PSMA: A Parallel Algorithm for Learning Regular Languages

Hasan Ibne Akram[1], **Alban Batard**[2], **Colin de la Higuera**[2], and **Claudia Eckert**[1]

[1]Technische Universität München, Munich, Germany,
{hasan.akram,claudia.eckert}@sec.in.tum.de
[2] University of Nantes , Nantes, France, {batard,cdlh}@univ-nantes.fr

Abstract

Inferring a regular language from examples and counter-examples is a classical problem in grammatical inference. It is also known as a variant of automata synthesis or grammar induction problems and corresponds to finding the smallest DFA consistent with a labelled sample of strings. The best known algorithm to solve this problem runs in polynomial (but cubic) time, and for large learning samples the algorithm cannot be used. We introduce a parallel version of the RPNI algorithm which solves the above question, and we study the main challenges toward parallelization of such class of algorithms to run in a multi-core environment. We report experiments showing the viability of the technique.

1 Introduction

Inferring a regular language from examples and counter-examples is a classical problem in grammatical inference [1]. It is also known as automata synthesis or grammar induction and corresponds to finding the smallest DFA consistent with a labelled sample of strings. The classical algorithm (RPNI [2]) to solve this problem runs in polynomial (but cubic) time, and in practical situations where the size of the learning sample is large the algorithm cannot be used. A number of alternative algorithms have been proposed in the past 40 years [3, 4, 5]. In grammatical inference, the only other attempt (to our knowledge) of parallelizing learning algorithms was made in the alternative framework of active learning [6, 7].

Our Parallel State Merging Algorithm (PSMA) is a EREW PRAM learning algorithm for learning DFA. This algorithm is strongly based on the sequential state merging algorithm RPNI [2] and adopts a multi-core processor computation paradigm that allows to test possible state merges in parallel.

2 Preliminaries

Let Σ be a non-empty set of symbols called ***letters***. Σ^* is the set of all strings over the alphabet Σ where a ***string*** $x \in \Sigma^*$ is a finite sequence of letters $x = x_1 x_2 \cdots x_n$. A ***language*** $\mathcal{L}$ is any subset of Σ^*. If $x = uv$ is a string, then u is a ***prefix*** of the string x.

The ***prefix set*** $Pref(\mathcal{L})$ of the language $\mathcal{L}$ is defined as $Pref(\mathcal{L}) = \{u \in \Sigma^* : uv \in \mathcal{L}\}$.

A ***Deterministic Finite Automaton (DFA)*** is a quintuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where Σ is an alphabet, Q is a set of finite states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta : Q \times \Sigma \to Q$ is the transition function.

3 The problem

Let $\langle S_+, S_- \rangle$ be a finite sample of some language $\mathcal{L}$ consisting of a subset $S_+ \subseteq \mathcal{L}$, set of positive strings of the language $\mathcal{L}$ and $S_- \subseteq \Sigma^* \setminus \mathcal{L}$, set of negative strings of the language $\mathcal{L}$. Throughout the paper we assume the samples to be ***non-conflicting***, *i.e.*, $S_+ \cap S_- = \emptyset$.

In a DFA learning (or synthesis) problem, we are given a sample $\langle S_+, S_- \rangle$ as above, and the goal is to find *the* language $\mathcal{L}$. Obviously, there is a number of languages are such that $S_+ \subseteq \mathcal{L}$ and $S_- \subseteq \Sigma^* \setminus \mathcal{L}$: such a language is said to be ***consistent*** with $\langle S_+, S_- \rangle$. As a combinatorial problem, the corresponding goal is to find the *smallest* consistent DFA. As an inference problem, we want to have an algorithm which returns a DFA and furthermore the algorithm converges with the data, *i.e.*, there is a guarantee that with more and more elements in S_+ and S_-, we can be sure to find $\mathcal{L}$.

The general strategy used by the most common family of DFA learning algorithms is that of ***state merging***. The starting point is the ***Prefix Tree Acceptor*** (PTA), built from S_+: this is a tree-like DFA $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ PTA(S_+) defined as follows: $\{Q = q_u : u \in Pref(S_+)\}$, $\forall ua \in Pref(S_+) : \delta(q_u, a) = q_{ua}$, $F = \{q_u : u \in S_+\}$.

