# 17 Massively Parallel Genetic Programming

**Hugues Juillé** and **Jordan B. Pollack**

As the field of Genetic Programming (GP) matures and its breadth of application increases, the need for parallel implementations becomes absolutely necessary. The transputer-based system presented in the chapter by Koza and Andre ([11]) is one of the rare such parallel implementations. Until today, no implementation has been proposed for parallel GP using a SIMD architecture, except for a data-parallel approach ([20]), although others have exploited workstation farms and pipelined supercomputers. One reason is certainly the apparent difficulty of dealing with the parallel evaluation of different S-expressions when only a single instruction can be executed at the same time on every processor. The aim of this chapter is to present such an implementation of parallel GP on a SIMD system, where each processor can efficiently evaluate a different S-expression. We have implemented this approach on a MasPar MP-2 computer, and will present some timing results. To the extent that SIMD machines, like the MasPar are available to offer cost-effective cycles for scientific experimentation, this is a useful approach.

## 17.1 Introduction

The idea of simulating a MIMD machine using a SIMD architecture is not new ([8, 15]). One of the original ideas for the Connection Machine ([8]) was that it could simulate other parallel architectures. Indeed, in the extreme, each processor on a SIMD architecture can simulate a universal Turing machine (TM). With different turing machine specifications stored in each local memory, each processor would simply have its own tape, tape head, state table and state pointer, and the simulation would be performed by repeating the basic TM operations simultaneously. Of course, such a simulation would be very inefficient, and difficult to program, but would have the advantage of being really MIMD, where no SIMD processor would be in idle state, until its simulated machine halts.

Now let us consider an alternative idea, that each SIMD processor would simulate an individual stored program computer using a simple instruction set. For each step of the simulation, the SIMD system would sequentially execute each possible instruction on the subset of processors whose next instruction match it. For a typical assembly language, even with a reduced instruction set, most processors would be idle most of the time.

However, if the set of instructions implemented on the virtual processor is very small, this approach can be fruitful. In the case of Genetic Programming, the "instruction set" is composed of the specified set of functions designed for the task. We will show below that with a precompilation step, simply adding a *push*, a *conditional*, and *unconditional branching* and a *stop* instruction, we can get a very effective MIMD simulation running.

This chapter reports such an implementation of GP on a MasPar MP-2 parallel computer. The configuration of our system is composed of 4K processor elements

(PEs). This system has a peak performance of $17,000$ Mips or $1,600$ Mflops. In the maximal configuration, with 16K PEs, the speed quadruples. As an example, using a population of 4096 members, we achieved more than 30 generations/minutes on the trigonometric identities problem, and up to 5 matches per second for each individual for the co-evolution of Tic-Tac-Toe players.
Section 2 describes the implementation of the kernel of our current GP, which deals with the evaluation of S-expressions. Then, the implementation of different models for fitness evaluation and interactions among individuals of the population are presented in section 3. Results and performance are presented in section 4.

## 17.2 Description of the implementation

### 17.2.1 The Virtual Processor

The individual structures that undergo adaptation in GP are represented by expression trees composed from a set of primitive functions and a set of terminals (either variables or functions of no argument). Usually, the number of functions is small, and the size of the expression trees are restricted, in order to restrict the size of the search space.
In our implementation, each PE simulates a virtual processor. This virtual processor is a *Stack Machine* which is composed of the following elements:

- a memory block where the program is stored,
- a memory block where constants and variables are stored,
- a stack where intermediate results are stored.
- a set of registers: the Instruction Pointer (IP), the Stack Pointer (SP) and general purpose registers: $A_0$, $A_1$, $\ldots$, $A_n$.

Figure 1 presents the memory mapping and registers of the virtual processor.
To be able to evaluate a GP expression, the following instructions are supported by the abstract machine:

- one instruction for each primitive function of the function set. At execution time, arguments for these instructions are popped from the stack into general purpose registers, the function is computed, and the result is pushed on the top of the stack.
- a `PUSH` instruction which pushes on the top of the stack the value of a terminal,
- a `IFGOTO` and a `GOTO` instruction which are necessary for branching if conditional functions are used,

