

## Chapter 13: Sub-machine-code Genetic Programming

**Riccardo Poli and William B. Langdon**

CPUs are often seen as sequential, however they have a high degree of internal parallelism, typically operating on 32 or 64 bits simultaneously. This chapter explores the idea of exploiting this internal parallelism to extend the scope of genetic programming (GP) and improve its efficiency. We call the resulting form of GP *sub-machine-code GP*. The differences between sub-machine-code GP and the usual form of GP are purely semantic and largely language independent, i.e. any GP system can potentially be used to do sub-machine code GP. In this chapter this form of GP and some of its applications are presented. The speed up obtained with this technique on Boolean classification problems is nearly 2 orders of magnitude.

### 13.1 Introduction

Genetic Programming (GP) [Koza, 1992; Koza, 1994; Banzhaf et al., 1998] is usually seen as 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. We recall the main speedup techniques published to date in Section 13.2.

Some of these techniques are now used in many GP implementations. Thanks to this and to the fact that the power of our workstations is increasing exponentially (today's CPUs are now more than 10 times faster than those used in early GP work), nowadays we can run 50 generations 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. This is because extensive experimental GP studies (like [Langdon and Poli, 1998] or [Luke and Spector, 1998]) and complex applications (like [Poli, 1996] or [Luke, 1998]) may still require from days to months of CPU time to complete.

Most computer users consider the machines on their desks as sequential computers. However, at a lower level of abstraction CPUs are really made up of parallel components. In this sense the CPU can be seen as a Single Instruction Multiple Data (SIMD) processor. In this chapter we present a novel way of doing GP which exploits this form of parallelism to improve the efficiency and the range of applications of genetic programming. We term this form of GP *sub-machine-code GP*.

Sub-machine-code GP which extends the scope of GP to the evolution of *parallel programs running on sequential computers*. These programs are faster as, thanks to the parallelism of the CPU, they perform multiple calculations during a single program evaluation. Since these programs may be very difficult to find for human programmers and are certainly beyond the scope of current optimising compilers, this is an important new extension.

In addition, in some domains, sub-machine-code GP can be used to speed up the evaluation of sequential and parallel programs very effectively. In the chapter we show that nearly 2 orders of magnitude speedups can be achieved by any standard GP system, independently of the implementation language used, with minimal changes.

The chapter is organised as follows. After describing earlier work on speed up techniques for GP (Section 13.2), in Section 13.3 we start exploring the idea of using modern CPUs at their very lowest level of computational resolution to do genetic programming. In Section 13.4 we describe applications of sub-machine-code GP to problems where different components of the CPU perform different calculations concurrently. We illustrate how to achieve substantial speedups in the evaluation of sequential programs using sub-machine-code GP in Section 13.5. We give some indications of possible extensions of this idea and we draw conclusions in Section 13.6.

## 13.2 Background

John Koza's first book [Koza, 1992] included a number of tricks to speed up his Lisp GP implementation. However, soon people started trying to go beyond the inefficiencies of the Lisp language, and some GP implementations in C and C++ appeared (see for example [Singleton, 1994]). These can be several times faster than the equivalent Lisp implementations. So, they spread very quickly in the community and nowadays many researchers use some sort of C or C++ GP system. To go beyond the speedup provided by the language, some drastic changes are required.

Some speed up techniques rely on a better representation for the trees in the population. An example is the idea, firstly proposed by Handley [Handley, 1994], of storing the population of trees as a single directed acyclic graph, rather than as a forest of trees. This leads to considerable savings of memory (structurally identical subtrees are not duplicated) and computation (the value computed by each subtree for each fitness case can be cached). Another idea is to evolve graph-like programs, rather than tree like ones. For example, the grid-based representation used in Parallel Distributed Genetic Programming (PDGP) [Poli, 1997] has been used to evolve very efficient graph-like programs. In PDGP the speedup derives from the fact that, on problems with a high degree of regularity, partial results can be reused by multiple parts of a program without having to be recomputed.

Other techniques are based on trying to speed up as much as possible the execution of programs. For example, recently techniques have been proposed which are based on the idea of compiling GP programs either into some lower level, more efficient, virtual-machine code or even into machine code. For example, in [Fukunaga et al., 1998] a genome compiler has been proposed which transforms standard GP trees into machine code before evaluation. The possibilities offered by the Java virtual machine are also currently being explored [Klahold et al., 1998; Lukschndl et al., 1998a; Lukschndl et al., 1998b]. Well before these recent efforts, since programs are ulti-

mately executed by the CPU, other researchers proposed removing completely the inefficiency in program execution inherent in interpreting trees and directly evolve programs in machine code form [Nordin, 1994; Nordin and Banzhaf, 1995; Nordin, 1997; Nordin, 1998]. This idea has recently led to a commercial GP system called Discipulus (Register Machine Learning Technologies, Inc.) which is claimed to be at least one order of magnitude faster than any GP system based on higher-level languages.

Since the evaluation of the fitness of programs often involves their execution on many different fitness cases, some researchers have proposed speeding up fitness evaluation by reducing the number of fitness cases. For example, this can be done by identifying the hardest fitness cases for a population in one generation and evaluating the programs in the following generation only using such fitness cases [Gathercole and Ross, 1997]. Alternatively, the evaluation of fitness cases can be stopped as soon as the destiny of a particular program (e.g. whether the program will be selected to be a parent or not and, if so, how many times) is known with a big enough probability [Teller and Andre, 1997; Langdon, 1998]. A related idea is to reduce the computation load associated with program execution by avoiding the re-evaluation of parts of programs. This can be done, for example, by caching the results produced by ADFs the first time they are run with a certain set of arguments, and using the stored results thereafter [Langdon, 1998].

