonetheless, in repeated runs it was possible to discover fairly good programs. The behaviour of best programs evolved using this population size is shown in Figure 5.

The results of the experiments with populations of 100,000 individuals instead were quite promising. The mean absolute error over the 48 fitness cases of the best of run programs (averaged over 50 runs) was below 0.1, and so was the generalisation (mean absolute error over the 256 generalisation cases) of the best of run programs. Figure 6 shows the behaviour of the program with the highest fitness discovered in the 50 runs, while Figure 7 shows the behaviour of the worst best-of-run program discovered. Considering the low resolution used to represent fixed point numbers (8 bit) this accuracy is remarkable.

The high generalisation achieved in these runs is probably a result of the relatively high number of fitness cases used, and of the implicit parsimony pressure created by point mutation with a fixed mutation rate per node [17]. Indeed the average size of the best-of-run programs was 36.6 nodes.

The 50 runs with a population size of 100,000 took a total of 27,006 seconds of CPU time to complete on a 400MHz DEC Alpha workstation. So, on average each run lasted 540 seconds, which corresponds to 10.8 seconds per generation and 108μs per individual. Since each individual was executed 6 times to assess its fitness, the average program execution time was 18 μs. During each program execution 8 fitness cases are evaluated, so the average fitness-case evaluation time is 2.25 μs.

Despite the presence of operations which could not be parallelised, SMCGP showed impressive performance in these experiments. The total number of nodes

```c
unsigned long t[9] = {      /* ----------- TERMINAL SET -------------- */
        0x0000000000000000, /* 8 different values of X are stored here */
        0x0101010101010101, /* C1 is stored here (8 times) */
        0x0202020202020202, /* C2 */
        0x0404040404040404, /* C3 */
        0x0808080808080808, /* C4 */
        0x1010101010101010, /* C5 */
        0x2020202020202020, /* C6 */
        0x4040404040404040, /* C7 */
        0x8080808080808080};/* C8 */

#define mul(bits,mask) \          /* do a*b */
    c1 = (arg1 >> bits) & 0xff; \ /* unpack fixed point number a */
    c2 = (arg2 >> bits) & 0xff; \ /* unpack fixed point number b */
    c1 *= c2; \                   /* multiply corresp. integers */
    c1 >>= SCALE; \               /* normalise */
    c1 &= 0xff; \                 /* mask overflow bits */
    arg1 = (arg1 & mask) | ( c1 << bits); /* replace a with a*b */

#define div(bits,mask) \          /* do a/b */
    c1 = (arg1 >> bits) & 0xff; \ /* unpack fixed point number a */
    c2 = (arg2 >> bits) & 0xff; \ /* unpack fixed point number b */
    c1 <<= SCALE; \               /* prescale before division */
    c1 = c2 ? c1 / c2 : c1; \     /* divide corresp. integers */
    c1 &= 0xff; \                 /* mask overflow bits */
    arg1 = (arg1 & mask) | ( c1 << bits); /* replace a with a/b */

unsigned long run() { /* Interpreter */
  register unsigned long arg1, arg2, c1, c2;
  switch ( *program++ ) {     /* ---------- FUNCTION SET ------------- */
    case PLUS : return( ( run() +  run() ) & 0xfefefefefefefefe );
    case MINUS : return( ( run() -  run() ) & 0x7f7f7f7f7f7f7f7f );
    case MUL : arg1 = run(); arg2 = run();
      mul(56,0x00ffffffffffffff);
      mul(48,0xff00ffffffffffff);
      mul(40,0xffff00ffffffffff);
      mul(32,0xffffff00ffffffff);
      mul(24,0xffffffff00ffffff);
      mul(16,0xffffffffff00ffff);
      mul( 8,0xffffffffffff00ff);
      mul( 0,0xffffffffffffff00);
      return( arg1 );
    case DIV : arg1 = run(); arg2 = run();
      div(56,0x00ffffffffffffff);
      div(48,0xff00ffffffffffff);
      div(40,0xffff00ffffffffff);
      div(32,0xffffff00ffffffff);
      div(24,0xffffffff00ffffff);
      div(16,0xffffffffff00ffff);
      div( 8,0xffffffffffff00ff);
      div( 0,0xffffffffffffff00);
      return( arg1 );
    default: return( t[*(program-1)] );
    }
}
```

Fig. 3. 64-bit C implementation of the primitive set used in the quartic polynomial problem.