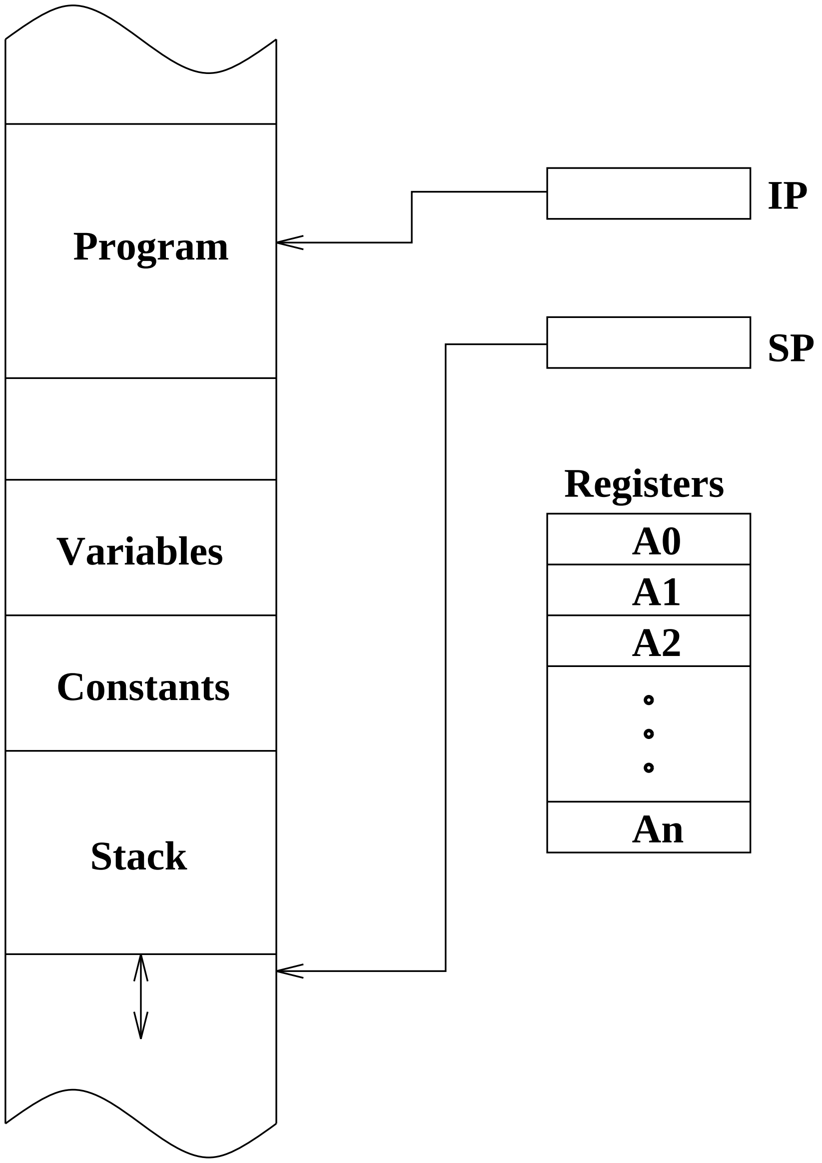


Figure 1: Memory mapping and registers of the virtual processor.

S-expression:

(- 1 (* (* (SIN X) (SIN X)) 2))

Corresponding program:

```
PUSH    ID_CONST_'1'
PUSH    ID_VAR_'X'
SIN
PUSH    ID_VAR_'X'
SIN
*
PUSH    ID_CONST_'2'
*
-
STOP
```

Figure 2: An S-expression and its postfix program.

- a `STOP` instruction which indicates the end of the program.

As we will argue in the next section, it is more effective to precompile prefix GP expressions into an equivalent postfix program which can be interpreted by the virtual machine. This postfix program is generated by traversing the tree representing the S-expression. Two program examples resulting from such a precompilation are provided in figures 2 and 3. The `IFGOTO` instruction jumps to the label if the result of the test is *FALSE*, otherwise, the execution of the program continues with the next instruction (the first instruction of the *Then* statement).
To reiterate, in our implementation, each parallel element is running a different genetic program. The parallel interpreter of the SIMD machine reads the current postfix instruction for each virtual processor and sequentially multiplexes each instruction, *i.e*, all processors for which the current instruction is a `PUSH` become active and the instruction is performed; other processors are inactive (*idle* state). Then, the same operation is performed for each of the other instructions in the instruction set in turn. Once a `STOP` instruction is executed for a processor, that processor becomes idle, leaving the result of its evaluation on the top of the stack. When all processors have reached their `STOP` instruction, the parallel evaluation of the entire population is complete.
Perkis ([16]) has already shown that the stack-based approach for Genetic Programming can be very efficient. However, in his approach, recombination can generate incorrect programs in the sense that it is unknown whether there are enough elements in the stack to satisfy the arity of a function at execution time. A constraint

S-expression:

(IF (< X 1) 1 (* X X))

Corresponding program:

```
           PUSH    ID_VAR_'X'
           PUSH    ID_CONST_'1'
           <
           IFGOTO  Label_1
           PUSH    ID_CONST_'1'
           GOTO    Label_2
Label_1:   PUSH    ID_VAR_'X'
           PUSH    ID_VAR_'X'
           *
Label_2:   STOP
```

Figure 3: An S-expression and its postfix program. If the test returns *FALSE*, the Instruction Pointer jumps to **Label_1** and the *Else* statement is executed.

was implemented to protect the stack from underflow.
In our implementation, since the postfix program is the precompilation of a S-expression, it is always correct and one doesn't have to deal with stack underflow. Moreover, the stack is protected from overflow by restricting the depth of S-expressions resulting from recombination, as described in [11].

