Comparison of Parallel Linear Genetic Programming Implementations

David Grochol and Lukas Sekanina

Brno University of Technology, Faculty of Information Technology
IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, Czech Republic
igrochol@fit.vutbr.cz, sekanina@fit.vutbr.cz

Abstract. Linear genetic programming (LGP) represents candidate programs as sequences of instructions for a register machine. In order to accelerate the evaluation time of candidate programs and reduce the overall time of evolution, we propose various parallel implementations of LGP suitable for the current multi-core processors. The implementations are based on a parallel evaluation of candidate programs and the island model of the parallel evolutionary algorithm in which the subpopulations are evolved independently, but some genetic material can be exchanged by means of the migration. Proposed implementations are evaluated using three symbolic regression problems and a hash function design problem.

1 Introduction

Linear genetic programming (LGP) is a form of genetic programming in which candidate programs are encoded as sequences of instructions and executed on a register machine. LGP is especially useful for designing relatively short but well-tuned programs simultaneously showing an excellent quality of processing and implementation effectiveness. Examples of evolved programs include hash functions, game strategies, communication protocols, and low-level machine code routines. If the LGP implementation is fast it can autonomously provide the optimized programs to such systems in which the specification is modified in the runtime.

The performance of LGP primarily depends on the problem encoding, search operators, fitness evaluation and efficiency of the implementation. In order to accelerate the evaluation time of candidate programs and reduce the overall time of evolution, we propose parallel implementations of LGP suitable for the current multi-core processors. Contrasted to the research dealing with efficient genetic operators and search methods [13, 4], our work if focused on an efficient distribution of workload (represented mainly by the fitness evaluation as the most time consuming component of LGP) among the available computing resources.

The goal of this paper is to develop parallel implementations of LGP and compare their performance in terms of throughput measured as the number of candidate solutions evaluated per second (primary objective) and the quality of

evolved programs (secondary objective). The implementations are based on the parallel evaluation of candidate programs and the island model of the parallel evolutionary algorithm in which subpopulations are evolved independently, but some genetic material can be exchanged by means of the migration. Proposed implementations are evaluated using three symbolic regression problems and a hash function design problem.

The rest of the paper is organized as follows. Section 2 surveys the relevant work. The proposed parallel LGP methods are presented in Section 3. Experimental setup and benchmark problems are introduced in Section 4. Results are reported in Section 5. Conclusions are given in Section 6.

2 Related work

Genetic programming (GP) is an artificial intelligence method capable of automated design of programs in a given programming language [9]. There are several branches of GP such as tree-based GP, linear GP and Cartesian GP. They primarily differ in the representation of candidate programs. This section introduces the field of genetic programing and provides relevant details about LGP and its parallelization.

2.1 Linear genetic programming

Linear genetic programming [1, 13, 18] uses a linear representation of computer programs. Every program is composed of operations, which are called instructions, and operands, which are stored in registers.

Program representation An instruction is typically represented by an instruction code, destination register and two source registers, for example [+, r0, r1, r2] denotes $r0 = r1 + r2$. The input data are stored in pre-defined registers or an external memory. The result is returned in a given register. The number of instructions in a candidate program is variable, but the minimal and maximal values are defined. The number of registers available in a register machine is constant. The function set known from GP corresponds with the set of available instructions. The instructions are general-purpose (e.g. addition and multiplication) or domain-specific (e.g. read sensor 1). Conditional and branch instructions are important for solving general problems. Protected versions of instructions (e.g. a division returns a value even if the divisor is zero) are employed in order to execute all programs without exceptions such as the division by zero.

Genetic operations LGP is usually used with a tournament selection, one-point or two-point crossover and a mutation operator modifying either the instruction type or register index. Advanced genetic operators have been proposed, see for example, [4].

Fitness function The most computationally expensive part of LGP is the fitness function evaluation, in which a candidate program is executed for a set of training inputs, the program's outputs are collected and the fitness value is determined.

There are essentially two types of linear GP: a machine code GP, where each instruction is directly executable by the CPU, and an interpreted linear GP, where each instruction is executable by a virtual machine (simulator) implemented for a given processor.

