Fast Parallel Recognition of LR Language Suffixes

E. Bertsch [a] and M.-J. Nederhof [b,1]

[a] *Ruhr University, Faculty of Mathematics, Universitätsstraße 150, D-44780 Bochum, Germany*

[b] *University of Groningen, Faculty of Arts, P.O. Box 716, NL-9700 AS Groningen, The Netherlands*

Abstract

It is shown that suffix recognition for deterministic context-free languages can be done on a PRAM multi-processor within the upper complexity bounds of the graph reachability problem.

Key words: Computational complexity, parallel algorithms, programming languages.

1 Introduction

The computational problem of suffix recognition with respect to a formal language L may be stated as follows: for a given string w, decide whether there is some v such that $vw \in L$. For deterministic context-free languages, also known as LR languages, the time complexity of suffix recognition with a single processor was an open problem for a full decade. It had first been addressed in [1] in the context of practical error-recovery techniques for compilers. The set of suffixes of any deterministic language is easily shown to be context-free, but it is not itself deterministic in the general case. To see this immediately, consider the deterministic language

$$L_D = \{da^n b^n c^m \mid n, m > 0\} \cup \{ea^n b^m c^m \mid n, m > 0\}.$$

[1] Supported by the PIONIER project *Algorithms for Linguistic Processing*, funded by NWO (Dutch Organization for Scientific Research).

Informally, depending on whether a possible suffix of the form $a^p b^q c^r$ finishes a string starting with d or with e, the number q of b's must either exceed p or else be equal to r. A pushdown automaton (PDA) simply guessing the right mode would of course be non-deterministic. Note that prefix recognition, by contrast, can be solved trivially by regarding all states of an automaton as being final.

Independent solutions to the suffix recognition problem were presented by [2] and [3]. The approaches differ e.g. in the amount of support they lend to parsing. It turned out that linear time suffices. In view of what follows, it may be noteworthy that graph reachability is also a linear-time one-processor problem (by means of depth-first search).

Assuming the CREW-PRAM model, the time complexity of parallel recognition of context-free languages (CFLs) is known to be $\mathcal{O}(\log^2 n)$ with n^6 processors [4]. For some subclasses the processor bound has been improved: it is n^2 for deterministic CFLs [5], n^3 for linear CFLs [6], and n^3 for unambiguous CFLs [5]. For these three subclasses, [7] offers lower processor bounds, for time complexity $\mathcal{O}(n^{1-\alpha} \log^2 n)$ with $0 < \alpha < 1$.

Parallel recognition of general CFLs within $\mathcal{O}(\log n)$ time has been established on the CRCW model with n^6 processors [8]. On the CREW model, however, it has only been established for subclasses, such as the unambiguous CFLs, with n^7 processors [9], and the deterministic CFLs [10–12]. The solution by [10] to logarithmic-time recognition of deterministic CFLs required n^3 processors. The proof of correctness of the main result was presented on 12 pages of text. The problem was investigated by [11] with a severely restricted processor model (CROW), achieving essentially the same upper bounds as [10]. In [12], Monien, Rytter and Schäpers, henceforth abbreviated as MRS, developed a significantly simpler technique than [10], and showed by additional considerations that the processor bound can be reduced to $n^{2+\epsilon}$. The proof of this added property made strong use of their specific stack item construct.

We are going to show that parallel recognition of suffixes can be done by first executing the MRS method on a given input in such a way that certain intermediate results are stored in further tables, and subsequently computing what amounts to graph reachability in a graph of size $\mathcal{O}(n)$ where n is the string length. The first phase can obviously be done within the complexity bounds proved by MRS; the second phase can be done within the bounds of CREW-PRAM solution of the latter problem, namely $\mathcal{O}(\log^2 n)$ time on $n^{2.376}$ processors [13, p. 248], which dominate the bounds in MRS. Alternatively, the first phase can be performed by the algorithm from [5], with time complexity $\mathcal{O}(\log^2 n)$ and n^2 processors, but also in this case, the second phase dominates the joint costs.

It is also worth noting that in the alternative CRCW-PRAM model (which permits simultaneous assignments to one location whenever the values are identical), graph reachability can be determined in $\mathcal{O}(\log n)$ time on n^3 processors [13, pp. 249-250], which concurs with the results from [10] and [12] for recognition of deterministic CFLs. This is the best combined bound that we are currently able to state for our problem.

2 Informal description of the MRS method

This section gives a survey of the parallel recognition technique published by Monien, Rytter and Schäpers. It is needed here because we cannot provide a simple interface between their method and our use of the specific tables computed during its execution. Our description is, however, less precise than a full presentation of the algorithm would have to be. For further details the original text of [12] should be consulted.