Finally, some research has been devoted to parallel and distributed implementations of GP (see for example [Andre and Koza, 1996; Stoffel and Spector, 1996; Juille and Pollack, 1996; Sian, 1998]). These are usually based on the idea of distributing a population across multiple machines with some form of communication between them to exchange useful genetic material. A similar, but more basic, speed up technique is to perform independent multiple runs of a same problem on different machines. It should be noted that in a sense these are not really speed up techniques, since the amount of CPU time per individual is not affected by them.<sup>1</sup>

### 13.3 Sub-machine-code GP

As indicated in the introduction CPUs can be seen as made up of a set of interacting SIMD 1-bit processors. In a CPU some instructions, such as all Boolean operations, are performed in parallel and independently for all the bits in the operands. For example, the bitwise AND operation (see Figure 13.1(a)) is performed internally by the CPU by concurrently activating a group of AND gates within the arithmetic logic unit as indicated in Figure 13.1(b). In other instructions the CPU 1-bit processors can be imagined to interact through communication channels. For example, in a shift left operation each processor will send data

---

<sup>1</sup>It has been reported in [Andre and Koza, 1996] that the use of subpopulations and a network of transputers delivered a super-linear speed-up in terms of the ability of the algorithm to solve a problem. So, the amount of CPU time per individual was reduced. This happened because partitioning the population was beneficial for the particular problem being solved. The same benefits could be obtained by evolving multiple communicating populations on a single computer.

to its left neighbour while in an add operation some 1-bit processors will send a carry bit to one of their neighbouring processors. Other operations might involve more complicated patterns of communication.

Some operations (like Boolean operations or shifts) can be imagined to be executed synchronously by all 1-bit processors at the same time. Others, like addition, require some different form of synchronisation (e.g. in some CPUs carry bits are only available after the corresponding 1-bit processors have performed their bit-addition). Nonetheless, as far as the user of a CPU is concerned the CPU 1-bit processors run in parallel, since the results of the operation of all processors become available at the same time.

If we see the CPU as a SIMD computer, then we could imagine that each of its 1-bit processors will be able to produce a result after each instruction. Most CPUs do not allow handling single bits directly. Instead all the values to be loaded into the CPU and the results produced by the CPU are packed into bit vectors, which are normally interpreted as integers in most programming languages.<sup>2</sup> For example, in many programming languages the user will see a bitwise AND operation as a function which receives two integers and returns an integer, as indicated in Figure 13.1(c).

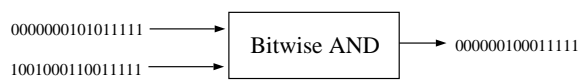
All this powerful parallelism inside our CPUs has been ignored by the GP community so far, perhaps because many of us are not used to think in terms of bits, nibbles, carries, registers, etc. For example, to the best of our knowledge, in every implementation operations like the logical AND shown in Figure 13.2(a) are executed sequentially. This exercises only one of the 1-bit processors within the CPU as indicated in Figure 13.2(b).

The simplest ways to exploit the CPU's parallelism to do GP is to make it execute the same program on different data in parallel and independently. This can be done as follows:

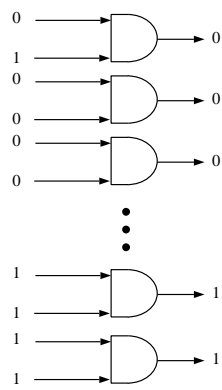
- The function set should include operations which exploit the parallelism of the CPU, e.g. bitwise Boolean operations.
- The terminal set should include integer input variables and constants, which should be interpreted as bit vectors where each bit represents the input to a different 1-bit processor. For example, the integer constant 21, whose binary representation is 00010101 (assuming an 8-bit CPU), would be seen as 1 by the 1-bit processors processing bits 1, 3 and 5. It would be seen as 0 by all other 1-bit processors.
- The result produced by the evaluation of a program should be interpreted as a bit vector, too. Each bit of this vector represents the result of a different 1-bit processor. E.g. if the output of a GP program is the integer 13, this should be converted into binary (obtaining 00001101) and decomposed to obtain 8 binary results (assuming an 8-bit CPU).

---

<sup>2</sup>Often bit vectors of different sizes are allowed (e.g. bytes, words, double words), but this is irrelevant for the present discussion.



(a)



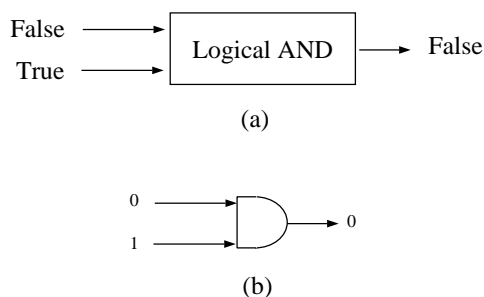
(b)



(c)

**Figure 13.1**

Three different ways of looking at a bitwise AND operation as performed by a 16-bit CPU: (a) bitwise AND between binary numbers, (b) implementation of the operation in (a) within the CPU, and (c) the same bitwise AND as seen by a CPU's user as an operation between integers.



**Figure 13.2**  
A logical AND (a), and its equivalent using AND gates (b).

To exploit the parallelism of the CPU it is not necessary to manipulate machine code directly. In fact, most high level languages include some operations which the compiler converts directly into the corresponding machine code operations. E.g. integer arithmetic operations (multiplication, addition, subtraction, division), bitwise logical operations (AND, OR, NOT), bit shift and rotate operations. By using such high level instructions in the function set of GP, it is possible to exploit the parallel nature of the CPU in most high level languages. Since the other differences w.r.t. the usual form of GP are related only to the interpretation of the inputs and outputs of each program, *any GP system can potentially be used to do sub-machine code GP.*

An ideal application for this paradigm is to evaluate multiple fitness cases in parallel. Boolean induction problems lend themselves 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 describe this in detail in Section 13.5.

More complex applications of this idea can be envisaged in which the CPU 1-bit processors, although executing the same program, can still perform different computations. Obviously, this could be done by re-wiring a CPU so as to do something it has not being designed to do (except when the CPU's microcode is stored in rewritable memory, which is not the case for the CPUs in standard workstations). However, it is possible to do it by passing different data to different processors thus making them behave differently. This can be done either by feeding different data in the various bits of the input variables, or by using constants with different bits set. For example, let us consider a 2-bit CPU which is computing a bitwise XOR between a variable  $x$  and a constant  $c = 2$  (which is the 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  $(XOR\ x\ 1)$  and will therefore return the negation of the corresponding bit in  $x$ , i.e. it will compute the function  $(NOT\ x)$ . The second 1-bit processor will instead compute  $(XOR\ x\ 0)$  and so will simply copy the corresponding bit in  $x$ , i.e. it will compute the identity function.