State-merging algorithms maintain a set of RED states corresponding to the *confirmed* states, *i.e.*, those present in the final DFA, and a set of BLUE states, successors of the RED states and candidates for merging. The goal is to generalise the language recognised by the running DFA by iteratively choosing one RED state and one BLUE state and attempting to merge these together: if the result of this merge is not over-general (*i.e.*, no string from S_- is recognised: the two states are then said to be ***compatible*** and ***incompatible*** otherwise) the merge is kept; if not, it is rejected. Whenever a particular BLUE state can be merged with no RED state it gets promoted: it becomes RED and all its non RED successors become BLUE.

RPNI [2] is a deterministic algorithm where the merge compatibilities between two states are checked sequentially in a predefined (length lexicographic) order. Whenever a merge is rejected, RPNI tries to merge another pair of states and continues until no further merges are possible. It is known that RPNI ensures identification of DFA in the limit and works in polynomial time [1].

4 The PSMA Algorithm

We introduce a new learning algorithm called PSMA (*Parallel State Merging Algorithm*) which obtains the same result as RPNI but makes use of a multi processor architecture.

Let us denote by n the number of states of the PTA and suppose these are numbered (in a breadth-first way) from 0 to $n-1$. Pairs of (indexes of) states will be ordered: $\langle i, j \rangle < \langle i', j' \rangle$ if $j < j'$ or $j = j'$ and $i < i'$. The first element of each pair corresponds to a RED state, the second to a BLUE state. For example, $\langle 3, 5 \rangle < \langle 2, 6 \rangle$ and $\langle 3, 5 \rangle < \langle 4, 5 \rangle$.

The master processor node M takes the responsibility of initialising and updating the shared data:

- a set $\mathcal{R}$ of RED states; initially $\mathcal{R} = \{0\}$;
- a set $\mathcal{B}$ of BLUE states; initially $\mathcal{B} = \{i : \exists a \in \Sigma \text{ such that } \delta(q_0, a) = q_i\}$;
- a table containing for each pair $\langle i, j \rangle$ with $i \in \mathcal{R}$, $j \in \mathcal{B}$, the information $T[i][j]$;
- a counter called SIT which corresponds to the position in the table up to which the merges are validated by the master so far.

T can be implemented as a queue in order to have a small data struicture: it will only contain the values corresponding to *active* pairs of states. $T[i][j]$ will take the following possible values:

- if the merge between state $i \in \mathcal{R}$ and $j \in \mathcal{B}$ has an unknown status the value is **U**;
- if the merge is being considered in the actual context by some processor the value is **C**;
- if the merge is being considered by some processor, but the master has accepted a merge the value is **R**;
- if the merge has been discarded the value is **D**;
- if the merge is proposed in the current context by one processor the value is **P**;
- if the merge is accepted and has been validated by the master processor, the value is **A**;

The **master processor** is in charge of updating the shared information. It does so by looking at the entry in the table T corresponding to SIT= $\langle i, j \rangle$:

- if $T[i][j] =$**D** and is the last corresponding to that particular BLUE (j) the table is updated by promoting state j. This consists in (1) updating the local $\mathcal{B}$ and $\mathcal{R}$ sets (*i.e.*, state

j is moved from $\mathcal{B}$ to $\mathcal{R}$, and successors of state j are moved to $\mathcal{B}$; (2) pairs $\langle j, k\rangle$ are inserted in their correct position in the table, where k is any BLUE state; (3) counter SIT is incremented.

- if $T[i][j]$ =**P** the following update takes place: (1) $T[i][j] \leftarrow$ **A**; (2) $\forall k > j,\ T[i][k] \leftarrow$ **D**; (3) $\forall\langle h, k\rangle >$SIT, if $T[h][k]$ =**C**, $T[h][k] \leftarrow$**R** and if $T[h][k]$ =**P**, $T[h][k] \leftarrow$**U**; (4) $\mathcal{B}$ and $\mathcal{R}$ are updated in the usual way RPNI does it; (5) SIT is incremented to $< \langle i', j'\rangle$ where i' is the first RED state and j' is the next BLUE state.