The steps of the algorithm correspond to steps of a deterministic pushdown automaton, which are simulated in terms of a partial representation of stacks. This representation consists of surface configurations of the form $x = (s, i, a)$, where $\mathrm{state}(x) = s$ is the state, $\mathrm{pos}(x) = i$ is the position in the input, and $\mathrm{symb}(x) = a$ is the top element of the stack. The overall goal of performing no more than a logarithmic number of steps is achieved by computing certain data for a substring of length 2^k *after* the corresponding data for the two substrings of length 2^{k-1} of which it consists. If such computations for a given interval take no more than constant time on the set of available processors, the time bound follows. Let us call these substrings of length 2^k k-intervals.

For each k-interval, several kinds of tabular entries are computed. For each surface configuration x, $\mathrm{stack}_k(x)$ is that stack which results from x by deterministic recognition of subsequent input symbols at the very first position of the *next* k-interval. It should be clear that starting with an x of height 1, intervals of length 2^k may lead to corresponding stack heights. While this is not of direct concern to us here, one must note that enough processors are available to e.g. copy any partial stack in constant time.

The top element (surface configuration) of $\mathrm{stack}_k(x)$ is designated as $\mathrm{next}_k(x)$. Furthermore, there is a function, or alternatively a table, $\mathrm{stack}_k(x, y, h)$ with three parameters where x and y are surface configurations and h is a height. It is defined as the stack consisting of a bottom segment of $\mathrm{stack}_k(x)$ of height h, in which the top element has been replaced by y.

The technically most advanced definition of the MRS paper, which happens, however, to contribute greatly to the essential simplicity of the correctness

proof, is that of the function $\mathrm{POP}_k(x, y, h)$. It yields a pair of values (z, h'). Focusing for a moment on y, assume that y is indeed the top element of some $\mathrm{stack}_k(x, y, h)$ as just described. We are interested in the *shortest* stack that follows within the remainder of the current k-interval relative to the input position of y. The top element and height of that lowest stack are the z and h' components of the POP_k entry, respectively. How tabular entries belonging to a k-interval are computed out of entries belonging to the $(k-1)$-interval is shown with notable elegance by a sequence of figures in the MRS paper.

3 Application of the MRS method

The data we need to collect during execution of the MRS algorithm on a candidate suffix is stored in an $\mathcal{O}(n^2)$ sized Boolean matrix $\mathrm{Sub}(x, z)$, where x and z are surface configurations. The entries in this table should become such that $\mathrm{Sub}(x, z) = \mathrm{true}$ if and only if there is a subcomputation of the pushdown automaton starting with a stack represented by x and ending in a stack represented by z, and the height of these two stacks is equal, and all intermediate stacks are at least as high, and z describes a pop configuration. Such an element z 'terminates' the subcomputation started with x.

The table can be constructed by the MRS method as follows. Initially $\mathrm{Sub}(x, z)$ = false for all x and z. After the assignment $(z, h') := \mathrm{POP}_k(x, \mathrm{next}_k(x), h)$, where $h = \mathrm{height}(\mathrm{stack}_k(x))$, in stage (I) of the inner loop in [12, p. 424], execute an additional statement

> if ($h' = 1$ and element z describes a pop configuration) then $\mathrm{Sub}(x, z) :=$ true;

To show that all terminating subcomputations are indeed caught by the above condition, we must note that for any subcomputation beginning at $\mathrm{pos}(x)$ and ending at some $\mathrm{pos}(z)$, there is a smallest k and m such that

$$m * 2^k \leq \mathrm{pos}(x) < (m+1) * 2^k \leq \mathrm{pos}(z) < (m+2) * 2^k.$$

The clearest expression of this is the fact that the numbers $\mathrm{pos}(x)$ and $\mathrm{pos}(z)$ written in binary notation differ in some of their respective binary digits, and the power of 2 of the most significant digit for which they differ is equal to this k. By definition of the POP_k function, some entry at stage k will therefore contain the element that terminates x.

In order to also capture terminating subcomputations of length zero, we let $\mathrm{Sub}(x, x) :=$ true for each element x that describes a pop configuration. This can be performed in constant time by n processors.

Although we have not tried to formulate this observation more precisely, it can certainly be stated that there is a very close correspondence between POP_k in MRS and the notion of k-lowness in [10], and that the first phase of our algorithm could alternatively have been based on that approach. Please note that the second phase as described below does not depend on k-intervals.