2.2 Parallel GP

This section provides a brief overview of approaches developed to parallelize genetic programming [2, 5, 15]. The approaches differ in the algorithms and hardware employed.

The farmer-model, or the master-slave model, operates with a global population of programs. The master processor performs all genetic operations and assigns the candidate solutions to remaining processors (slaves) to be evaluated [14]. One processor typically evaluates a subpopulation of candidate solutions.

In the island model [16], independent populations are evolved concurrently and separately. In every kth generation, each population sends its best individuals, according to a pre-designed communication pattern and network topology, to other populations. Synchronous and asynchronous versions of this model have been developed.

Modern CPUs support special vector instructions. If a code vectorization is enabled, CPU provides a restricted type of parallel processing, often referred to as the single instruction multiple data (SIMD) or the single program multiple data (SPMD). Then, a program response can be obtained for several test vectors in parallel.

Parallel GP implementations were also developed for specialized hardware such as graphic processing units (GPU) [2, 5] and general-purpose accelerators (Xeon Phi) [6].

3 Parallel LGP

The most computationally expensive part of LGP is the fitness function evaluation. In this section, two approaches to the fitness evaluation are presented. A method enabling the parallelization of these evaluation approaches is then discussed in Section 3.2. Finally, an island-based parallel LGP is introduced.

3.1 Fitness Evaluation

The first method (Method A, see Algorithm 1) determines the fitness value in a straightforward manner according to a common sequential code for genetic programming. Each candidate program is executed for each vector of the training

Algorithm 1: Fitness: Method A

```
INPUT: Population, Training set
FOR EACH individual of the population DO
  FOR EACH vector of the training set DO
    Initialize registers by the vector
    Sequentially execute instructions of the individual
  Update the fitness value
```

set. In general, conditional branches make difficult to effectively employ the code vectorization because it has to be determined for each instruction whether it will be executed or not. Method A might be suitable for instruction sets containing conditional branches, because it can skip a number of instructions if a conditional branch is present. This leads to the reduction of the execution time of a candidate solution.

The second method (Method B, see Algorithm 2) divides the training set into chunks of training vectors. The chunk size is one of LGP parameters. All individuals are executed for all vectors from a given chunk which can be exploited in the subsequent parallel processing. If candidate programs contain the conditional branches the code vectorization is difficult. This method requires creating an array of registers for each vector form the chunk. The chunk size is determined experimentally. While large chunks can invoke many L2 cache misses, small chunks introduce a significant overhead and make the method inefficient.

Algorithm 2: Fitness: Method B

```
INPUT: Population, Training set, Chunk size s
Divide the training set into chunks according to s
FOR EACH chunk DO
  FOR EACH individual from the population DO
    Initialize registers by the vector from the chunk
    Sequentially execute instructions of the individual
    for all vectors from the chunk
  Compute a partial fitness function for the chunk
At the end compute the final fitness value
```

3.2 Parallel Fitness Evaluation

The parallel evaluation proposed for method A consists in assigning the individuals of the population to processes. As there is no communication among the individuals during the evaluation, the speedup is given by the number of available cores. The same approach is also adopted in method B, but each process evaluates the candidate programs with the subsets of training vectors (chunks).

The parallel fitness evaluation is realized with OpenMP [3] which is a multi-platform shared-memory parallel programming paradigm developed for C/C++ and Fortran. Programs utilizing OpenMP are limited in the number of processes which should be less or equal to the number of logic cores of a given CPU. If the number of processes is higher the processes start to compete for resources and the overall overhead is growing.

3.3 Island Model for Parallel LGP

The proposed implementation utilizes the island-based model with a ring topology. The individuals occupying a given island are evaluated using method B. The communication between the islands is asynchronous. As the evaluation of population(s) on the islands may consume a different time, faster islands do not have to wait for slower islands. After a predefined number of generations, every island sends the best individuals to its neighbors. All islands try to receive some individuals from other islands in every generation. Newly incoming individuals replace randomly chosen individuals of the population, but the best scored individuals are always preserved. The individuals are sent as integer array messages. The implementation is based on MPI [10].