## 13.4 Examples

In this section we will describe a few sample applications of sub-machine-code GP. The runs described in the section were performed with our own GP implementation in Pop-11. The results presented in Section 13.5 were instead obtained with a C implementation. Reimplementing these examples within GP systems written in other languages is trivial.

### 13.4.1 1-bit and 2-bit Adder Problems

Let us consider the problem of evolving a 1-bit adder program based on Boolean operations. The adder has two inputs `a` and `b` and two outputs `sum` and `carry`. The truth table for the adder is:

a	b	sum	carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Each row in this table can be used as a fitness case.

This problem would be extremely easy to solve by hand if one could use two separate program trees to express the solution. In that case it would be sufficient to use  $(XOR\ a\ b)$  to compute `sum` and  $(AND\ a\ b)$  to compute `carry`. However, by exploiting the parallelism in the CPU it is possible to perform both computations at the same time. This could be done by slicing the output of a *single* program into its component bits and interpreting one of them as `sum` and one as `carry`. Of course to do this it would be necessary to include in the program some appropriate constants which would excite differently different parallel components of the CPU.

This version of the problem is much harder to solve for humans, but can be easily solved by evolution using sub-machine-code GP. To do that and to keep the example simple we decided to use only the bit-0 and bit-1 processors within the CPU since there are only two outputs in this problem. The function set was  $\{AND, OR, NOT, XOR\}$ , where all functions were implemented using bitwise operations on integers. The terminal set was  $\{x1, x2, 1, 2\}$ . The constants 1 and 2 were selected since they have the binary representation 01 and 10 which allows GP to modify selectively the behaviour of different parts of the program for different 1-bit processors.

For each of the four fitness cases, the variable `x1` took the value 3 (which corresponds to the binary number 11) when the corresponding entry in column `a` of the truth table above was 1, the value 0 (i.e. the binary number 00) otherwise. Likewise, `x2` took the value 3 when `b` was 1, 0 otherwise. This was done to provide the same inputs both to the bit-0 processor and to the bit-1 processor of the CPU.

Each program in the population was run with each of the four possible settings for the variables `x1` and `x2`. The two least significant bits of the integer produced as output in each fitness case were compared with the target values for `sum` and `carry` indicated in the previous table. The fitness function was the sum of the number of matches between such bits, i.e. a perfect program had a fitness of 8, 4 for `sum` and 4 for `carry`.

Except for the fitness function, no change of our standard-GP system was necessary to run sub-machine-code GP. As far as GP was concerned it was solving, using bitwise primitives, a symbolic regression problem with fitness cases:

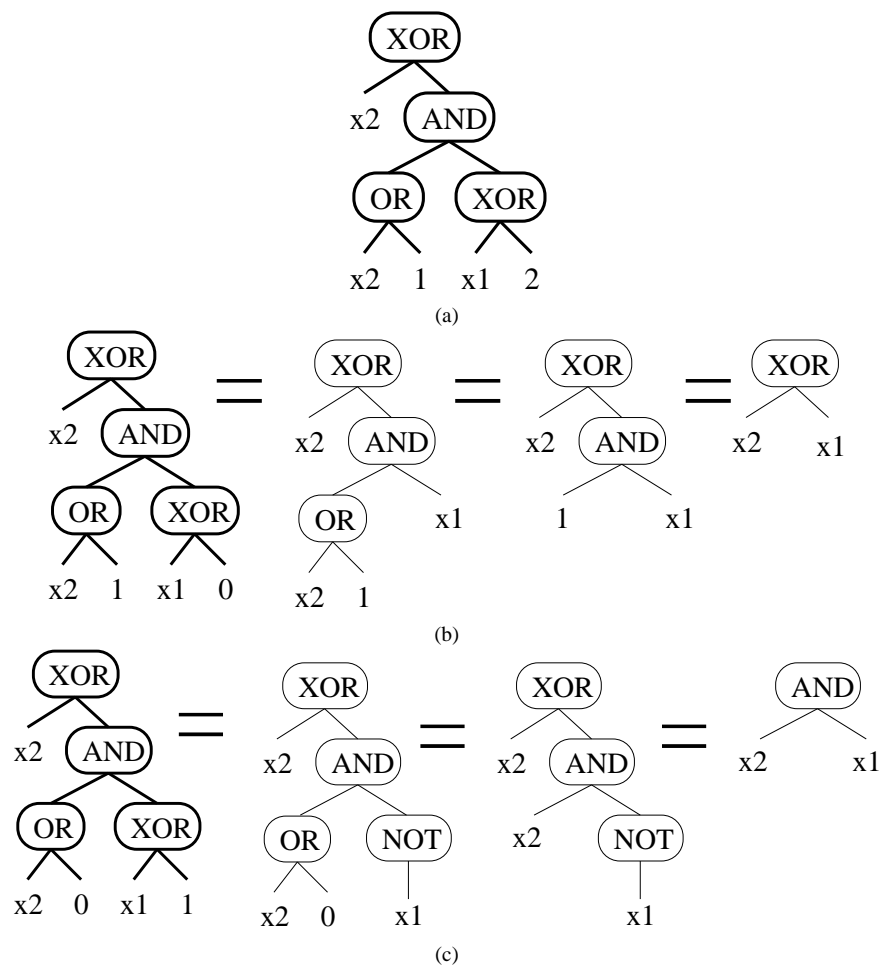
<code>x1</code>	<code>x2</code>	target
0	0	0
3	0	2
0	3	2
3	3	1

rather than the 1-bit adder problem. Under this interpretation, in the fitness function the program outputs and the target outputs were compared to produce a fitness measure in a way quite unusual for symbolic regression problems. For example, if in one fitness case the program output was 2 (binary 10) and the target was 1 (01) that gave a contribution of 0 to the total fitness, while if the program output was 3 (11) the fitness contribution of the fitness case was 1!

In our runs we used a population of 1000 individuals, up to 100 generations, standard subtree crossover (with uniform probability of selection of the crossover points) with crossover probability 0.7, no mutation 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 50 nodes.