4 The set of possible stacks

We must now discuss the properties of suffixes in more detail with respect to computations of a deterministic PDA. Assume a string $vw \in L$ of which we only have w. String w could be recognized by the PDA if the precise stack built up by reading the prefix v were available for further computation. More concretely, for one such stack, we could split up the remaining string (suffix) w into substrings $w[i_0 : i_1]$, $w[i_1 : i_2]$, $\ldots$, $w[i_{m-1} : i_m]$, where $i_0 = 0$ and $i_m = n$, each one of which consumes one symbol of the stack in existence prior to the reading of w. Here, $w[i : j]$ is the substring from the $i + 1$-st up to and including the j-th symbol of w.

Now each such consumption is precisely a terminating computation for that stack symbol; and we noted before that Sub contains the terminator positions for all stack elements. In other words, any pair of positions (i_p, i_{p+1}) of the above substring sequence will correspond to a pair $(\mathrm{pos}(x), \mathrm{pos}(z))$ with $\mathrm{Sub}(x, z) = \mathrm{true}$ if the pop move from z consumes no input symbol and to $(\mathrm{pos}(x), \mathrm{pos}(z) + 1)$ if it does.

The task of suffix recognition requires that we can deal with the infinite set of stacks built up by all possible prefixes of the language. That set can, however, be described by the possible paths through a finite directed graph with nodes labelled by stack symbols and the states reached right after they are pushed and edges connecting such pairs of stack symbols and states. It would also be possible to express this fact in terms of finite automata over the stack alphabet as shown in [14]. The graph will here be represented by the relation On, over the set of pairs of stack symbols and states: $\mathrm{On}((s, a), (t, b))$ is to mean that stack symbol a can be placed on stack symbol b such that the state reached after a gets pushed is s, and t was the state reached after b got pushed.

The computation of On is performed once and for all prior to the recognition process. It can be easily obtained from a relation Ret on States $\times$ States $\times$ Symbols. This relation is similar to Sub in that subcomputations are represented where the stack height at the end has returned to what it was at the beginning. A notable difference with Sub is however that the aspect of PDA moves dealing with input symbols is completely ignored, in order to analyse all computations that could take place during scanning of the unknown prefix

v. Its inductive definition is:

- for each s and a, $\mathrm{Ret}(s, s, a)$;
- if there is a push move on s_1 and a that enters state t_1 and pushes b, and if $\mathrm{Ret}(t_1, t_2, b)$, and if there is a pop move on t_2 and b that enters state s_2, then $\mathrm{Ret}(s_1, s_2, a)$;
- if $\mathrm{Ret}(s_1, s_2, a)$ and $\mathrm{Ret}(s_2, s_3, a)$ then $\mathrm{Ret}(s_1, s_3, a)$.

Assuming the PDA is reduced, we now define: $\mathrm{On}((s, a), (t, b))$ if and only if

- there is a push move that enters state t and pushes b, or t is the initial state and b is the initial stack symbol; and
- for some t', $\mathrm{Ret}(t, t', b)$ and there is a push move on t' and b that enters state s and pushes a.

5 Construction of directed graph for suffix recognition

We are now able to present our construction of the graph needed to recognize the suffix, which merges the monotonous progress through the input in terms of Sub with progress through the possibly cyclic graph represented by On. Define a digraph with the $\mathcal{O}(n)$ nodes labelled by elements of the Cartesian product States $\times$ States $\times$ Positions $\times$ Symbols. A node (s', s, i, a) represents that symbol a was pushed during reading of the unknown prefix, and the state right after the push was s', and at input position i, the same symbol a became again exposed as top-of-stack, while the state was s.

The digraph has an edge from (s', s, i, a) to (t', t, j, b) iff $\mathrm{Sub}(x, z)$ for $x = (s, i, a)$, the state of the configuration following z by means of the pop move equals t, the input position reached by that move (by adding 0 or 1) equals j, and $\mathrm{On}((s', a), (t', b))$. This is related to development of the stack as illustrated in Figure 1. It is clear that the MRS procedure can be extended with the task of constructing all such edges, without an increase in the number of processors or in (the order of) the time consumption.

