Parallel Genetic Programming on Fine-Grained SIMD Architectures

Hugues Juillé and **Jordan B. Pollack**
Computer Science Department
Volen Center for Complex Systems
Brandeis University
Waltham, MA 02254-9110
<{hugues, pollack}@cs.brandeis.edu>

Abstract

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 recently presented by Koza ([8]) 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 ([16]), 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 paper 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.

1 Introduction

The idea of simulating a MIMD machine using a SIMD architecture is not new ([11]). One of the original ideas for the Connection Machine ([5]) 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 branching*, an *unconditional branching* and a *stop* instruction, we can get a very effective MIMD simulation running.

This paper 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.

2 Description of the implementation

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

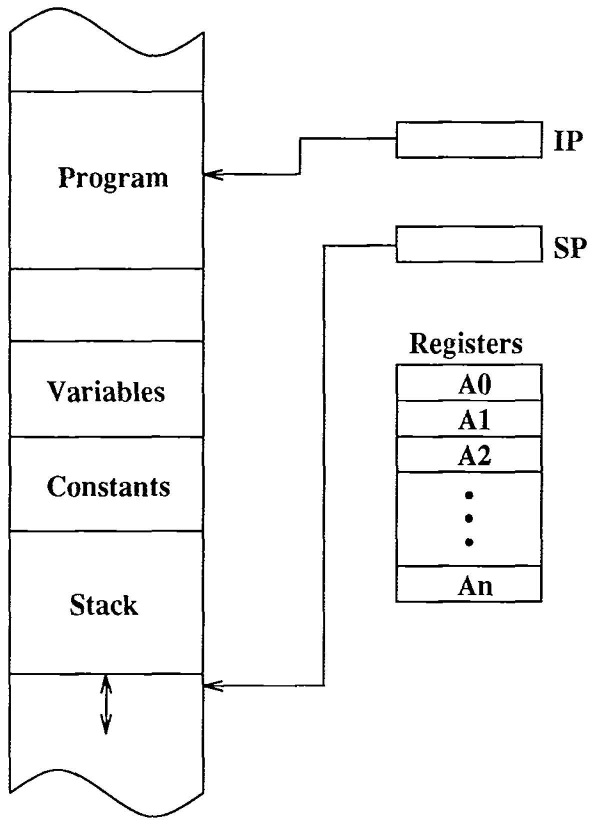

Figure 1: Memory mapping and registers of the virtual processor.

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 registers, the function is computed, and the result is pushed on the top of the stack.

<u>S-expression:</u>

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

<u>Corresponding program:</u>

```
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.

<u>S-expression:</u>

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

<u>Corresponding program:</u>

```
          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.

- 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,
- 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 ([12]) 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 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 [8].

2.2 Parallel Precompilator 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 ([16]). 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.

```
     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: Precompilator algorithm.

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

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.

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 simplest method involves the management of a sub-population by each processor, with some form of load-balancing. We could also implement a cutoff ([14]) 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 3.3 where the fitness is evaluated only for new individuals.

2.4 Population Evolution

The previous sections 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.

```
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.

In the current implementation, recombination operations are performed on S-expressions and not on the postfix program. This explains why S-expressions are precompiled at the beginning of each generation. We are currently evaluating whether crossover can be fully integrated with precompilation, so that we can cross over on postfix programs with embedded conditional clauses. One of the major problem is to deal effectively with label updating.

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 sections.

In the next section, an approximation of the canonical GP implemented around our architecture is presented. Then, we will describe an example of co-evolution where individuals fitness is the result of a tournament. Finally, an example of an implementation involving local interactions between sub-populations will be presented.

3.1 Implementation of Canonical GP

Taking fitness definition from [8], 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 [8], 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 subpopulation 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.

This model has been tested on 2 problems from [8]. Results are presented in section 4.

3.2 Fitness Evaluation with Tournament

The aim of this implementation is to reproduce some experiments of Angeline and Pollack ([4]). We have not yet implemented modular subroutines. In their experiments, a population of Tic-Tac-Toe (TTT) players co-evolved. 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 competitive co-evolution: each time an individual evolves a new move that would defeat most of the individuals, the emergence of new individuals that would be able to counter this move is facilitated. Ultimately, one could expect the emergence of a perfect player (a player that could only win or draw). Such a result has been achieved by Rosin and Belew ([13]), where TTT strategies were represented as a table lookup. However, the representation of strategies as a S-expression is more general and attractive.Unfortunately, the emergence of a "perfect" GP player hasn't been achieved yet.

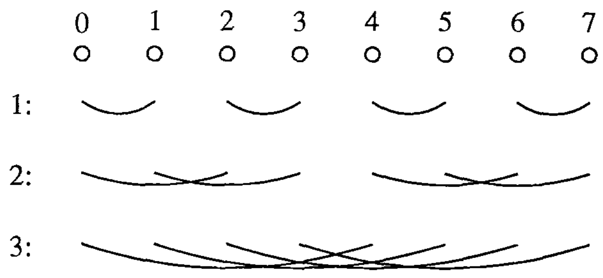

Figure 7: Divide-and-conquer communication pattern.

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 as in [4] because this is not an effective use of SIMD. In order to keep using all the processors to refine our 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:

- 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: 2 points are assigned for a win, 1 point for a draw and 0 for a defeat. 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. Some results of our experiments are presented in section 4.

Table 1: Results and time performance.

Problems:	Discovery of Trigonometric Identities (section 10.1 from [8])	Symbolic Integration (section 10.5 from [8])
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.

3.3 Population Evolution with Local Interactions

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]. This model has also been tested with the Tic-Tac-Toe problem.

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).
- 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.

4 Results and Performance

We performed our first experiments with a population of 4096 individuals. Table 1 presents results and performance on two problems from [8], using the same specifications (except for the population size). 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.

We also reproduced experiments presented in [4] with this same population size, using the model of "global tournament" presented in section 3.2, or with a population size of $64K$, using the model of sub-populations interacting locally described in section 3.3. In those experiments, 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. For this model, each individual plays 24 matches at each generation, being the first player for half of them, and the second for the other half. 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 2.3.

In their experiments, Angeline and Pollack used a population of 1000 individuals and each run was about 200 generations. In our first experiments, we observed the same results as Angeline and Pollack, still not achieving a perfect player.

For the sub-population model, time performance and results are similar to the ones we got with the "global tournament" model. For a very long run (more than 3,000 generations), the emergence of an individual that *cannot lose when playing first* has eventually been observed. This let us think that the emergence of a perfect player using the GP approach and coevolution should be possible. Rosin and Belew ([13]) managed to evolve a perfect strategy for Tic-Tac-Toe after a simulated coevolution for which about 3 million games were played. However, their genotype representation is such that only legal moves can be considered. In the more general representation proposed by Angeline and Pollack, 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). As a result, the size of the search space is considerably larger.

5 Conclusion

This paper described an implementation of parallel Genetic Programming on a SIMD computer and showed its efficiency on a few representative problems. Despite the

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

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 ([6; 7; 15]).

This technique has also a few drawbacks: In particular, implementation of high-level features like modular subprograms ([3; 9]) may require a great deal of effort. Although, it is possible that a simple trick like a CALL instruction and a return stack would also work.

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

[6] 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.

[7] 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.

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

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

[10] John R. Koza and David André. Parallel Genetic Programming on a Network of Transputers. Technical report. Stanford University. 1995.

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

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

[13] 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.

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

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

[16] 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.