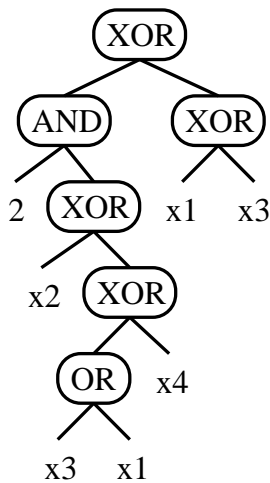
Figure 13.3(a) shows a solution to this problem discovered by GP at generation 3 of a run. As far as the bit-0 processor of the CPU (the one which computes `sum`) is concerned this program is equivalent to the one shown on the left of Figure 13.3(b). This version of the program has been obtained by replacing the constants in Figure 13.3(a) with the LSB of their binary representation (i.e. 1 (binary 01) is replaced with 1 and 2 (binary 10) with 0). As shown on the right of the same figure, as expected this program is equivalent to the XOR function. Figure 13.3(c) shows the program as seen by the bit-1 processor (the one which computes `carry`), which is equivalent to the AND function.

In another run we solved also the two-bit adder (without carry) problem which requires the use of four input variables: `x1`, `x2` which represent the least significant bit and the most significant bit of the first operand, respectively, and `x3` and `x4` to represent the second operand. All the other parameters were the same as in the 1-bit adder problem except that we included also the constants 0 and 3 in the terminal set and that the output bits are interpreted as `sum1` and `sum2` rather than `sum` and `carry`. Figure 13.4 shows a solution found by sub-machine-code GP at generation 11.



**Figure 13.3**

(a) Program evolved using sub-machine-code GP to solve the 1-bit adder problem, (b) the same program as seen by the bit-0 processor within the CPU (in thick line) and some of its simpler equivalents (in thin line), and (c) the corresponding program executed by the bit-1 processor and its simplifications.

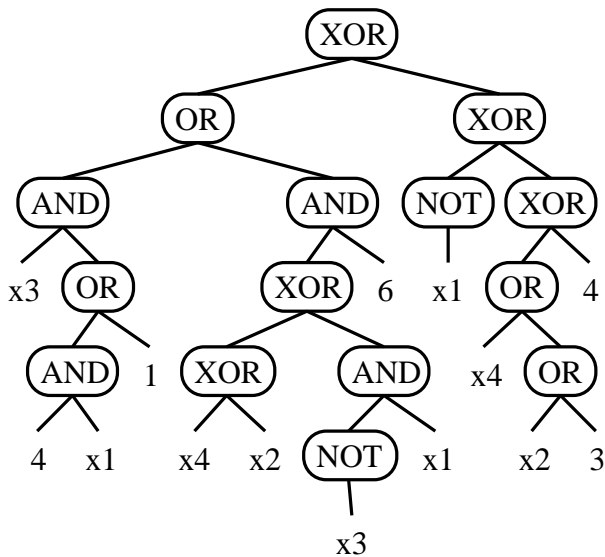


**Figure 13.4**  
Program evolved using sub-machine-code GP to solve the 2-bit adder problem without carry.

By slightly modifying the setup of the experiments described above we were also able to evolve a solution to the two-bit adder problem where the carry bit is also computed. This problem has three output bits and so requires using three 1-bit processors. So, the inputs variables  $x_1$ ,  $x_2$ ,  $x_3$  and  $x_4$  (which have the same meaning as in the previous paragraph) took the values 0 and 7 (i.e. binary 111) depending on whether the corresponding bit of the 2-bit adder truth table was 0 or 1. The constants 0, 1, 2, 3, 4, 5, 6 and 7 were included in the terminal set. The fitness function was the sum (over all the 16 possible combinations of inputs) of the number of matches between the actual output of each program (i.e. `sum1`, `sum2` and `carry`) and the desired output (maximum score 48) decreased by 0.001 times the number of nodes in a program. This was done to promote parsimonious solutions. All the other parameters were as in the previous experiments. A solution to this problem was found by GP at generation 27 of a run. The solution included 80 nodes. By allowing the run to continue, by generation 69 GP had simplified this solutions down to 29 nodes. The simplified solution is shown in Figure 13.5.

### 13.4.2 Character Recognition Problem

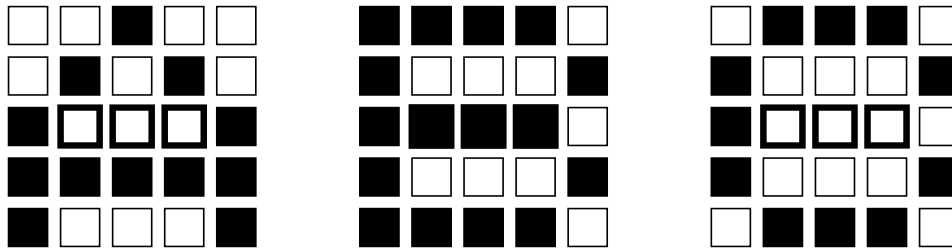
In the adder problems the 1-bit processors used to perform the task did not communicate in any way. However, in some problems, like the character recognition problem described below, adding some form of communication between processors can be useful.



**Figure 13.5**  
 Program evolved using sub-machine-code GP to solve the 2-bit adder problem with carry.

The character recognition problem we considered to illustrate this idea included only three input patterns, an A, a B and a C, represented by the  $5 \times 5$  bitmaps shown in Figure 13.6. The black squares represent 1's and white ones represent 0's of the corresponding bit map (the additional role of the squares with gray borders is explained below). Each character represented one fitness case. The objective was to evolve a program capable of identifying the three characters.