We may assume a perpetual bottom-of-stack symbol c, and assume that acceptance is by final state s_f and empty stack (apart from c). Digraph reachability is now defined with a number of possible 'source' nodes and a single 'sink' node. Any node $(s', s, 0, a)$ can be source if there is a push move that enters state s' and pushes a and $\mathrm{Ret}(s', s, a)$. The sink is (s_0, s_f, n, c). If it is reachable from some source $(s', s, 0, a)$, then there is a sequence $w[i_0 : i_1]$, $w[i_1 : i_2]$, $\ldots$, $w[i_{m-1} : i_m]$, where $i_0 = 0$ and $i_m = n$, with the following properties. Their concatenation equals the input string w, which constitutes a correct suffix of the language L. There is a stack with m symbols, each of which is removed at

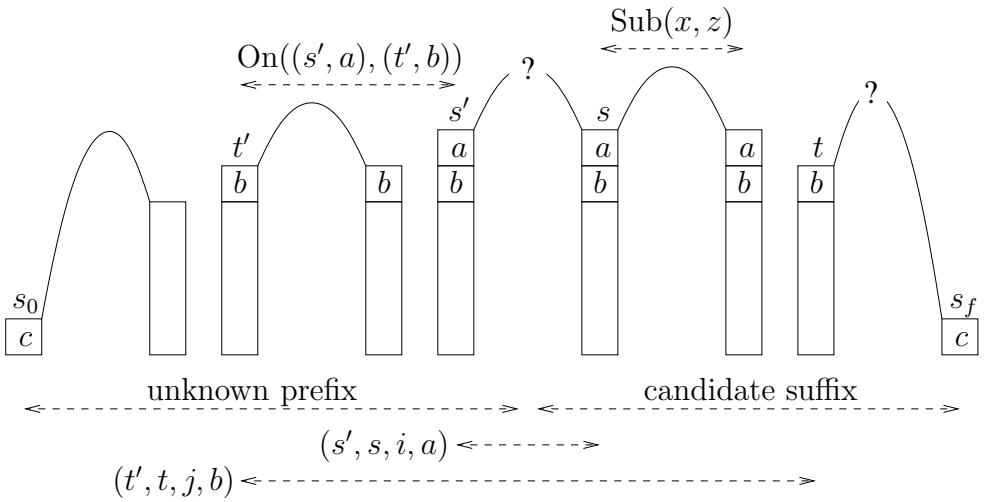

Fig. 1. Development of stack, in relation to table items.

the end of processing of some $w[i_p : i_{p+1}]$, in the manner explained in Section 4. This stack may exist after reading some unknown prefix v that precedes w in some string $vw \in L$. Conversely, if w is a *proper* suffix of a string $vw \in L$, then (s_0, s_f, n, c) must be reachable from some $(s', s, 0, a)$, by an appropriate partition of w into substrings. The special case that $w \in L$, with v empty, is solved already by MRS. We may recognize infixes in addition to suffixes by taking all (s', s, n, a), for any s', s and a, to be sink nodes.

Acknowledgments

Ideas and suggestions provided by anonymous referees are gratefully acknowledged.

References

[1] H. Richter, Noncorrecting syntax error recovery, ACM Transactions on Programming Languages and Systems 7 (3) (1985) 478–489.

[2] J. Bates, A. Lavie, Recognizing substrings of LR(k) languages in linear time, ACM Transactions on Programming Languages and Systems 16 (3) (1994) 1051–1077.

[3] M.-J. Nederhof, E. Bertsch, Linear-time suffix parsing for deterministic languages, Journal of the ACM 43 (3) (1996) 524–554.

[4] W. Rytter, The complexity of two-way pushdown automata and recursive programs, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Springer-Verlag, 1985, pp. 341–356.

[5] M. Chytil, M. Crochemore, B. Monien, W. Rytter, On the parallel recognition of unambiguous context-free languages, Theoretical Computer Science 81 (1991) 311–316.

[6] W. Rytter, On efficient parallel computations of costs of paths on a grid graph, Information Processing Letters 29 (2) (1988) 71–74.

[7] L. Larmore, W. Rytter, Almost optimal sublinear time parallel recognition algorithms for three subclasses of context free languages, Theoretical Computer Science 197 (1998) 189–201.

[8] W. Rytter, On the complexity of parallel parsing of general context-free languages, Theoretical Computer Science 47 (1986) 315–321.

[9] W. Rytter, Parallel time $O(\log n)$ recognition of unambiguous context-free languages, Information and Computation 73 (1987) 75–86.

[10] P. Klein, J. Reif, Parallel time $O(\log n)$ acceptance of deterministic CFLs on an exclusive-write P-RAM, SIAM Journal on Computing 17 (3) (1988) 463–485.

[11] P. Dymond, W. Ruzzo, Deterministic context-free language recognition, Journal of the ACM 47 (1) (2000) 16–45.

[12] B. Monien, W. Rytter, L. Schäpers, Fast recognition of deterministic cfl's with a smaller number of processors, Theoretical Computer Science 116 (1993) 421–429.

[13] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, 1992.

[14] E. Bertsch, M.-J. Nederhof, Regular closure of deterministic languages, SIAM Journal on Computing 29 (1) (1999) 81–102.