### 17.2.2 Parallel Precompiler and Interpreter

For many GP problems the fitness of an expression is computed by evaluating it across a variety of inputs. For example, in curve-fitting, or decision tasks, or sorting networks, the expression must be evaluted multiple times on different data in order to be judged as to its fitness. This leads to the idea of using a data-parallel approach where the same expression is simply evaluated with different data in parallel ([20]). Another approach to take advantage of this feature is to precompile S-expressions from prefix to postfix. This operation can be executed once, and then the postfix program is evaluated multiple times, amortizing the small cost of the precompilation.
The tree traversal algorithm which is the main component of precompilation can be performed efficiently in parallel by simulating on each processor a similar abstract stack machine. In memory, a S-expression is represented by the list of its atoms, without the parentheses. As long as we use a fixed arity (number of arguments)

for each primitive, and the S-expressions are syntactically valid, this string contains enough information to fully represent the given tree. In the current implementation, each atom is coded on 1 byte. The most significant bit indicates whether the atom is an operator or a terminal and the remaining 7 bits represent its ID. For terminals (variables or constants), the ID is an index in the variables/constants area of the memory mapping. The preorder tree traversal is performed simply by reading sequentially the list where the S-expression is stored. Then, using a stack, the postfix program is generated by the algorithm presented in figure 4.
In order to be readable, this algorithm doesn't present the processing of the `IF` operator. To process this operator, another stack is required to store the location of labels whose address calculation is delayed. When the instruction at the top of the stack is a `IF`, the end of the *Then* statement is tested in order to insert a `GOTO` instruction and to jump after the *Else* statement. The label of the `IFGOTO` instruction is calculated at the end of the *Then* statement and the label of the `GOTO` instruction is calculated at the end of the *Else* statement. The result is a program like the one presented in figure 3.

### 17.2.3 Principal Sources of Overhead

There are three main sources of overhead in our parallel model for GP. The first one is intrinsic to the SIMD architecture itself: different instructions cannot be executed at the same time on different processors. In our model, this overhead is directly related to the size of the instruction set interpreted by the virtual processor, which is a few instructions more than the primitive function set for a given task. The second source of overhead comes from the range of S-expression sizes across the population. The third one comes from duplicated operation from one generation to the next one (*e.g.*, the re-evaluation of an unchanged individual with the same test cases).
For the first source of overhead, due to the SIMD simulation, it is possible to use simultaneous table lookup operations to reduce the actual size of the instruction set. For example, if the domain of the primitives for the problem is finite and small, *e.g.* bits or bytes, all arithmetic and logical operations with the same arity can be performed at the same time, without multiplexing. Figure 5 presents a program that evaluates a boolean expression using several different functions (And, Or, XOR, ...) but only 2 actual instructions: `TBL_LK_1D` and `TBL_LK_2D`, which pop 1 and 2 arguments from the stack, respectively, execute the table lookup in the table whose ID is provided as a parameter, and push the result on the top of the stack. Without the table lookup feature, many more problem specific instructions would have been required. Besides simple boolean functions, we expect that simultaneous table lookup will have applications in other symbolic problems.
For the second source, variance in program size, several techniques apply. The

```
     program precompile (in: s_expression,
                             out: postfix_prog);
     begin
       do begin
(1)      read next atom a of s_expression;
(2)      if a is an operator then begin
(3)        stack_item.op = a;
(4)        stack_item.counter = 0;
(5)        push(stack_item);
         end
         else begin
(6)        output(postfix_prog, "PUSH");
(7)        output(postfix_prog, a);

           do begin
(8)          pop(stack_item);
(9)          stack_item.counter = stack_item.counter + 1;
(10)         if arity(stack_item.op) = stack_item.counter then begin
(11)           output(postfix_prog, stack_item.op);
             end
             else begin
(12)           push(stack_item);
             end;
(13)       until (arity(stack_item.op) <> stack_item.counter) or stack is empty;
         end;
(14)   until stack is empty;
     end;
```

Figure 4: Precompiler algorithm.

S-expression:

(AND (OR X Y) (NOT (XOR X Y)))

Corresponding program:

```
PUSH        ID_VAR_'X'
PUSH        ID_VAR_'Y'
TBL_LK_2D   ID_TBL_OR
PUSH        ID_VAR_'X'
PUSH        ID_VAR_'Y'
TBL_LK_2D   ID_TBL_XOR
TBL_LK_1D   ID_TBL_NOT
TBL_LK_2D   ID_TBL_AND
STOP
```

Figure 5: An S-expression and its postfix program, using the table lookup feature.

simplest method involves the management of a sub-population by each processor, with some form of load-balancing. We could also implement a cutoff ([18]) where the largest and slowest population members are simply expunged. Finally, we could use a "generation gap" or generational mixing, where whenever, say, 50% of the population were idle, we could apply reproduction to that subset of the population, crossed with its parents. We would continue to evaluate the larger programs while beginning to evaluate the new members.

The third source, duplication of effort, can be minimized by using an appropriate strategy to manage the evolution of the population, using ideas from steady-state GA's and caching fitness. Such a technique is proposed in section 17.3.3 where the fitness is evaluated only for new individuals.

### 17.2.4 Population Evolution

The previous subsections presented the kernel of our parallel GP implementation. The main part is the parallel evaluation of different S-expressions. The evolution of the population is then managed according to the classical GA framework sketched by the algorithm in figure 6.