4 Setup and Benchmark Problems

This section defines the experimental setup and benchmark problems utilized for comparison of the proposed LGP implementations.

The LGP code was written in C and compiled with gcc version 4.9.3, with full optimization (O3) and vectorization enabled. All experiments were carried out on a Linux machine equipped with the Intel Xeon E5-2630 processor. There is a limitation of 12 processes in the parallel model.

4.1 Setup

Two scenarios were developed to evaluate LGP implementations. The purpose of the first one is to measure the performance in terms of the number of instructions/generations that can be executed within a given time. LGP parameters are summarized in Table 1. The second scenario is used to evaluate the quality of evolution (i.e. the obtained fitness). This scenario is employed for testing the island model. LGP parameters are given in Table 1. The fitness function for symbolic regression problems is defined as the mean absolute error between the program outputs and desired output.

In both scenarios, the program size is constant which allows for a straightforward comparison of the execution time. In order to investigate the impact of branch instructions, two function sets are defined. Both function sets contain arithmetic operations. However, the second function set ($T2$) contains a branch instruction which introduces, in principle, numerous difficulties during the code vectorization. Evaluation of these candidate solutions is thus slower.

Table 1. LGP parameters for symbolic regression

Parameter	Performance tests	Island model
Population size	1000	1000
Crossover probability	90%	90%
Mutation probability	15%	15%
Program length	40	40
Registers count/type	16/double	16/double
Instruction set T_1	{+, -, *, /}	{+, -, *, /}
Instruction set T_2	{+, -, *, /, IF}	-
Terminal set	{1, indepen. variables}	{1, indepen. variables}
Tournament size	4	4
Maximum number of generations	1000	1000
Crossover type	One-point	One-point
Migration	-	40 generations

4.2 Benchmarks problems

Symbolic regression Three standard symbolic regression problems were taken from [17, 7]. Table 2 gives intervals used for construction of the training sets.

F1:

$$f(x,y) = \frac{x^2+y^2}{2y}$$

F2:

$$f(x) = x^4 + x^3 + x^2 + x$$

F3:

$$f(x,y) = \frac{x^3}{5} + \frac{y^3}{2} - y - x$$

Table 2. Parameters of training sets generated using benchmark functions

	Performance tests			Island model		
Function	Training set size	Range	Step	Training set size	Range	Step
F1	10000	$x, y \in <0, 100)$	1	10000	$x, y \in <0, 100)$	1
F2	10000	$x \in <0, 10000)$	1	100	$x \in <0, 100)$	1
F3	10000	$x, y \in <0, 100)$	1	900	$x, y \in <0, 30)$	1

Hash function design In order to evaluate LGP on a real-world problem, we will employ the hash function design problem. The objective is to find a hash function which maps the data of an arbitrary size to the data of a fixed size. A good hash function should have some important properties, for example, a small input change should invoke a large output change to minimize potential collisions, it has to be deterministic and easy to compute. Detailed description

of the principles of hash functions is available in [8]. A hash function is often used in hash tables. A hash function produces an index to the table from input data. In the case of a collision, several possibilities exist to solve it [12]. One of the approaches is a perfect hashing in which a special hash function is created for a given data set such that it does not produce any collisions for this set [11].

In the area of computer networks, it is often needed to track specific users or devices. They can be identified by IP addresses. However, the number of IP address may vary over time. It makes sense to develop a specialized hash function in order to eliminate disadvantages of universal hash functions that can produce a large amount of collisions for a specific set of IP addresses. It is time expensive and very difficult to create good hash functions manually. Automated methods are sought that can quickly provide a good hash function for monitored traffic. Providing a good hash function in a short time enables to start network monitoring sooner and catch more important data. LGP can be employed to search for the perfect hash function for a given set of IP addresses.

For this study, a set of IP addresses was randomly selected from the firewall in our computer network. Experiments will be performed with two data sets containing 1000 and 5000 IP addresses and a 16 bit hash table. The fitness function computes the number of collisions produced by a candidate hash function on a given training set. The goal of LGP is to minimize the number of collisions.