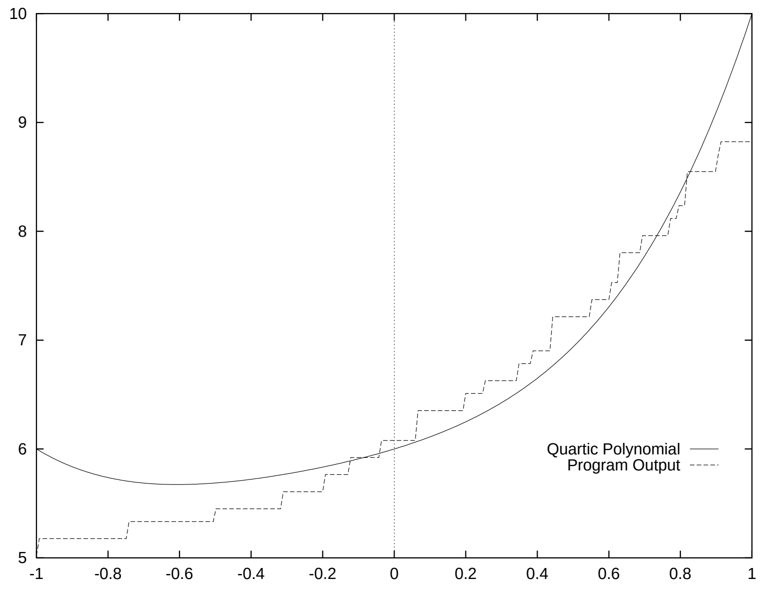

Fig. 4. Behaviour of an average quality best-of-run program evolved using sub-machine-code GP with populations of size 1,000 to solve the quartic polynomial problem.

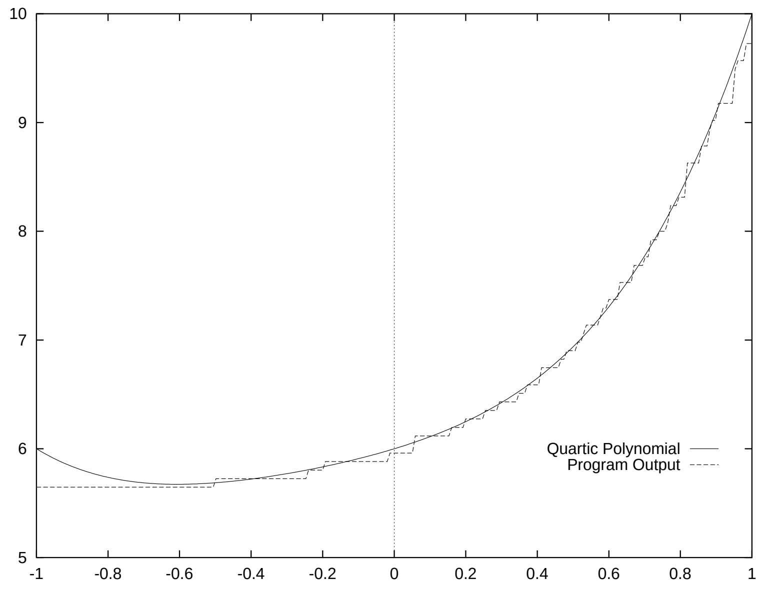

Fig. 5. Behaviour of the best best-of-run program evolved using sub-machine-code GP with populations of size 1,000 to solve the quartic polynomial problem.

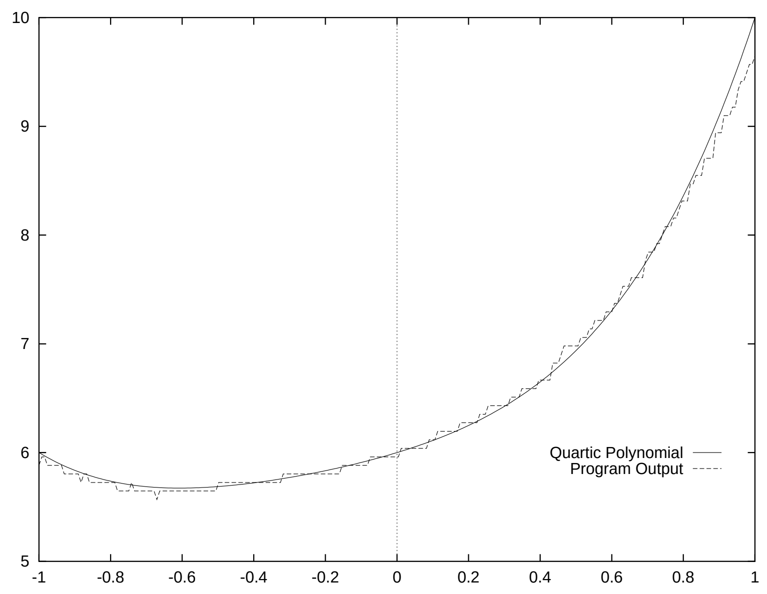

Fig. 6. Behaviour of the best program evolved using sub-machine-code GP with populations of size 100,000 to solve the quartic polynomial problem.