In the current implementation, recombination operations are performed on infix S-expressions and not on the postfix. Obviously, crossover can be performed directly on two postfix representations, but is not clear yet how to do this effectively in SIMD style, especially with changing branch labels and distances.

**begin_in_parallel**
/* Generate initial population*/
$random\_generate(s\_expression)$;

**do**
$precompile(s\_expression,\ postfix\_prog)$;
$evaluate\_fitness(postfix\_prog)$;
$selection()$;
$recombination()$;
**until** stop condition is achieved;
**end_in_parallel**;

Figure 6: Population evolution.

## 17.3 Models for Fitness Evaluation, Selection and Recombination

The MasPar MP-2 is a 2-dimensional wrap-around mesh architecture. In our implementation, the population has been modeled according to this architecture: an individual or a sub-population is assigned to each node of the mesh and, therefore, has 4 neighbors. This architecture allows us to implement different models for fitness evaluation, selection and recombination, using the kernel of the parallel GP described in the previous subsections.
We have used 3 different approaches. First, we discuss an approximation of the canonical GP, then a tournament style of co-evolution, and finally, a model of sub-population evolution and migration.

### 17.3.1 Implementation of Canonical GP

Taking fitness definition from [11], the *raw fitness*, the *standardized fitness* and the *adjusted fitness* can be computed independently by each processor. Then, the computation of the *normalized fitness* requires a *reduce* step to sum over all the individuals the adjusted fitness and a *broadcast* step to provide the result of this global sum to each processor. These two parallel operations require $O(\log n)$ time, where $n$ is the number of processors.
Using normalized fitness, we implemented both an asexual and a sexual reproduction system, where each member reproduces on average according to its fitness.
Given an asexual reproduction rate, say 0.2, 20% of the individuals will replace themselves with an individual selected using fitness-proportionate probability from a specified local neighborhood. We chose this local neighborhood, including self, of size $N_{loc} = 15 \times 15 = 225$ as a compromise between getting a correct approximation

of the roulette wheel method and the memory and communication cost of the SIMD machine.
The sexual reproduction, or *crossover* operation for GP, described in detail in [11], which involves cutting and splicing between two S-expressions, is performed in the following way in our implementation:

- the 80% of individuals which have not been asexually replaced select two individuals in their local neighborhood (including self), according to fitness-proportionate probability (the same rule as for asexual reproduction).
- Crossover is performed for these two parents.
- One of the two offsprings is arbitrarily chosen to replace this individual.

This last operation is different from the basic GP which keeps both offsprings. However, our approach is more SIMD oriented, yet doesn't introduce any bias in the search since the new offsprings are still produced accordingly to the distribution of the fitness among individuals of the population. Moreover, this slight difference can be eliminated if each processor is in charge of a sub-population of individuals.
The time complexity of the crossover and asexual reproduction system is $O(N_{loc})$ and its space complexity is $O(\sqrt{N_{loc}})$ for each processor. The crossover operation is performed on a string representation of the S-expressions in parallel using another variant of our stack machine.

### 17.3.2 Tournament Fitness

Our second approach to fitness follows the co-evolution paradigm, e.g. Angeline and Pollack ([4]). There is no absolute fitness measurement for an individual, fitness is determined by competition in tournament with the other individuals in the population at the current generation. As the individuals in the population improve, survival gets more difficult.
To evaluate the fitness of each individual in the population, a tournament has been organized in the following way:

- First, we did not use single-elimination because it is not an effective use of SIMD. In order to keep using all the processors to refine our relative fitness estimate, winners at a round will meet in the same pool at the next round and losers will compete in another pool.
- At the end of the tournament, each individual's fitness is calculated from its total number of wins and draws across matches.

For each round of the tournament, all the processors are paired according to the divide-and-conquer communication pattern (such a pattern is presented in figure 7, in the case of 8 processors) and perform the following operations:

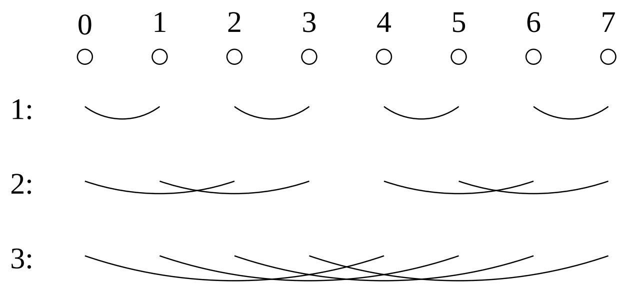


Figure 7: Divide-and-conquer communication pattern.

- the program of the other paired processor is copied into their own memory,
- a match is played for which the local program is the first to move. As a result, each individual plays two matches: it is the first to move for one of them and it is the second to move for the other match.
- the result of the two matches is analyzed by one processor from each pair. The program that gets the larger score is assigned to the left processor and the second one is assigned to the right one (randomly in case of draw). This way, using divide and conquer, winners will meet each other in the next round, and more information will be gathered for strategy evaluation, while the same $\log n$ number of tournament steps are performed.

