EXTENDED FINITE-STATE MACHINE INFERENCE WITH PARALLEL ANT COLONY BASED ALGORITHMS

Daniil Chivilikhin
Computer Technologies Laboratory
ITMO University, Saint-Petersburg, Russian Federation
chivdan@rain.ifmo.ru

Vladimir Ulyantsev
Computer Technologies Laboratory
ITMO University, Saint-Petersburg, Russian Federation
ulyantsev@rain.ifmo.ru

Anatoly Shalyto
Computer Technologies Laboratory
ITMO University, Saint-Petersburg, Russian Federation
shalyto@mail.ifmo.ru

Abstract This paper addresses the problem of inferring extended finite-state machines (EFSMs) with parallel algorithms. We propose a number of parallel versions of a recent EFSM inference algorithm MuACO. Two of the proposed algorithms demonstrate super-linear speedup.

Keywords: Extended finite-state machine, Induction, Learning, Parallel algorithm, Synthesis learning

1. Introduction

The automata-based programming paradigm [6] allows to infer the logical structure of software automatically using behavior examples and temporal properties [8]. This logical structure is expressed in the form of an extended finite-state machine (EFSM), which is quite appropriate for reactive (event-based) systems [9]. The EFSM acts as a controller in an *automated-controlled object*, which consists of an EFSM and a controlled object (software system). The EFSM receives input events from *event suppliers*, and sends commands to the controlled object, which processes them and makes corresponding actions (e.g. calls its methods). By using search-based optimization such as evolutionary algorithms [2] to automatically infer EFSMs from examples of desired behavior, and also by incorporating temporal properties that the target program should satisfy, we can get correct-by-design control programs automatically. This is in contrast with the traditional approach when the system is first designed and implemented and only then it is tested and verified.

The main issue in this scheme is that the problem of inferring EFSMs from behavior examples is extremely computationally hard. Even sophisticated algorithms require a long time to find EFSMs in quite simple cases [8]. Therefore, in order to bring automata-based programming closer to being practically applicable in industry, efficient parallel EFSM inference algorithms are needed.

An example of early work in parallel algorithms for FSM inference is [7], where a parallel genetic algorithm (GA) was used to synthesize FSMs from input/output sequences. Recent publications on EFSM inference such as [8] and [3] do not study parallel implementations.

In this work we study parallelization schemes for the Mutation-Based Ant Colony Optimization (MuACO) algorithm proposed in [3], which proved to be more efficient than GA for the problem of FSM inference. A number of parallel versions of MuACO are presented and experimentally evaluated on a shared-memory multi-core machine. Some of the presented parallel algorithms demonstrate super-linear speedup.

The rest of this paper is structured as follows. Section 2 gives an overview of the MuACO algorithm. In Section 3 the EFSM inference problem is described. Section 4 describes the proposed parallel algorithms. Experimental results are presented in Section 5 and Section 6 concludes.

2. MuACO: Mutation-Based Ant Colony Optimization

In this section we briefly describe MuACO. We only discuss details that are essential to understand what is done in this paper. For a full description of the algorithm please refer to [3].

MuACO works with a special graph called a *search graph*, where nodes correspond to FSMs and edges correspond to EFSM mutations. Each

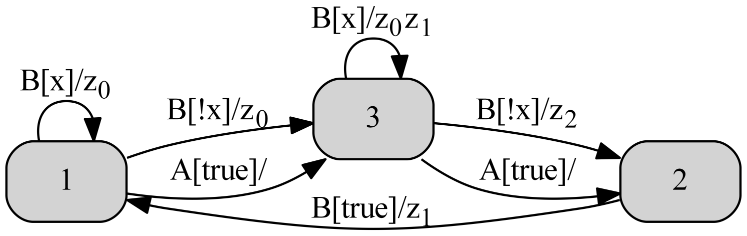

Figure 1. An example of an extended finite-state machine with three states. Each edge is marked with an input event and Boolean formula over the input variables (before the slash) and a sequence of output actions (after the slash).

search graph edge has associated pheromone and heuristic information values. The algorithm uses two types of EFSM mutation operators. The first mutation operator selects a random transition in the EFSM and changes its destination state to a new state, which is selected uniformly and randomly from the set of all states, excluding the old destination state of the transition. The second mutation operator with a certain probability modifies the set of transitions in each state. It can either delete a transition from a state, or add a new random transition to the state.