Each other processor P_z plays the part of a slave processor. It finds the smallest $\langle i, j\rangle$, from SIT onwards such that $T[i][j]$ =**U**. P_z sets $T[i][j]$ to **C**.

Then P_z runs RPNI with the help of the table T: every time a merge between two states r and b is to be tested, P_z checks the table T. If $T[r][b]$ is **A** the merge is done; in all other cases the compatibility result is *fail*. When P_z has to test the merge between i and j, it really does it and returns the compatibility result.

Processor P_z does the following when it has finished its task, consisting in testing the merge between i and j:

1. If $T[i][j]$=**C** then if the merge P_z has tested is OK, it updates $T[i][j]$ to **P** (proposed). If the merge is not OK, it updates $T[i][j]$ to **D**.
2. If $T[i][j]$=**R** (which means that at least one new merge has been taken into account), then if the merge is not OK, it updates $T[i][j]$ to **D**. If the merge is OK, nothing can be decided: it updates $T[i][j]$ to **U**.
3. It searches for another merge to be checked.

A small analysis of the algorithm

The key idea is that each slave processor is kept busy at all times: it finds the next job in the queue and will try to check a merge in which the context is that all the previous unfinished jobs are going to fail. If the result of this job is **incompatible**, then the result will hold even if an unfinished job returns **compatible**. The *bad* case occurs when the job returns **compatible**: the result in only kept if all previous compatibility tests fail.

5 Experimental Results

We have conducted our experiments on a multi-core machine having Intel® Xeon® processors, 4x6 cores, 2.66 GHz, 16 MB cache memory, RAM: 128 GB. A series of runs of the algorithm was performed using datasets generated by Gowachin[1], with different sizes of targets DFA, number of examples, and number of slaves used in the algorithm.

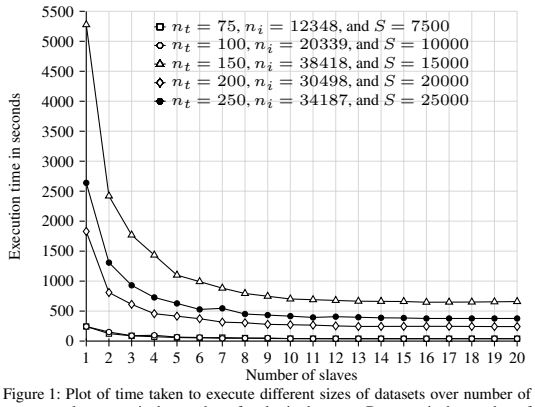

Figure 1: Plot of time taken to execute different sizes of datasets over number of slaves. n_t is the number of nodes in the target DFA, n_i is the number of nodes in the initial PTA and S is the sample size.

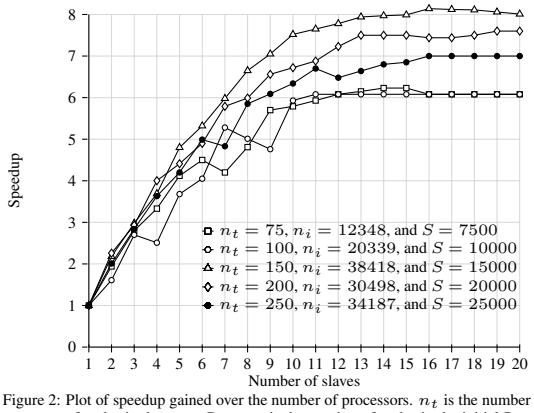

Figure 2: Plot of speedup gained over the number of processors. n_t is the number of nodes in the target DFA, n_i is the number of nodes in the initial PTA and S is the sample size.