The instruction set contains operations that can be found in common hash functions (RightRotation, NOT, AND, OR, XOR, +, *). A set of 10 prime numbers used in the initialization phase of the SHA-2 function represent the constants available to LGP. The program length is restricted to 20 instructions as common hash functions are of this size. LGP uses 8 integer registers. Other settings are given in Table 1.

5 Results

Results of experiments are structured into two parts in this section: symbolic regression benchmarks and hash function design.

5.1 Symbolic Regression Benchmarks

In total, LGP was employed to solve 60 specifications which differ in the method (A or method B with 5 different chunk sizes - 500, 1000, 2500, 5000 and 10,000 vectors), function set (T_1, T_2) and the number of cores (1, 2, 4, 6 and 12) used. In order to obtain basic statistics, 20 independent LGP runs were performed for each specification. Each run produced 1000 generations.

In the first experiment, we compared the sequential implementations of both methods. The boxplots shown in Fig. 1 give LGP performance per one generation in MFLOPs (measured by PAPI). The boxplots used in these figures represent the minimum, first quartile, median, third quartile and maximum. The experiments confirmed our assumption that method B provides higher performance than method A. An optimal chunk size seems to be 1000 vectors. The chosen

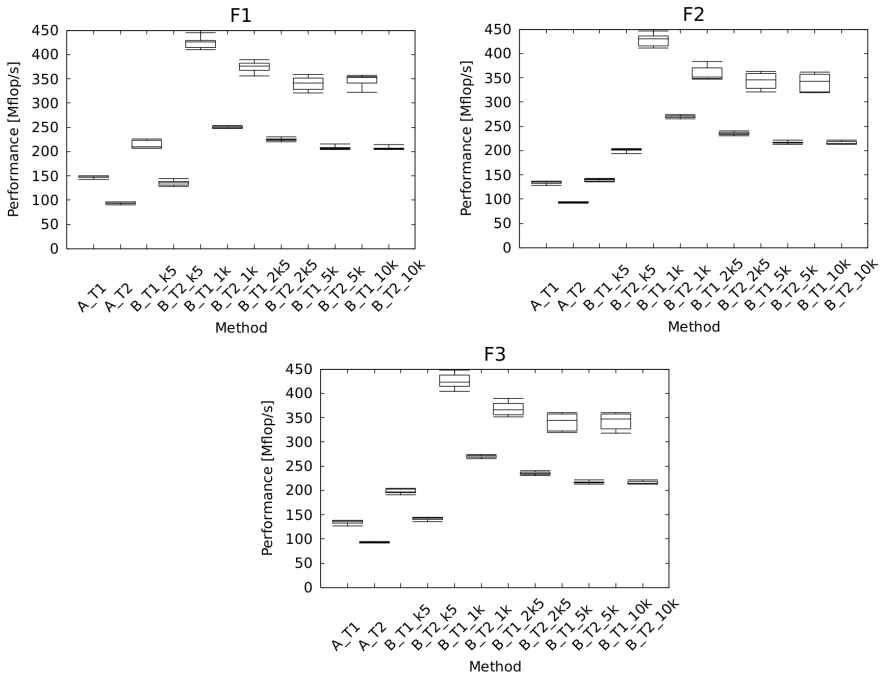

Fig. 1. Sequential performance of methods A and B on test problems F1, F2 and F3. Method B is used with chunk sizes 500, 1000, 2500, 5000 and 10000 vectors. The x-axis is in the format m_t_c, where m is the method (A or B), t is the instruction set ($T1$ or $T2$) and c is the chunk size (for method B).

instruction set significantly influences the performance. Instruction set $T1$ (without IF) leads to higher performance than $T2$, because the code vectorization is more efficient.