At the end of the tournament, it is straightforward to collect total score (or fitness) for each individual.

### 17.3.3 Sub Populations with Migration

The idea of this implementation is to study a model of sub-populations that interact locally one with each other, similar to the model presented by Ackley and Littman in [1] and [2].
In our experiments, each processor manages a sub-population of 16 individuals. A table in which is stored the result of the competition between all possible pairs of individuals in the sub-population is maintained by each processor. At each generation, 2 successive operations are performed by each processor:

- a selection/reproduction round: 2 parents in the sub-population are selected according to a fitness-proportionate probability and are crossed. The resulting offspring replaces one of the less fit individuals (using an inverse fitness-proportionate probability rule).

Table 1: Results and time performance.

| Problems: | Discovery of Trigonometric Identities (section 10.1 from [11]) | Symbolic Integration (section 10.5 from [11]) |
|---|---|---|
| Objective function | $\cos(2x)$ | $\cos x + 2x + 1$ |
| Number of runs | 10 | 10 |
| Number of Generations | 5 to 29 gen. (average: 17.5) | 4 to 7 gen. (average: 5.6) |
| Execution time (for one run) | 7.24 to 50.13 seconds (average: 30.48 sec.) | 23.09 to 40.38 seconds (average: 32.31 sec.) |
| Average execution time for 1 generation | 1.75 sec. | 5.75 sec. |

- a migration round: an individual is selected uniformly randomly in each sub-population and all those individuals migrate in the same direction to one of the neighboring sub-population.

Therefore, only the results of matches against the 2 new individuals have to be updated in the table.

## 17.4 Results and Performance

We have explored the use of MPGP on a few problems to date, Symbolic integration, Tic-tac-toe, and the Intertwined Spirals problem.

### 17.4.1 Canonical GP

We performed our first canonical GP experiments with a population of 4096 individuals, one per processor. The two problems are from Chapter 10 of [11], and involve repetitive testing of expression against a range of data. Table 1 presents results and performance, using the same primitives and parameter specifications (except for the population size) as Koza. We were able to achieve the evaluation of about $2,350$ S-expressions in 1 second (on average) for the discovery of trigonometric identities and the evaluation of about 710 S-expressions in 1 second (on average) for the symbolic integration problem.

### 17.4.2 Tic-Tac-Toe

We have replicated Angeline and Pollack's (1993) model of the co-evolution of Tic-Tac-Toe players, although we have not yet implemented Modular subroutines. No "expert" player was used to evaluate the fitness of the different individuals, but more and more effective strategies appeared as a result of this relative fitness co-evolution, and the resultant player (after 200 generations) was stronger than players evolved using absolute fitness optimal and heuristic strategies.

In their experiments, Angeline and Pollack used a population of 1000 individuals and each run was about 200 generations in single-elimination. In ours, we used 4096 players in a tournament as described above, for 200 generations, in which each player plays 12 pairs of games. In each game, 2 points are assigned for a win, 1 point for a draw and 0 for a defeat, and the sum over the 24 games is its fitness.

In our first experiments, we observed the similar results and dynamics as Angeline and Pollack, and didn't achieve a "perfect" player. In these timings, the size of S-expressions was limited either to 256, 512 or 1024 atoms, and a maximal depth of 50. Table 2 presents the execution time for one generation in the case of the "global tournament" model, once the size of the largest S-expressions reached the upper limit. We were able to achieve up to $8,192$ games in one second (with a maximum of 256 atoms) on our $4K$ processors MasPar. This performance has been achieved using the table look-up feature presented in section 17.2.3.

Ultimately, one should expect the emergence of a perfect player (a player that could only win or draw). We fully tested the best of each generation off-line, and have not yet achieved a "perfect" GP TTT player. Such a result has been achieved in 3 million games by Rosin and Belew ([17]), where TTT strategies were represented as a table lookup, and only legal moves were considered. In our more general GP representation, individuals have to learn the game rules, *i.e.*, they have to evolve a strategy that prevent them from playing in a position which is already occupied (for example), and how to play and block effectively. As a result, the size of the search space is considerably larger.

In order to see if it is a matter of scale, we used our sub-population model with 16 individuals on each processor, for a population size of $64K$. For the sub-population model, time performance and results are similar to the ones we got with the tournament model. Furthermore, for a very long run (more than $3,000$ generations), we generated an individual player that *cannot lose when playing first*. This let us think that the emergence of a perfect TTT player using the GP approach and coevolution should be possible.