The algorithm starts off with a randomly generated solution (EFSM). Then, on each *colony iteration* a number of *ants* are launched to search for solutions. Each ant starts with the node of the search graph associated with the best-so-far solution. The ant has a limited number of steps. On each step it either:

- creates a number of new solutions by mutating its current solution and moves to the one having the largest fitness function value;
- or probabilistically selects the next node from the existing successors of the current node.

When all ants have finished searching for solutions, pheromone values are updated for all graph edges. If for a fixed number of colony iterations the best fitness value does not increase, the algorithm is restarted. In the original MuACO the algorithm was restarted from a randomly generated solution. However, in this work we found that it is more efficient to take the best-so-far solution as the initial one, manually decreasing its fitness value to prevent immediate stagnation.

3. Extended Finite-State Machine Inference

An *extended finite-state machine* is a septuple $\langle S, s_0, \Sigma, \Delta, Z, \delta, \lambda \rangle$, where S is a set of states, $s_0 \in S$ is the initial state, Σ is a set of input events and Δ is a set of output actions. Z is a set of Boolean *input variables*, $\delta\colon S \times \Sigma \times 2^Z \to S$ is the *transitions function* and $\lambda\colon S \times \Sigma \times 2^Z \to \Delta^*$ is the *actions function*. EFSMs in the inference algorithms are encoded in the form of full transition tables, where each cell corresponds to a transition in a certain state triggered by an event.

In the so-called specification-based EFSM inference the user provides some behavior samples that the target program should demonstrate. For instance, one could provide test examples [8], test scenarios [10] or negative scenarios. The user may also specify temporal properties that the EFSM should satisfy. In this work we use test scenarios and temporal properties expressed in Linear Temporal Logic (LTL).

A *test scenario* is a sequence of triples $\langle e, \varphi, O\rangle$ called *scenario elements*, where e is an event, φ is a Boolean formula over the input variables and O is a sequence of output actions. An EFSM is said to be compliant with a scenario element in state s if it has a transition from this state marked with e, φ and O. Correspondingly, an EFSM is compliant with a test scenario if it is compliant with all scenario elements in the corresponding states when scenario elements are processed sequentially. For example, the EFSM in Figure 1 is compliant with scenario $\langle B, x, (z_0)\rangle\langle B, x, (z_0)\rangle\langle A, \text{true}, ()\rangle$ and is not compliant with scenario $\langle B, x, (z_0)\rangle\langle A, \text{true}, (z_1)\rangle$.

A LTL formula consists of problem-specific propositional variables **Prop**, Boolean logical operators, and a set of temporal operators, such as **G** (**G**lobally in the future), **X** (ne**X**t), **F** (in the **F**uture), etc. For the case of LTL formulae expressing properties of EFSMs, the set of propositional variables can include terms:

- $\forall e \in \Sigma : \textbf{wasEvent}(e)$ – a transition marked with event e has been triggered;
- $\forall a \in \Delta : \textbf{wasAction}(a)$ – a transition marked with action a has been triggered.

The problem is therefore to find an EFSM with a given number of states that is compliant with all given test scenarios and LTL formulae. This problem was previously tackled in [8] with a genetic algorithm. We adopt a fitness function f proposed in that work to evaluate the degree of an EFSM's compliance with test scenarios and LTL formulae. The fitness function is based on string edit distance [4] and a specially developed EFSM verifier.

4. Parallel MuACO Algorithms

In this paper we propose a number of parallel versions of MuACO. These were implemented on a shared-memory machine but can be straightforwardly extended to work on a cluster of separate machines. All presented approaches implement the coarse-grained parallelization scheme, i.e. when the interaction between algorithms is rare.