In the second experiment, the scalability of the parallel implementations was evaluated using 2, 4, 6, and 12 cores. The time needed to evaluate one generation is reported in the form of boxplots in Fig. 2. The implementation scales almost linearly, but a small communication overhead is present for 6 and 12 cores. The impact of the function set selection on performance is identical to previous experiments. In order to maximize the performance, it is important to correctly choose the chunk size (1000 vectors). Results are only presented for F1 because the results for F2 and F3 are almost identical with F1.

The last experiment was devoted to the island-based LGP. In this case, the boxplots show the best fitness values obtained at the end of 20 independent runs for 1, 2, 4, 6 and 12 islands (Fig. 3). The execution time is identical for all runs. Let a perfect solution be such a solution which obtains a zero fitness. It can be seen that a perfect solution was discovered when more than one island is used for F1, independent of the islands count for F2 and never for F3. Especially for

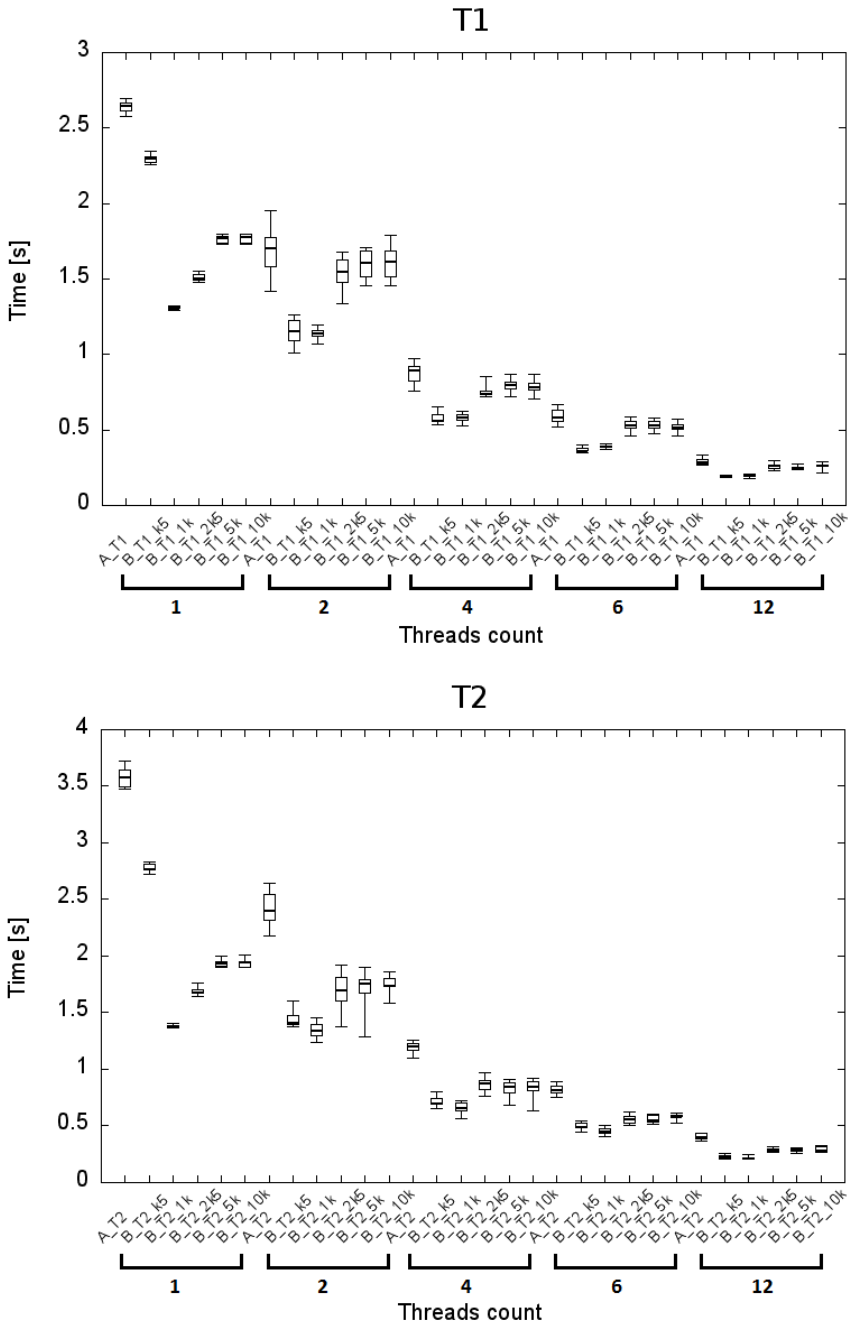

Fig. 2. The time needed to evaluate one generation using 1, 2, 4, 6 and 12 cores (F1 test problem) for instruction sets $T1$ and $T2$.