### 17.4.3 Intertwined Spirals

As another benchmark, we also performed some experiments comparing canonical GP evolution vs co-evolution for the intertwined spiral problem. This learning

Table 2: Time performance for one generation for the co-evolution of Tic-Tac-Toe players.

| Maximum number of atoms | 256 | 512 | 1024 |
|---|---|---|---|
| Execution time for one generation (on average) | 6 sec. | 10 sec. | 18 sec. |
| Total number of games per second (on average) | 8192 | 4915 | 2730 |

problem, originated by Alexis Wieland, perhaps based on the cover of Perceptrons, has been a challenge for pattern classification algorithms, and has been the subject of much work in the Neural Network field (e.g. [14, 7, 6]). It consists of classifying points into two classes according to two intertwined spirals. The data set is composed of two sets of of 97 points, on the plane between -7 and +7.
Koza ([11]) and Angeline's chapter ([5]) also investigate this problem using the Genetic Programming paradigm. Basically, we used the same form as them to define the problem and to perform our experiments. That is, the function set is composed of: $\{+, -, *, \%, iflte, sin, cos\}$, and the terminal set is composed of: $\{x, y, \Re\}$, where $\Re$ is the ephemeral random constant. Because we are using byte-coded instructions, our ephemeral constants are selected from a finite set.
With a population of 4096 individuals, we tried two different approaches to tackle this problem. In the first experiment, following Koza and Angeline, the fitness function was defined as the number of hits out of 194.
In the second experiment, the fitness was defined as the result of a tournament competition among the individuals. We ignored the fact that we really know the absolute fitness function, and set up a "game" in which only relative fitness was used as the basis for reproduction. In a classification game between two players, the score was the number of unique hits (those which the other player didn't also get). The final fitness of each individual is the sum of all its scores during the competition. In order to make each individual meet a significant number of opponent, 8 successive tournaments were performed at each generation. Thus, each individual met 96 opponents (there are 12 rounds in a tournament with a population size of 4096). Each tournament was organized according to the divide-and-conquer communication pattern described in section 17.3.2. Moreover, since one doesn't need to determine a winner at each round of the tournament (only the individuals' score is used), a winner was selected randomly, enabling a different individual pairing for

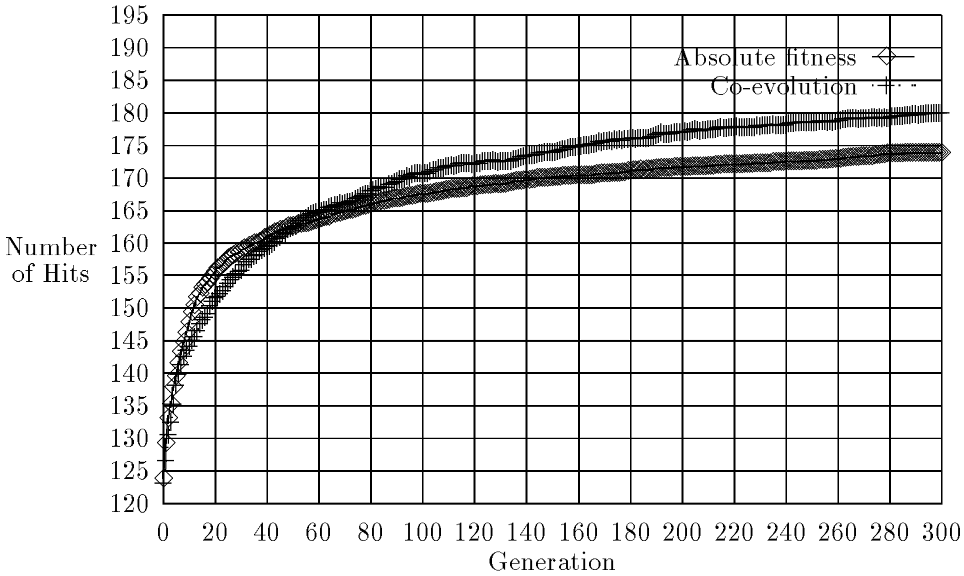


Figure 8: Absolute fitness versus co-evolution for the intertwined spiral problem.

the next tournaments.

For the two classes of experiments, we performed 25 runs and each run was stopped after 300 generations. Because of the use of floating point, each generation took about a minute.

Our hypothesis is that the co-evolution would work better because it would promote more diversity in the population, and allow subpopulations which covered different subproblems to emerge. As copies of individuals which perform well on parts of the spiral spread through the population, they will start to meet themselves in competition, and get a score of 0. This allows other individuals who may have less total hits, but cover other parts of the spiral to survive. Our preliminary results concerning performance, shown in figure 8, illustrate that co-evolution seems to outperform the absolute fitness approach. However, the large number of parameters that control the dynamics of the system doesn't allow us to conclude.

Only one run provided us with a perfect solution for the intertwined spiral problem. This was one of the co-evolution runs. We harvested some of the perfect classification solutions; One of the shortest of these S-expressions has 52 atoms and is shown in figure 9.

Because of the relatively small size of this result we were able to analyze it and simplify it mathematically, by collapsing constant calculations, removing insignifi-

```
(sin (% (iflte (- (- (- (* _A _A)
                        (sin (% (iflte  -0.52381
                                        _B
                                        (sin -0.33333)
                                        -0.33333)
                                -0.33333)))
                     (* _B _B))
                  (% _A (% -0.33333 _A)))
               -0.80952
               _B
               (sin (% (% _A
                          (- (cos (sin (* (cos (sin -0.52381))
                                          (% _B
                                             (% _A
                                                (- (cos -0.33333)
                                                   0.04762))))))
                             0.04762))
                       (sin (sin -0.33333)))))
        -0.33333))
```

Figure 9: A 52-atom S-expression scoring 194 for the intertwined spiral problem.