Each character in Figure 13.6 was represented by a 25-bit binary string by reading the bits from each bit map from left to right and top to bottom. This produces the strings  $A=0010001010100011111110001$ ,  $B=111101000111101000111110$  and  $C=0111010001100001000101110$ . So we decided to use 25 1-bit processors. The terminal set included only one variable,  $x$ , and random integer constants between 0 and 33554431 (which corresponds to the binary number 111111111111111111111111). The variable  $x$  takes the values 4540401, 32045630 or 15254062 which are the decimal equivalents of the bit-strings representing the characters A, B and C, respectively. In order to determine which character was present in input we designated the bit-11, bit-12 and bit-13 processors as result returning processors (i.e. all the bits of the integer resulting from the evaluation of each program were ignored except bits 11, 12 and 13). When an A was presented in input, the target output was the pattern 100. When a B was presented the target was 010. The



**Figure 13.6**  
A, B and C bitmaps used in the character recognition problem.

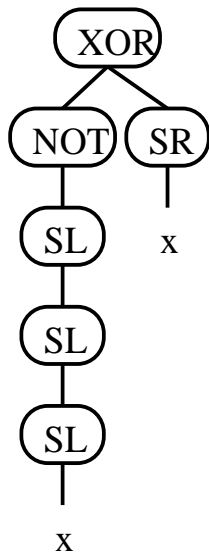
target was 001 when C was the input. The fitness function was the sum (over the 3 possible inputs, A, B and C) of the number of matches between the actual output and the desired output for bits 11, 12, and 13 (maximum score 9) decreased by 0.001 times the number of nodes in a program.

Bits 11, 12 and 13 were chosen as outputs since the characters A and C cannot be differentiated just by looking at bits 11, 12 and 13 of their binary representations as it can be easily inferred by comparing the patterns in the squares with gray borders in Figure 13.6. Therefore, this problem could not be solved without allowing some form of communication between processors. To keep things as simple as possible we decided to try using simple bit shift operations to start with. So, we used the function set {NOT, OR, AND, XOR, SL, SR}, where SR (SL) is an arity-1 function which performs a 1-bit shift to the right (left) of the bit pattern representing its integer argument.

Expecting this problem to be much harder than the previous ones we used a population of 10,000 individuals in our runs. The other parameters were as in the 2-bit adder with carry problem. However, the problem was in fact quite easy to solve: a solution was found at generation 0 of each of the five runs we performed. In one run the generation-0 solution included 31 nodes, but by allowing the run to continue for 7 more generations we obtained the simplified 8-node solution shown in Figure 13.7.

This solution is quite clever. It uses communication to create two modified (shifted) versions of the input bit pattern. In the first version the characters B and C can be differentiated from A by looking at bits 11, 12 and 13. In the second version A and C can be differentiated from B. Then by combining such modified bit patterns with an XOR not only the three characters can be properly differentiated, but the desired encoding for the result is also achieved. The steps performed by the program when A, B or C are presented in input are shown in Table 13.1.

The only changes necessary to solve this problem with our GP implementation in Pop-11 were within the fitness function. To clarify how little effort these required we report in Figure 13.8 the actual implementation of the fitness function used in the character recognition experiments.



**Figure 13.7**  
Program evolved using sub-machine-code GP to solve the character recognition problem.

### 13.4.3 Discussion

The evolution of the programs described in the previous subsections was very quick except for the 2-bit adder with carry, which was solved only in 2 runs out of 10. At this stage we cannot say whether evolving one program that does more than one job using sub-machine-code GP *always* requires less effort than evolving separate programs in independent runs. It is entirely possible that the constraints imposed by the SIMD nature of the CPU will make the search harder when one wants to use several 1-bit processors to do totally unrelated tasks. In this case, it would be possible that the advantage in terms of evaluation time offered by sub-machine-code GP programs would be outweighed by an increased number of evaluations required to solve the problem.

Given the relative difficulty with which the 2-bit adder with carry was solved, we expected that this would happen in the character recognition problem where we used 25 processors. However, evolution seemed to be greatly facilitated by the representation adopted. So, clearly there are domains where sub-machine-code GP can exploit the CPU parallelism fully. Even if there are cases where finding a parallel program with sub-machine-code GP is hard, if one is able to find one such program, then execution of it will presumably still be much faster than if using multiple standard sequential programs. This advantage may

**Table 13.1**

Steps in the execution of the program in Figure 13.7. The bits read as the output of the program are shown in bold.

Subexpression	Input pattern		
	A	B	C
x	00100	11110	01110
	01010	10001	10001
	<b>10001</b>	<b>11110</b>	<b>10000</b>
	11111	10001	10001
	10001	11110	01110
(SL (SL (SL x)))	00010	10100	10100
	10100	01111	01100
	<b>01111</b>	<b>10100</b>	<b>00100</b>
	11100	01111	01011
	01000	10000	10000
(NOT (SL (SL (SL x))))	11101	01011	01011
	01011	10000	10011
	<b>10000</b>	<b>01011</b>	<b>11011</b>
	00011	10000	10100
	10111	01111	01111
(SR x)	00010	01111	00111
	00101	01000	01000
	<b>01000</b>	<b>11111</b>	<b>11000</b>
	11111	01000	01000
	11000	11111	10111
(XOR (NOT (SL (SL (SL x)))) (SR x))	11111	00100	01100
	01110	11000	11011
	<b>11000</b>	<b>10100</b>	<b>00011</b>
	11100	11000	11100
	01111	10000	11000

become prevalent over the search effort, whenever the programs are used in applications where they are run repeatedly and frequently for an extended period of time.

CPUs are not the only computational devices available within modern workstations. For example, devices like graphic accelerators can also perform certain forms of computation in parallel on huge quantities of data very efficiently. For example, most graphic cards are able to shift and to perform bitwise operations on large portions of their video memory in microseconds using specialised, very high speed processors. Good graphics libraries will exploit such operations. Sub-machine-code GP could be run on a video card, rather than on the CPU, with minimum effort (perhaps one day we will even have sub-machine-code GP screen savers!). Some CPUs also include similar specialised high-throughput graphics operations (like the MMX extensions on Pentium processors) which could be exploited to do GP.

```

define char_rec_fitness_function( prog );
  vars match_count = 0, target, output, x, i;

  fast_for x, target in_vectorclass      ;; This FOR loop binds X and
                                          ;; TARGET simultaneously

    {2:0010001010100011111110001 ;; Inputs A, B, C
     2:1111010001111101000111110 ;; (2:XXXX = XXXX in base 2)
     2:0111010001100001000101110},

    {2:00000000000100000000000000 ;; Desired outputs for A, B, C
     2:00000000000100000000000000 ;; (all bits ignored except 11,
     2:000000000001000000000000};; 12 and 13)
  do

    eval( prog ) -> output;      ;; Run evolved program

    for i from 11 to 13 do
      if getbit(i,output) == getbit(i,target) then
        1 + match_count -> match_count;
      endif;
    endfor;

  endfor;
  match_count - 0.001 * nodes( prog ) -> fitness( prog );
enddefine;

```

**Figure 13.8**  
Pop-11 implementation of the character recognition problem fitness function.

### 13.5 Fast Parallel Evaluation of Fitness Cases

As indicated in the previous sections, sub-machine-code GP offers many advantages. However, given the SIMD nature of the CPU, it might also require an increased number of evaluations to solve problems where the 1-bit processors of a CPU are required to perform unrelated tasks in parallel. These additional search costs disappear when using the CPU processors to do exactly the *same* task but on different input data.