4.1 Independent Parallel MuACO

In the first and simplest algorithm we call the Independent Parallel MuACO, a number of algorithm instances is launched in parallel for the same problem. That is, having m processors, m MuACO algorithms are launched in parallel, each from a separate randomly generated initial solution. An important detail is to provide each algorithm with a separate random number generator.

The motivation to this simple approach is that the problems MuACO deals with are combinatorial ones. In most cases a single fitness computation does not take much CPU time, however many solutions have to be evaluated in order to find the optimal one. By using a number of parallel independent search algorithms we increase the amount of different solutions explored during the same time period, thus, possibly, decreasing the amount of time needed to find the optimal one.

4.2 Parallel MuACO with Shared Best Solutions

The second step is to see if adding some interaction between independent MuACO instances can increase performance. The conventional interaction scheme in parallel evolutionary algorithms is to migrate some solutions between individual populations of algorithms running in parallel (see, for example, [1]). However, due to the nature of MuACO, there is no clear way of adopting this scheme in our case, since MuACO does not keep a population of solutions.

Therefore, one of the easiest ways of adding interaction is to keep a cache of best solutions found by each algorithm and share it among them. Suppose with have m MuACO algorithms running in parallel. Let K be an array of length m for storing best found solutions, with K_i reserved for the i-th algorithm (Figure 2). Each i-th algorithm updates the value of K_i each time it finds a solution better than the one stored there. When a sequential MuACO comes to a stagnation state, it is restarted from its best found solution as described in Section 2. In this parallel variant the stagnated i-th algorithm will be restarted, taking

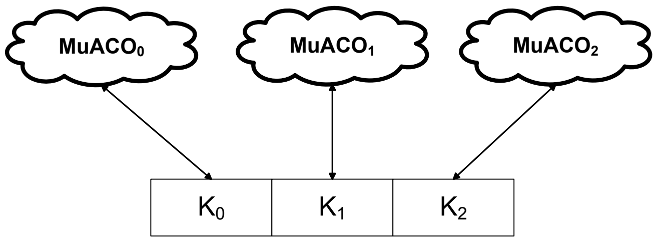

Figure 2. Parallel MuACO with shared best solutions.

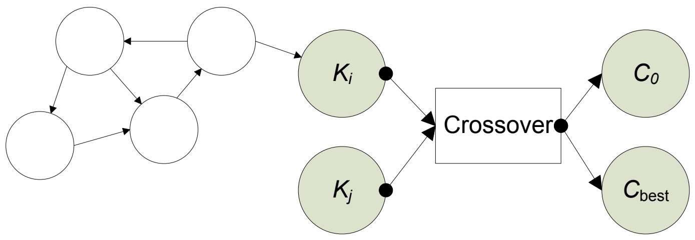

Figure 3. Parallel MuACO with crossover and shared best solutions: the best solution of the i-th thread K_i is crossed over with a remote thread best solution K_j. White nodes at the left represent a part of the i-th algorithm's search graph.

K_j as the initial solution, where j is selected uniformly randomly from $\{0 \ldots i-1, i+1 \ldots m-1\}$.

4.3 Parallel MuACO with Crossover

As the algorithm described above, this third and final version of parallel MuACO uses a cache of shared best solutions, but also adopts a crossover operator as in genetic algorithms.

On each colony iteration of the i-th MuACO (before ants are launched) a random solution K_j $(j \neq i)$ is picked from the cache of shared best solutions K. A crossover operator is applied to this solution K_j from a remote algorithm and the best solution K_i found by the i-th algorithm. The offspring produced by the crossover operator are then evaluated with the fitness function f, and only one child solution C_{best} with the best fitness value is left (Figure 3).

This solution C_{best} is added to the search graph of the i-th algorithm as a child of the best-so-far solution. In the consequent colony iteration one ant will be launched from C_{best}, and the rest of the ants will be launched, as before, from the node associated with the best-so-far solution.