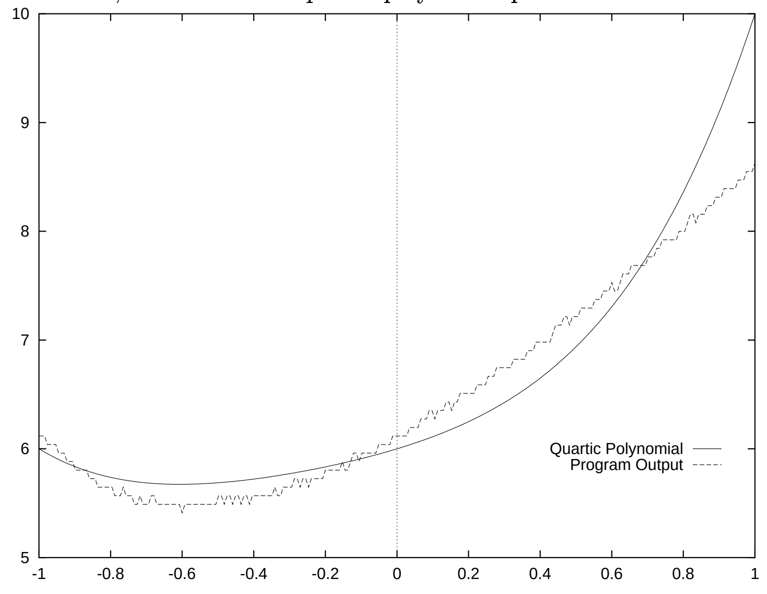

Fig. 7. Behaviour of the worst program evolved using sub-machine-code GP with populations of size 100,000 to solve the quartic polynomial problem.

evaluated during the runs was 52,404,275,616. From this we can infer that the average program size during the runs was 34.9 nodes, and that our GP system was executing 1.96×10^6 nodes per second, which, thanks to the use of SMCGP, corresponds to 15.53×10^6 primitives per second. This is a remarkable result, if one compares this with the performance of other very efficient C/C++ GP implementations. For example, in a number of tests [10, 11] using GP-QUICK [22] on the same machine on which we ran our experiments, the average primitive evaluation time was 0.8 μs for programs including floating point additions and multiplications. This corresponds to 1.25×10^6 primitives per second, which is more than 12 times slower than SMCGP.

5 Conclusions

SMCGP is a technique to exploit the internal parallelism of CPUs in GP. SMCGP can work around the SIMD nature of the CPU and evolve efficient parallel programs which solve multiple problems simultaneously. In this paper we demonstrated this on the problem of evolving parallel binary multipliers.

However, SMCGP can also be used to compute multiple fitness cases in parallel during a single execution of a program. In previous work [20] we showed how this can be done, at zero cost, in Boolean classification problems achieving speedups of up to 64 times on 64-bit processors. In this paper we have described how SMCGP can also be extended to process multiple fitness cases per program execution in continuous symbolic regression problems, indicating the limitations and tradeoffs inherent to this approach. Despite these, the speedups obtained with this technique are considerable.

All this can easily be obtained *without* any substantial change to the basic GP machinery. The only requirements to do SMCGP are a primitive set designed to exploit the internal parallelism of the CPU, and a fitness function which takes care of properly initialising/encoding the input variables and properly decoding the values returned by the interpreter. All this can easily be done in any programming language, making it possible to do SMCGP, with only minor changes, in any pre-existing GP implementation.

Acknowledgements

The author wishes to thank Nic McPhee and the members of the Evolutionary and Emergent Behaviour Intelligence and Computation (EEBIC) group at Birmingham for useful comments and discussion.

References

1. David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA, 1996.

2. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, January 1998.
3. T.C. Fogarty, J.F. Miller, and P. Thomson. Evolving digital logic circuits on Xilinx 6000 family FPGAs. In P.K. Chawdhry, R. Roy, and R.K.Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 299–305. Springer-Verlag, London, 1998.
4. Alex Fukunaga, Andre Stechert, and Darren Mutz. A genome compiler for high performance genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 86–94, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.
5. Chris Gathercole and Peter Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
6. S. Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 154–159, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.
7. Hugues Juille and Jordan B. Pollack. Massively parallel genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 17, pages 339–358. MIT Press, Cambridge, MA, USA, 1996.
8. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.
9. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Pres, Cambridge, Massachusetts, 1994.
10. W. B. Langdon and R. Poli. An analysis of the MAX problem in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 222–230, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
11. William B. Langdon. *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!* Kluwer, Boston, 24 April 1998.
12. J. F. Miller, P. Thomson, and T. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In D. Quagliarella, J. Periaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Stategies in Engineering and Computer Science: Recent Advancements and Industrial Applications.* Wiley, 1997.
13. Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.
14. Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications.* PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.
15. Peter Nordin. AIMGP: A formal description. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Stanford University Bookstore.

16. Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.
17. Riccardo Poli and W. B. Langdon. Genetic programming with one-point crossover and point mutation. Technical Report CSRP-97-13, University of Birmingham, School of Computer Science, Birmingham, B15 2TT, UK, 15 April 1997.
18. Riccardo Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 278–285, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
19. Riccardo Poli and William B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.
20. Riccardo Poli and William B Langdon. Sub-machine-code genetic programming. Technical Report CSRP-98-18, University of Birmingham, School of Computer Science, August 1998.
21. Nicol N. Schraudolph and R. K. Belew. Dynamic parameter encoding for genetic algorithms. *Machine Learning*, 9(1):9–21, 1992.
22. Andy Singleton. Genetic programming with C++. *BYTE*, pages 171–176, February 1994.
23. Kilian Stoffel and Lee Spector. High-performance, parallel, stack-based genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
24. Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.