One such case is the use of sub-machine-code GP to evaluate multiple fitness cases in parallel. This can be done very easily in Boolean classification problems. The approach used is a simple modification of the approach used in the examples in the previous sections. The only differences with respect to standard GP are: independently. This can be done as follows:

- Bitwise Boolean functions are used.
- Before each program execution the input variables need to be initialised so as to pass a different fitness case to each of the different 1-bit processors of the CPU.

- The output integers produced by a program need to be unpacked since each of their bits has to be interpreted as the output for a different fitness case.

In the Appendix we provide a simple C implementation of this idea which demonstrates the changes necessary to do sub-machine-code GP when solving the even-5 and even-10 parity problems.

In practical terms this evaluation strategy means that all the fitness cases associated with 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. Since this can be done with any programming language, this technique could lead to speedups of up to 1.5 or 1.8 orders of magnitude.

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. For example, it is possible to pre-pack the values for the input variables and store them in a table (this has been done only in part in the code in the Appendix). If also the targets are precomputed (we did not do that in our implementation), in many problems the only computation required with the output returned by a program would be the calculation of the Hamming distance between two integers.

The implementation in the Appendix (with a slightly different main function) was used to perform an evaluation of the overheads in sub-machine-code GP. We ran our tests on an Sun Ultra-10 300MHz workstation using a 32-bit compiler. In the tests we first evaluate 1,000,000 times the 20-node program `(NOT (XOR (X1 (XOR (X2 (XOR (X3 (XOR (X4 (XOR X5 (XOR X6 (XOR X7 (XOR X8 (XOR X9 X10)) . . . .))) . . . .))) . . . .))` using the even-10 parity function which involves 1024 test cases. This required 134 seconds of CPU time. Running the one-node program `x1` required 38 seconds. The difference between the two, 96 seconds, indicates that our implementation is able to evaluate

$$\underbrace{(20 - 1)}_{\text{nodes}} \times \underbrace{1,000,000}_{\text{evaluations}} \times \left( \underbrace{1024}_{\text{fitness cases}} / \underbrace{32}_{\text{CPU bits}} \right) / \underbrace{96}_{\text{CPU time}} \approx 6,300,000$$

primitives per second. However, there are 38 seconds of overheads (evaluating the target, unpacking, etc.), which reduce the actual number of primitives per second to around 4,800,000 (i.e.  $20 \times 1,000,000 \times (1024/32)/134$ ). So, rather than a 32-fold speed up we obtain a speed up of approximately 24 times for programs of 20 nodes. However, the speed up is better for longer programs. For example, for a program including 200 nodes the actual number of primitives per second is around 6,100,000 with a 31-fold speedup. To match this with standard GP one would have to find a computer capable of evaluating around 190 million primitives per second (in C): not an easy task. For larger programs the speed up can be even better. For 64 bit machines these performance figures can be substantially improved. Indeed we found speedups exceeding 60-fold.

In a typical run of our C implementation of sub-machine-code GP on the even-10 parity problem, a DEC Alpha 500 workstation with a 400MHz 64-bit CPU can evolve a population of 1000 individuals for 100 generations in 140.6 seconds. The number of nodes evaluated by the system in this time is more than 1.2 billion ( $10^9$ ) which is approximately 8,600,000 nodes per second. This figure needs to be multiplied by 64 to obtain the number of primitives per second. The result is above 550 millions, which is bigger than the clock speed of the machine (400MHz). Sub-machine-code GP executes 1.3 operations per clock tick. Since this is more than any other GP implementation (where at most one instruction per clock tick is executed), the claim that on 64-bit machines our GP implementation is the fastest in the world would seem justifiable.

Together with other new techniques, sub-machine-code GP has allowed us to solve very high-order parity problems without ADFs (manuscript in preparation).

Finally, it should be noted that the technique described in this section and those used in the previous section can be combined. For example, if for a particular problem only a small number of 1-bit processors is necessary, it is then possible to evaluate multiple fitness cases of the same problem using the remaining processors.

### 13.6 Conclusions

In this chapter we have presented a GP technique which exploits the internal parallelism of CPUs, either to evolve efficient programs which solve multiple problems in parallel, or to compute multiple fitness cases in parallel during a single execution of a program. We call this form of genetic programming sub-machine-code GP. We have demonstrated remarkable speedups and presented examples where we have evolved parallel programs which can be executed directly and efficiently on standard computer hardware.

Sub-machine-code GP has a considerable potential which we will continue to explore in future research. One particularly interesting issue is whether it is possible to use this form of GP to solve efficiently also problems which require primitives with side effects. Another issue is to see on which classes of problems the bias imposed by the SIMD nature of the CPU is not a limit but an advantage for sub-machine-code GP.

The potential of sub-machine-code GP is real and already available. This is demonstrated by the 24- to 60-fold speedups it can achieve in Boolean classification problems without requiring any significant change to the GP system used and independently of the language it is implemented in. On a 64-bit machine, with this technique GP is able to evaluate 64 fitness cases with a single program evaluation. This means that the fitness function for a problem like the even-10 parity problem, which has never been solved with standard GP without ADFs or recursion, now has about the same computation load as the fitness function of an even-4 parity problem.