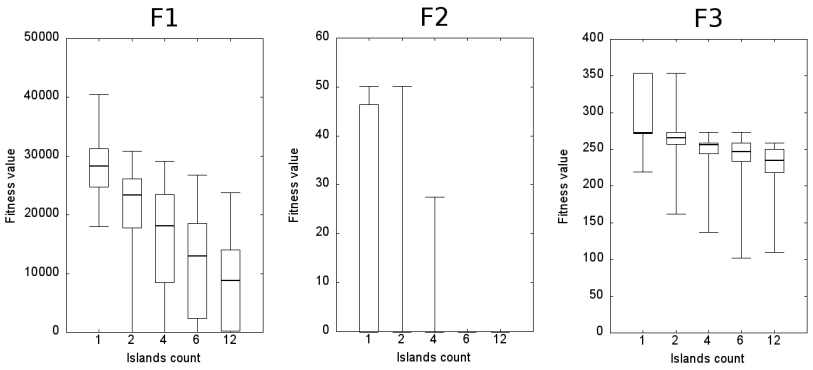

Fig. 3. The best obtained fitness values for F1, F2 and F3 from 20 independent runs on the island model.

F1 and F3, increasing the number of islands has a positive impact on the quality of discovered solutions.

5.2 Hash function design

Figure 4 summarizes the execution time (performance) and fitness values obtained for the hash function design problem using different LGP implementations.

A comparison of sequential versions of methods A and B (with the chunk size of 1000 vectors) revealed that method B can save about 40% of the design time. The results are given for 1000 and 5000 IP addresses in the data set. Note that 20 independent runs with identical seeds for method A and B in each run were performed to obtain these boxplots.

The third and fourth graph in Figure 4 compares the quality of solutions evolved using the island model. Results were obtained using 10 independent randomly seeded runs. It can be seen that if the training IP data set is small (1000 IP addresses), a solution is discovered on one of the islands before any migration is carried out. Hence other islands are not needed. LGP utilizing the island model becomes useful for larger data sets - see the rightmost graph in Figure 4 showing a significant improvement in the fitness on 6 islands for a data set containing 5000 IP addresses. Example of evolved hash function is given in Fig. 5.

6 Conclusions

In this paper, we presented several parallel implementations of LGP devoted to the common multi-core processors. In particular, we proposed an efficient method for evaluation of candidate programs based on dividing the training data into chunks. The main criterion for our evaluation was the throughput,

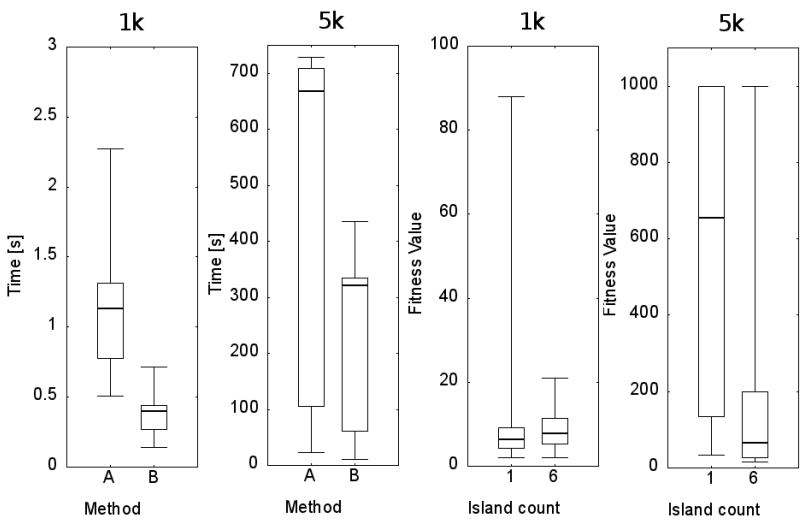

Fig. 4. Sequential execution time for methods A and B (graph 1 and 2) and fitness values for the island model (graph 3 and 4) in the evolutionary design of the hash function using 1k and 5k IP addresses in the dataset.