[1] Gowachin allows to generate artificial datasets for testing DFA learning methods: `http://www.irisa.fr/Gowachin/`

$p \rightarrow$ $n \downarrow$	1	2	3	4	5	6	7	8	9	10	12	14	16	18	20
12348	1	0.97	0.93	0.83	0.82	0.75	0.6	0.6	0.63	0.58	0.51	0.45	0.38	0.34	0.3
20339	1	0.8	0.9	0.63	0.74	0.68	0.75	0.63	0.53	0.59	0.51	0.43	0.38	0.34	0.3
38418	1	1.09	0.99	0.92	0.96	0.89	0.85	0.83	0.78	0.75	0.65	0.57	0.51	0.45	0.4
30498	1	1.13	0.99	1	0.88	0.82	0.83	0.75	0.73	0.67	0.6	0.53	0.46	0.42	0.38
34187	1	1.01	0.95	0.91	0.84	0.83	0.69	0.73	0.68	0.63	0.54	0.49	0.44	0.39	0.35

Table 1: Efficiency E as a function of n (PTA size) and p (number of processors).

From Figure 1 it appears that significant speedup can be gained when using large datasets (leading to large PTA). As we increase the number of slaves, the speedup decreases until reaching a saturation point (here with 13 slaves). This behaviour can be explained as follows: the hypothesis under which a processor checks the merge it has to test has a probablity of being invalidated that grow with the number of processors.Therefore, at some point, increasing the number of processors brings no speedup.

Let the *sequential execution time* [8], *i.e.*, the time taken to execute the algorithm with a single processor be denoted by T_1. The *parallel execution time*, *i.e.*, the execution time for the algorithm with p processors be denoted by T_p. The *speedup* is defined as $S = \frac{T_1}{T_p}$. The *efficiency* is defined as $E = \frac{S}{p}$. Figure 2 depicts the speedup of the algorithm over the number of processors. Table 1 shows the efficiency table as a function of number of states in the PTA and number of slaves. Efficiency appears to be very high (Table 1) with lower number of slaves and decreases as the number of slaves increases due to the similar reason as limitation in speedup gain explianed earlier. Adding more resources to gain relatively small fraction of speedup (or no speedup) results in low efficeincy.

6 Conclusion & Future Outlook

In this paper we have described a parallel version of a state merging algorithm for inferring regular languages (RPNI). Experimental results have been presented based on a Java implementation[2] of the algorithm, where a significant performance gain has been obtained. However, the results also indicate that there is a limit for the speedup gain.

In this version, the algorithm remains deterministic and depends on a predefined ordering of the states. A parallelisation of the Evidence Driven State Merging algorithm (EDSM, [4]) can also be done[3]: in this case, between the different **P** values returned by the slave processors, the master will choose the one with the highest score.

References

[1] de la Higuera, C.: Grammatical inference: learning automata and grammars. Cambridge University Press (2010)

[2] Oncina, J., García, P.: Identifying regular languages in polynomial time. In Bunke, H., ed.: Advances in Structural and Syntactic Pattern Recognition. Volume 5 of Series in Machine Perception and Artificial Intelligence. World Scientific (1992) 99–108

[3] Trakhtenbrot, B., Barzdin, Y.: Finite Automata: Behavior and Synthesis. North Holland Pub. Comp., Amsterdam (1973)

[4] Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In Honavar, V., Slutski, G., eds.: Grammatical Inference, Proceedings of ICGI '98. Number 1433 in LNAI, Springer-Verlag (1998) 1–12

[5] Heule, M.J.H., Verwer, S.: Exact dfa identification using sat solvers. In Sempere, J.M., García, P., eds.: Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings. Volume 6339 of Lecture Notes in Computer Science., Springer (2010) 66–79

[6] Balcázar, J.L., Diaz, J., Gavaldà, R., Watanabe, O.: An optimal parallel algorithm for learning DFA. In: Proceedings of the 7th COLT, New York, ACM Press (1994) 208–217

[7] Angluin, D.: Learning regular sets from queries and counterexamples. Information and Control **39** (1987) 337–350

[8] Gupta, A., Kumar, V.: Performance properties of large scale parallel systems. J. Parallel Distrib. Comput. **19**(3) (1993) 234–244

[2] http://pagesperso.lina.univ-nantes.fr/~cdlh/Downloads/RPNI.tar.gz

[3] http://pagesperso.lina.univ-nantes.fr/~cdlh/Downloads/RPNIP.tar.gz