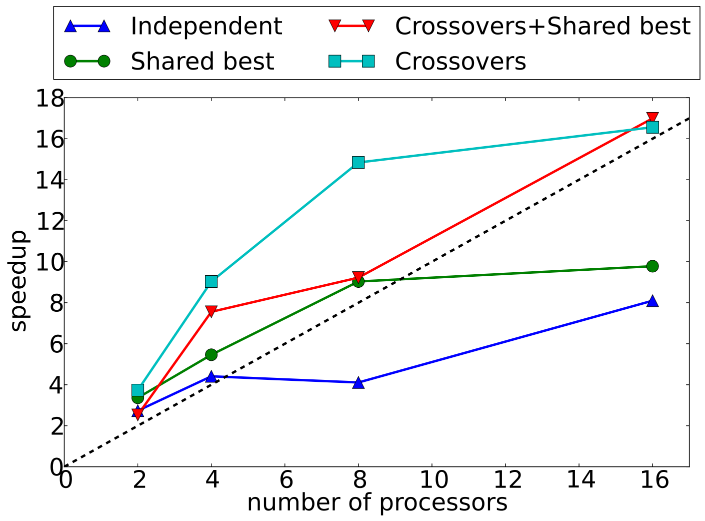

Figure 4. Speedup plots for different parallel MuACO algorithms.

Of course, the crossover operator applied here is problem-specific. Here we studied two crossover operators used in EFSM inference. The first crossover operator randomly mixes the transition tables of the two parent EFSMs and is thus called "Simple". The second crossover operator called "Test-based" proposed in [8] takes into account the transitions of the parent EFSMs the were used during fitness evaluation. The type of crossover is selected uniformly and randomly each time it has to be applied.

5. Experiments

Prior to conducting experiments the parameter values of sequential MuACO were automatically selected using the *irace* package [5]. This package implements a racing protocol and allows to derive appropriate parameter values that allow the algorithm to perform efficiently on a certain class of problem instances.

All algorithms were run on 50 randomly generated problem instances. Each of them was derived from a random EFSM with 10 states. For each problem instance each algorithm was run until finding an EFSM that satisfies all test scenarios and LTL formulae. To calculate speedup we measured the average wall clock running time of the sequential algorithm and divided it by the average running time of a particular parallel MuACO algorithm.

The following parallel algorithms were compared:

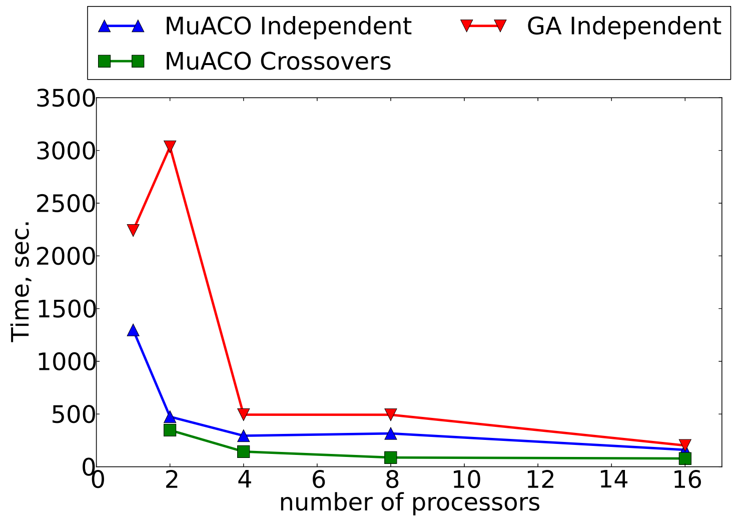

Figure 5. Execution time of independent and crossover parallel MuACO in comparison to independent parallel GA.

- independent parallel MuACO ("Independent");
- parallel MuACO with shared best solutions ("Shared best");
- parallel MuACO with crossovers, each algorithm is restarted from its own best solution ("Crossovers");
- parallel MuACO with crossovers and shared best solutions ("Crossovers + Shared best").

We used a machine with a 24-core AMD Opteron 6234 2.4 GHz processor. Experimental results in the form of speedup plots are shown in Figure 4. The mean runtime of the sequential MuACO was 1392 seconds.