i.e. how many candidate programs can be evaluated in a given time. The best performing implementation utilizes the island model and the training data partition into chunks. These implementations were compared using three symbolic regression problems and hash function design problem. Unfortunately, the available literature does not provide results from other parallel LGP implementations suitable for a fair comparison. Our future work will be focused on the evolutionary design of efficient hash functions for network applications using the proposed parallel LGP.

```
int Hash (int x ){
  r[0] = x
  r[4] = 0x5be0cd19
  r[6] = r[4]
  r[1] = r[0]
  r[7] = r[4] and 0x3c6ef372
  r[0] = RightRotation(r[0], r[7])
  r[0] = r[6] xor r[0]
  r[4] = not r[1]
  r[0] = r[4] xor r[0]
  return r[0]
}
```

Fig. 5. Example of LGP individual.

Acknowledgments. This work was supported by Brno University of Technology project FIT-S-14-2297 and The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602.

References

1. Brameier, M., Banzhaf, W.: Linear genetic programming. Springer, New York (2007)
2. Cheang, S.M., Leung, K.S., Lee, K.H.: Genetic parallel programming: Design and implementation. Evolutionary Computation 14(2), 129–156 (2006)
3. Dagum, L., Enon, R.: Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE 5(1), 46–55 (1998)
4. Defoin Platel, M., Clergue, M., Collard, P.: Maximum homologous crossover for linear genetic programming. In: Genetic Programming, vol. 2610, pp. 194–203. Springer Berlin Heidelberg (2003)
5. Harding, S., Banzhaf, W.: Fast genetic programming on gpus. In: Genetic Programming, Lecture Notes in Computer Science, vol. 4445, pp. 90–101. Springer Berlin Heidelberg (2007)
6. Hrbacek, R.: Bent functions synthesis on xeon phi coprocessor. In: Mathematical and Engineering Methods in Computer Science. pp. 88–99. LNCS 8934, Springer Verlag (2014)
7. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Genetic programming, pp. 70–82. Springer (2003)
8. Knuth, D.E.: The art of computer programming (volume 3) (1973)
9. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, vol. 1. MIT press (1992)
10. Lusk, E., Huss, S., Saphir, B., Snir, M.: Mpi: A message-passing interface standard (2009)
11. Majewski, B.S., Wormald, N.C., Havas, G., Czech, Z.J.: A family of perfect hashing methods. The Computer Journal 39(6), 547–554 (1996)
12. Maurer, W.D., Lewis, T.G.: Hash table methods. ACM Computing Surveys (CSUR) 7(1), 5–19 (1975)
13. Oltean, M., Grosan, C.: A comparison of several linear genetic programming techniques. Complex Systems 14(4), 285–314 (2003)
14. Oussaidene, M., Chopard, B., Pictet, O.V., Tomassini, M.: Parallel genetic programming and its application to trading model induction. Parallel Computing 23(8), 1183–1198 (1997)
15. Poli, R., Langdon, W.B., McPhee, N.F., Koza, J.R.: A field guide to genetic programming. Lulu. com (2008)
16. Tomasini, M.: Spatially structured evolutionary algorithms (2005)
17. Uy, N.Q., Hoai, N.X., ONeill, M., McKay, R.I., Galván-López, E.: Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genetic Programming and Evolvable Machines 12(2), 91–119 (2011)
18. Wilson, G., Banzhaf, W.: A comparison of cartesian genetic programming and linear genetic programming. In: Genetic Programming, pp. 182–193. Springer (2008)