On Boolean classification problems, the speedup achieved by sub-machine-code GP makes our 64-bit C implementation of the fastest GP system in the world. If we believe

Moore's law (which predicts a doubling in speed of computers every 1.5 years), the speed up obtained is equivalent to the one we should expect to obtain in approximately 10 years using standard GP.

### 13.A Appendix: Implementation

This appendix describes a simple C program which illustrates the ideas behind the fast parallel evaluation of fitness cases with sub-machine-code GP. The code has been only partly optimised. No optimisation has been performed in the packing of program inputs, in the target output determination, and in the comparison between actual and target output.

#### 13.A.1 Description

The program includes the following functions:

`run()` is a simple interpreter capable of handling variables and a small number of Boolean functions. 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.

`e5parity()` computes the target output in the even-5 parity problem for a group of 32 fitness cases.

`e10parity()` does the same but in the even-10 parity problem.

`even5_fitness_function( char *Prog )` given a vector of characters representing a program, executes it and returns the number of entries of the even-5 parity truth table correctly predicted by the program.

`even10_fitness_function( char *Prog )` does the same for the even-10 parity problem. In this case the program is executed 32 times (instead of 1024), once of each iteration of the five `for` loops. These loops are used to make the interpreter evaluate the program on a different part of the truth table of the even-10 parity function. This is why the variables `x1` to `x5` initialised in the loops take the binary values 000...000 or 111...111 (i.e. FFFFFFFF in hexadecimal).

`main()` runs the even-5 parity fitness function on two programs: a non-solution, `(XOR X1 X2)`, and a solution, `(NOT (XOR (X1 (XOR (X2 (XOR (X3 (XOR (X4 (X5))))))))))`. It then does the same for the even-10 parity fitness function with the programs `(XOR X1 X2)` (a non-solution) and `(NOT (XOR (X1 (XOR (X2 (XOR (X3 (XOR (X4 (XOR X5 (XOR X6 (XOR X7 (XOR X8 (XOR X9 X10))))))))))))))` (a solution).

## 13.A.2 Code

```
#include <stdio.h>

enum {X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, NOT, AND, OR, XOR};

unsigned long x1, x2, x3, x4, x5, x6, x7, x8, x9, x10;
char *program;

unsigned long run() /* Interpreter */
{
    switch ( *program++ )
    {
        case X1 : return( x1 );
        case X2 : return( x2 );
        case X3 : return( x3 );
        case X4 : return( x4 );
        case X5 : return( x5 );
        case X6 : return( x6 );
        case X7 : return( x7 );
        case X8 : return( x8 );
        case X9 : return( x9 );
        case X10 : return( x10 );
        case NOT : return( ~run() ); /* Bitwise NOT */
        case AND : return( run() & run() ); /* Bitwise AND */
        case OR : return( run() | run() ); /* Bitwise OR */
        case XOR : return( run() ^ run() ); /* Bitwise XOR */
    }
}

unsigned long e5parity() /* Bitwise Even-5 parity function */
{
    return(~(x1^x2^x3^x4^x5)); /* (NOT (XOR x1 (XOR x2 (XOR x3 (XOR x4 x5)))) */
}

int even5_fitness_function( char *Prog ) /* Even-5 parity fitness function */
{
    char i;
    int fit = 0;
    unsigned long result, target, matches, filter;

    x1 = 0x0000ffff; /* x1 = 00000000000000001111111111111111 */
    x2 = 0x00ff00ff; /* x2 = 00000000111111110000000011111111 */
    x3 = 0x0f0f0f0f; /* x3 = 00001111000011110000111100001111 */
    x4 = 0x33333333; /* x4 = 00110011001100110011001100110011 */
    x5 = 0xaaaaaaaa; /* x5 = 01010101010101010101010101010101 */

    program = Prog;
    result = run();

    target = e5parity();
    matches = ~(result ^ target); /* Find bits where TARGET = RESULT */
    filter = 1;
    for( i = 0; i < 32; i ++ ) /* Count bits set in MATCHES */
    {
        if( matches & filter ) fit ++;
        filter <<= 1;
    }

    return( fit );
}
```

```

unsigned long e10parity() /* Bitwise Even-10 parity function */
{
    return ~(x1^x2^x3^x4^x5^x6^x7^x8^x9^x10);
}

int even10_fitness_function( char *Prog ) /* Even-10 parity fitness function */
{
    char cx1, cx2, cx3, cx4, cx5, i;
    int fit = 0;
    unsigned long result, target, matches, filter;

    for( cx1 = 0; cx1 < 2; cx1 ++ ) /* Set x1, ..., x5 to 000...000 and 111....111 */
    {
        x1 = cx1 ? 0 : 0xffffffff;
        for( cx2 = 0; cx2 < 2; cx2 ++ )
        {
            x2 = cx2 ? 0 : 0xffffffff;
            for( cx3 = 0; cx3 < 2; cx3 ++ )
            {
                x3 = cx3 ? 0 : 0xffffffff;
                for( cx4 = 0; cx4 < 2; cx4 ++ )
                {
                    x4 = cx4 ? 0 : 0xffffffff;
                    for( cx5 = 0; cx5 < 2; cx5 ++ )
                    {
                        x5 = cx5 ? 0 : 0xffffffff;
                        x6 = 0x0000ffff; /* x6 = 00000000000000001111111111111111 */
                        x7 = 0x00ff00ff; /* x7 = 00000000111111110000000011111111 */
                        x8 = 0x0f0f0f0f; /* x8 = 00001111000011110000111100001111 */
                        x9 = 0x33333333; /* x9 = 00110011001100110011001100110011 */
                        x10 = 0xaaaaaaaa; /* x10 = 01010101010101010101010101010101 */

                        program = Prog;
                        result = run();

                        target = e10parity();
                        matches = ~(result ^ target); /* Bits where TARGET = RESULT */
                        filter = 1;
                        for( i = 0; i < 32; i ++ ) /* Count bits set in MATCHES */
                        {
                            if( matches & filter ) fit ++;
                            filter <<= 1;
                        }
                    }
                }
            }
        }
    }

    return( fit );
}