As can be seen from Figure 4, all parallel MuACO algorithms demonstrate an increase of performance as the number of used processors grows. All algorithms are pretty much the same for the case of two processors. The "Independent" algorithm with no interaction is the worst among them, demonstrating a speedup of only about 8 for 16 processors. However, theoretically it is very robust, since for this algorithm performance will never decrease with the increase of the processors amount.

The "Shared best" algorithm scales well up until 8 processors, however its performance does not significantly change when 16 processors are used. And only two algorithms that use crossovers demonstrate super-linear speedup all the way, with the "Crossovers" algorithm being on average slightly better than "Crossovers+Shared best" for 2–8 processors.

The plot in Figure 4, however, only displays speedup measured from mean execution time of the algorithms. To verify the statistical significance of the differences between parallel algorithm performance, we used the Wilcoxon statistical test [11]. The test was applied to each pair of algorithms, comparing the distributions of running time for the case of 16 processors. The results of statistical tests are presented in Table 1.

As expected, the performance of all parallel algorithms (2–5) significantly differs from the performance of the single-thread MuACO (1), which is indicated by small p-values in the first row of the table. Algorithms with crossover (4) and (5) are significantly better than independent parallel MuACO (2). For the case of 16 processors, algorithms with crossover yield similar running time (p-value = 0.939).

We also implemented an independent parallel version of the GA from [8]. GA parameters were also tuned using *irace*. Plots showing mean execution time of this GA in comparison with independent and crossover parallel MuACO are shown in Figure 5. It can be seen that the independent GA is always slower than independent MuACO, which is in turn always slower than crossover MuACO. The independent GA is 2–9 times slower than crossover MuACO.

Table 1. Wilcoxon test p-values for algorithms Single (1), Independent (2), Shared best (3), Crossovers (4), Crossovers+Shared best (5).

	(1)	(2)	(3)	(4)	(5)
(1)	–	1.602×10^{-8}	1.946×10^{-9}	6.581×10^{-14}	4.067×10^{-14}
(2)	–	–	0.8582	0.103	0.128
(3)	–	–	–	0.026	0.025
(4)	–	–	–	–	0.939
(5)	–	–	–	–	–

6. Conclusion and Future Work

We have proposed several parallel versions of the MuACO EFSM inference algorithm. It has been shown that the use of crossover in parallel MuACO significanly improves performance. Parallel MuACO algorithms with crossover demostrated super-linear speedup.

Future work includes creating a hybrid GA–MuACO parallel algorithm where solutions will be exchanged between GA and MuACO instances.

Acknowledgements

This work was financially supported by the Government of Russian Federation, Grant 074-U01, and also partially supported by RFBR, research project No. 14-01-00551 a.

References

[1] E. Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.

[2] T. Back, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation. 1st edition.* IOP Publishing Ltd., 1997.

[3] D. Chivilikhin and V. Ulyantsev. MuACOsm: a new mutation-based ant colony optimization algorithm for learning finite-state machines. In *Proc. 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO)*, pages 511–518, 2013.

[4] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.

[5] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The `irace` package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[6] A. A. Shalyto. Software automation design: algoritmization and programming of problems of logical control. *J. Comput. Syst. Sc. Int.*, 6:899–916, 2000.

[7] S. Tongchim and P. Chongstitvatana. Parallel genetic algorithm for finite-state machine synthesis from input/output sequences. In *Proc. Conference on Genetic and Evolutionary Computation (GECCO)*, pages 20–24, 2000.

[8] F. Tsarev and K. Egorov. Finite state machine induction using genetic algorithm based on testing and model checking. In *Proc. 13th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 759–762, 2011.

[9] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach. 1st edition* McGraw-Hill, 1998.

[10] V. Ulyantsev and F. Tsarev. Extended finite-state machine induction using SAT-solver. In *Proc. 10th International Conference on Machine Learning and Applications and Workshops (ICMLA)*, pages 346–349, 2011.

[11] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bull.*, 1(6):80–83, 1945.