**If** $(4 * x^2 - y^2) \; < \; 0.0$ **then**
  **return** $(sin(-3.0 * y))$;
**else**
  **return** $\left( sin(\frac{0.3214 * x}{0.04762 - cos(sin(\frac{y}{x} * 0.7874))}) \right)$;
**endif**

Figure 10: Interpretation of the solution for the intertwined spiral problem.

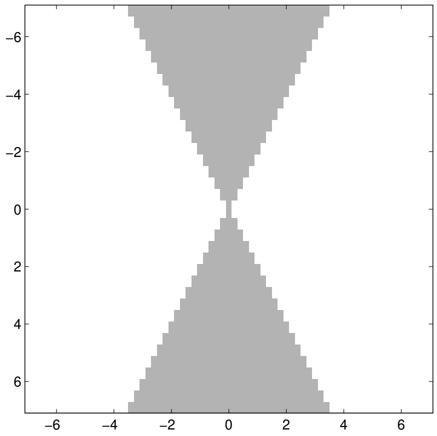


Figure 11: $4x^2 - y^2 < 0$, used to divide the plane into two domains.

cant digits, algebraic simplification, and elimination of redundant "introns". This analysis resulted in the conditional function presented in figure 10.

Basically, this solution splits the geometric plane into two domains and a different function is used for each domain. Figure 11 displays the $4x^2 - y^2$ function which multiplexes the two other functions to create the spiral, shown in figure 12.

The resulting function is shown in figure 13, which plots the function (above/below 0) along with the training data on the range -10 to 10. Although it does not form a perfect spiral, it does continue to simulate a spiral way outside the original training range. Furthermore, we believe that compared to neural network solutions, which are often the composition of hundreds of clusters or decision boundaries, and some of the GP solutions shown by Koza, ours is the most perspicacious to date. The fact that the spiral is composed of a synergy of two functions which cover separate parts of the data supports the hypothesis that the relative fitness co-evolution strategy may be more effective than an absolute fitness function.

Finally, a few remarks could be made about the difficulty of this problem and the limitations of our massively parallel implementation in this case. In his experiments, Koza used a population of 10,000 individuals and the over-selection mechanism. In our case, the population size is 4096 and efficient implementation of over-selection is not really compatible with the geographical distribution of processors in the mesh architecture. Indeed, we believe that to get an optimal solution, one needs a "good" individual in the earliest generations that will be an interesting "seed" for the following generations. According to our canonical GP experiments on the spiral problem, the convergence takes about 100 generations and the best individual is then very difficult to improve. In fact, for some experiments that led to worst solutions, the population even converged only after 50 generations. A large population and

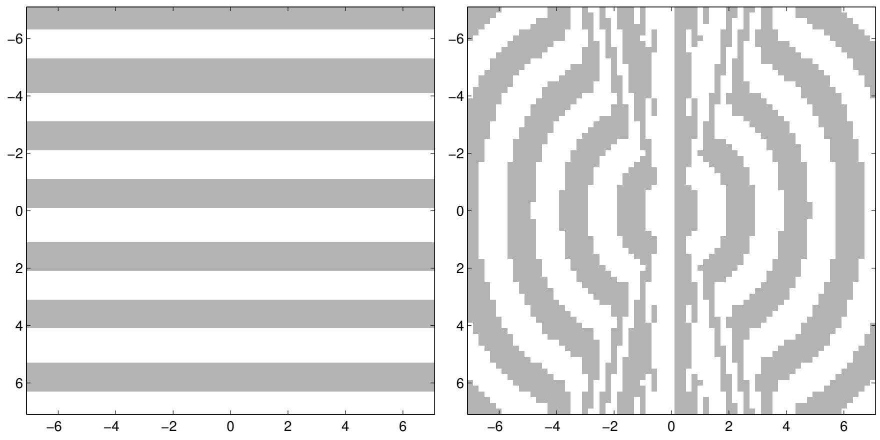


Figure 12: $sin(-3y)$ and the other function which are selectively added to make a spiral.

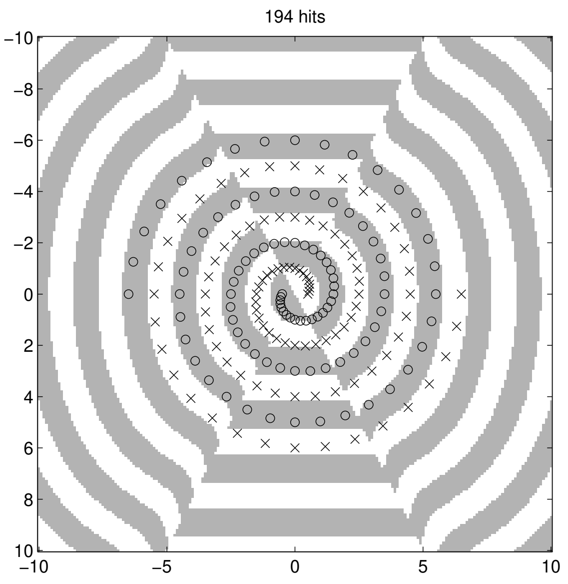


Figure 13: Perfect score generalizing classification of the two intertwined spirals.

over-selection may have helped Koza to remedy this problem.