```

```

void main()
{
    /* Incorrect solution */
    char s1[] = {XOR, X1, X2};

    /* Even-5 parity solution */
    char s2[] = {NOT, XOR, X1, XOR, X2, XOR, X3, XOR, X4, X5};

    /* Even-10 parity solution */
    char s3[] = {NOT, XOR, X1, XOR, X2, XOR, X3, XOR, X4, XOR, X5,
                XOR, X6, XOR, X7, XOR, X8, XOR, X9, X10};

    printf("Even-5 Problem\n"
           "Testing (XOR X1 X2)\n"
           "Score %d\n\n", even5_fitness_function( s1 ) );

    printf("Even-5 Problem\n"
           "Testing (NOT (XOR (X1 (XOR (X2 (XOR (X3 (XOR (X4 (X5))))))))))\n"
           "Score %d\n\n", even5_fitness_function( s2 ) );

    printf("Even-10 Problem\n"
           "Testing (XOR X1 X2)\n"
           "Score %d\n\n", even10_fitness_function( s1 ) );

    printf("Even-10 Problem\n"
           "Testing (NOT (XOR (X1 (XOR (X2 (XOR (X3 (XOR (X4 \n"
           "(XOR X5 (XOR X6 (XOR X7 (XOR X8 (XOR X9 X10))))))))))\n"
           "Score %d\n\n", even10_fitness_function( s3 ) );
}

/* This file is also available at:
   ftp://ftp.cs.bham.ac.uk/pub/authors/R.Poli/code/smc_gp.c */

```

## Acknowledgements

The authors wish to thank the members of the Evolutionary and Emergent Behaviour Intelligence and Computation (EEBIC) group at Birmingham for useful comments and discussion.

## Bibliography

- Andre, D. and Koza, J. R. (1996), "Parallel genetic programming: A scalable implementation using the transputer network architecture," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinneer, Jr. (Eds.), Chapter 16, pp 317–338, Cambridge, MA, USA: MIT Press.
- Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998), *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann, dpunkt.verlag.
- Fukunaga, A., Stechert, A., and Mutz, D. (1998), "A genome compiler for high performance genetic programming," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (Eds.), pp 86–94, University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.
- Gathercole, C. and Ross, P. (1997), "Tackling the boolean even N parity problem with genetic programming and limited-error fitness," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 119–127, Stanford University, CA, USA: Morgan Kaufmann.

- Handley, S. (1994), "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*, pp 154–159, Orlando, Florida, USA: IEEE Press.
- Juille, H. and Pollack, J. B. (1996), "Massively parallel genetic programming," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr. (Eds.), Chapter 17, pp 339–358, Cambridge, MA, USA: MIT Press.
- Klahold, S., Frank, S., Keller, R. E., and Banzhaf, W. (1998), "Exploring the possibilities and restrictions of genetic programming in Java bytecode," in *Late Breaking Papers at the Genetic Programming 1998 Conference*, J. R. Koza (Ed.), University of Wisconsin, Madison, Wisconsin, USA: Stanford University Bookstore.
- Koza, J. R. (1992), *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press.
- Koza, J. R. (1994), *Genetic Programming II: Automatic Discovery of Reusable Programs*, Cambridge, Massachusetts: MIT Press.
- Langdon, W. B. (1998), *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, Boston: Kluwer.
- Langdon, W. B. and Poli, R. (1998), "Why ants are hard," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (Eds.), pp 193–201, University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.
- Luke, S. (1998), "Genetic programming produced competitive soccer softbot teams for robocup97," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (Eds.), pp 214–222, University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.
- Luke, S. and Spector, L. (1998), "A revised comparison of crossover and mutation in genetic programming," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (Eds.), pp 208–213, University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.
- Lukschandl, E., Holmlund, M., and Moden, E. (1998a), "Automatic evolution of Java bytecode: First experience with the Java virtual machine," in *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, R. Poli, W. B. Langdon, M. Schoenauer, T. Fogarty, and W. Banzhaf (Eds.), pp 14–16, Paris, France: CSRP-98-10, The University of Birmingham, UK.
- Lukschandl, E., Holmlund, M., Moden, E., Nordahl, M., and Nordin, P. (1998b), "Induction of Java bytecode with genetic programming," in *Late Breaking Papers at the Genetic Programming 1998 Conference*, J. R. Koza (Ed.), University of Wisconsin, Madison, Wisconsin, USA: Stanford University Bookstore.
- Nordin, P. (1994), "A compiling genetic programming system that directly manipulates the machine code," in *Advances in Genetic Programming*, K. E. Kinnear, Jr. (Ed.), Chapter 14, pp 311–331, MIT Press.
- Nordin, P. (1997), *Evolutionary Program Induction of Binary Machine Code and its Applications*, PhD thesis, der Universität Dortmund am Fachbereich Informatik.
- Nordin, P. (1998), "AIMGP: A formal description," in *Late Breaking Papers at the Genetic Programming 1998 Conference*, J. R. Koza (Ed.), University of Wisconsin, Madison, Wisconsin, USA: Stanford University Bookstore.
- Nordin, P. and Banzhaf, W. (1995), "Evolving turing-complete programs for a register machine with self-modifying code," in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, L. Eshelman (Ed.), pp 318–325, Pittsburgh, PA, USA: Morgan Kaufmann.
- Poli, R. (1996), "Genetic programming for image analysis," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 363–368, Stanford University, CA, USA: MIT Press.
- Poli, R. (1997), "Evolution of graph-like programs with parallel distributed genetic programming," in *Genetic Algorithms: Proceedings of the Seventh International Conference*, T. Back (Ed.), pp 346–353, Michigan State University, East Lansing, MI, USA: Morgan Kaufmann.
- Sian, C. F. (1998), "A java based distributed approach to genetic programming on the internet," Master's thesis, Computer Science, University of Birmingham.

Singleton, A. (1994), "Genetic programming with C++," *BYTE*, pp 171–176.

Stoffel, K. and Spector, L. (1996), "High-performance, parallel, stack-based genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 224–229, Stanford University, CA, USA: MIT Press.

Teller, A. and Andre, D. (1997), "Automatically choosing the number of fitness cases: The rational allocation of trials," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 321–328, Stanford University, CA, USA: Morgan Kaufmann.