## 17.5 Conclusion

This chapter described an implementation of parallel Genetic Programming on a SIMD computer and showed its efficiency on a few representative problems. Despite the fact that there is overhead in multiplexing basic operations, and in precompiling prefix expressions to postfix programs, we were able to achieve quite an efficient parallel GP engine.

The initial goal of this project was to exploit the huge peak performance of our SIMD computer (17 Gips for a 4K processor MP-2) for evolutionary learning research applications. With 4k processors, even utilitizing 1/10th of the capacity of this machine would be more productive than running over a small group of workstations. We were surprised that our first experimental results showed that this goal could be easily achieved at the condition that the virtual processor's instruction set can be kept small, the performance being directly (linearly) related to the size of this set. We have also seen that while expression evaluation involves a lot of overhead, reproduction and crossover have effective massively parallel models ([9, 10, 19]).

This technique has also a few drawbacks: In particular, implementation of high-level features like modular subprograms or automatically defined functions ([3, 12]) are not as easy to implement as on a flexible MIMD architecture. We believe a simple addition like a `CALL` instruction in conjunction with a return stack might work for modular form.

We believe that this technique is very promising and even more impressive results can be achieved for problems in which the function set can be specified in the same instruction set as our overall model. Indeed, in that case, it may be possible to overlap execution of the primitive functions using table look-up techniques.

There is still a lot of work to do, but we have shown that our SIMD approach to massively parallel Genetic Programming is both plausible and efficient.

# References


[1] David H. Ackley and Michael L. Littman. A Case for Lamarckian Evolution. In *Artificial Life III*, Ed. Christopher G. Langton, Addison-Wesley, 1994.

[2] David H. Ackley and Michael L. Littman. Altruism in the Evolution of Communication. In *Artificial Life IV*, Brooks and Maes, Eds. MIT Press, 1994, pp. 40-48.

[3] Peter J. Angeline and Jordan B. Pollack. The Evolutionary Induction of Subroutines. In *The Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington Indiana, 1992.

[4] Peter J. Angeline and Jordan B. Pollack. Competitive Environments Evolve Better Solutions for Complex Tasks. In *The Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, 1993, pp. 264-270.

[5] Peter J. Angeline. Two Self-Adaptive Crossover Operations for Genetic Programming. In this book.

[6] Gail Carpenter, Stephen Grossberg, Natalya Markuzon, John Reynolds, and David Rosen. Fuzzy ARTMAP: A Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps. In *IEEE Transactions on Neural Networks*, Vol. 3, No. 5, 1992, pp. 698-713.

[7] Scott E. Fahlman and Christian Lebiere. The Cascade-Correlation Learning Architecture. In *Advances in Neural Information Processing Systems 2*, Touretzky, Ed. Morgan Kauffman, 1990.

[8] W. Daniel Hillis and Guy L. Steele Jr. Data Parallel Algorithms. In *IEEE Computers*, 29, 1986, pp.1170-1183.

[9] W. Daniel Hillis. Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure. In *Artificial Life II*, Langton, et al, Eds. Addison Wesley, 1992, pp. 313-324.

[10] David Jefferson, Robert Collins, Claus Cooper, Michael Dyer, Margot Flowers, Richard Korf, Charles Taylor, and Alan Wang. Evolution as a Theme in Artificial Life: The Genesys/Tracker System. In *Artificial Life II*, Langton, et al, Eds. Addison Wesley, 1992, pp. 549-578.

[11] John R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.

[12] John R. Koza. Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, 1994.

[13] John R. Koza and David André. Parallel Genetic Programming on a Network of Transputers. This Volume.

[14] Kevin J. Lang and Michael J. Witbrock. Learning to tell two spirals apart. In *Proceedings of the 1988 Connectionist Summer Schools*, Morgan Kaufmann.

[15] Michael S. Littman and Christopher D. Metcalf. An Exploration of Asynchronous Data-Parallelism. Personal communication. 1990.

[16] Timothy Perkis. Stack-Based Genetic Programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press.

[17] Christopher D. Rosin and Richard K. Belew. Methods for Competitive Co-evolution: Finding Opponents Worth Beating. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, 1995, pp. 373-380.

[18] Karl Sims. Evolving 3D Morphology and Behavior by Competition. In *Artificial Life IV*, Brooks and Maes, Eds. MIT Press, 1994, pp. 28-39.

[19] Reiko Tanese. Distributed Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, pp. 434-439.

[20] Patrick Tufts. Parallel Case Evaluation for Genetic Programming. In *1993 Lectures in Complex Systems*, Eds. L. Nadel and D. Stein, SFI Studies in the Sciences of Complexity, Lec. Vol. VI, Addison-Wesley, 1995, pp.591-596.