Theoretical Computer Science 120 (1993) 303–310
Elsevier

Note

Parallel on-line parsing in constant time per word

Klaas Sikkel

Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Communicated by M.A. Harrison
Received May 1992
Revised August 1992

Abstract

Sikkel, K., Parallel on-line parsing in constant time per word, Theoretical Computer Science 120 (1993) 303–310.

An on-line parser processes each word as soon as it is typed by the user, without waiting for the end of the sentence. Thus, in an interactive system, a sentence will be parsed almost immediately after the last word has been presented.

The complexity of an on-line parser is determined by the resources needed for the analysis of a *single word*, as it is assumed that previous words have been processed already. Sequential parsing algorithms like CYK or Earley need $O(n^2)$ time for the nth word. A parallel implementation in $O(n)$ time on $O(n)$ processors is straightforward. In this paper a novel parallel on-line parser is presented that needs $O(1)$ time on $O(n^2)$ processors.

1. Introduction

A large class of practical parsing algorithms needs $O(n^3)$ time to parse a sentence of n words. The canonical, earliest example of a cubic time parser is the *Cocke–Younger–Kasami* (CYK) algorithm [12], for grammars in Chomsky normal form. Basically, it is a recognition algorithm, which fills an upper triangular matrix

Correspondence to: K. Sikkel, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands. Email: sikkel@cs.utwente.nl.

$T_{i,j}$ with nonterminal symbols such that $A \in T_{i,j}$ iff $A \Rightarrow^* a_{i+1} \ldots a_j$, a substring of the sentence $a_1 \ldots a_n$; see Fig. 1. A similar algorithm for arbitrary context-free grammars has been defined by Earley [3] and improved by Graham et al. [5].

Parallel implementations of the CYK algorithm using $O(n)$ time on $O(n^2)$ processors were known already in the late sixties [6]. Similarly, $O(n^2)$ time is needed on $O(n)$ processors for the CYK algorithm. Processors can always be traded against time, obviously, by sequentialising a parallel algorithm. The reverse need not be true. A parsing algorithm that uses $O(1)$ time on $O(n^3)$ processors does not exist. The first algorithm that runs in logarithmic time on a polynomial number of processors is given by Ruzzo [8]. He claims (but does not present) a refinement of it using $O(n^6)$ processors. A well-known logarithmic-time algorithm is due to Rytter [9] taking $O(\log^2 n)$ time on $O(n^6)$ processors. An almost identical algorithm has been independently discovered by Brent and Goldschlager [2]. The time complexity can be reduced from $O(\log^2 n)$ to $O(\log n)$ if concurrent write to the same memory location by different processors is allowed.

An *on-line* parser starts working as soon as the user types the first word of a sentence. It is assumed that the parser works faster than the user. A partial parse of the sentence is updated when a new word is entered. In that way, a parse can be completed (almost) immediately after the last word is known. On-line parsing may speed up the response in natural languages interfaces.

The complexity of an on-line parser is the time – or time × number of processors – needed for the analysis of a single word, in particular the last word of the sentence. The CYK algorithm and Earley's algorithm can be trivially implemented as an on-line parser with $O(n)$ processors using $O(n)$ time. Processing one word is equivalent to computing one column of the CYK table, i.e., all entries $T_{i,j}$ with fixed j and i ranging frm 0 to $j-1$. Further parallelisation is not possible for an on-line CYK parser: $T_{i,j}$ cannot be computed before $T_{i+1,j}$ has been completed (see Fig. 1). In this paper an on-line parser is presented that needs $O(1)$ time on $O(n^2)$ processors. It resembles

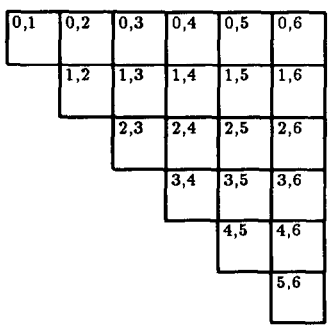

Fig. 1. CYK recognition table for a sentence of 6 words.

Rytter's algorithm, but because of the different nature of on-line parsing, its complexity is much lower.

Following Gibbons and Rytter [4], we use a WRAM as abstract machine model, i.e. a parallel random access machine allowing concurrent write *only if processors writing the same memory location write the same value.* If the tables to be constructed are represented by bit-vectors, i.e., one bit for each possible table entry, the write condition is trivially satisfied, as only 1-bits are written.

After some preliminaries and definitions, in Section 3 an on-line recogniser for grammars in Chomsky normal form is presented. In Section 4 it is sketched briefly how a parser for arbitrary context-free grammars can be constructed, based on the recogniser for CNF grammars.

2. Preliminaries and definitions

A context-free grammar $G=(N,\Sigma,P,S)$ is in Chomsky normal form (CNF) if P contains productions of the form $A\to BC$ and $A\to a$, with $A,B,C\in N$, $a\in\Sigma$. The basic algorithm is defined for grammars in Chomsky normal form.

The algorithm is based on sets of *recognised items.* Like in the CYK algorithm, an item $[i,A,j]$ will be added to the set of recognised items if we have been able to determine the fact $A\Rightarrow^* a_{i+1}\dots a_j$. Items of the form $[i,A,j]$, called *complete* items, comprise a nonterminal symbol and two *place markers.* Unlike CYK, however, we also need *right-incomplete items* $[i,A,j;B]$ denoting collected facts $A\Rightarrow^* a_{i+1}\dots a_jB$, with $a_{i+1}\dots a_j$ part of the sentence and B a nonterminal (see Fig. 2).

Definition 2.1. We define the set of complete items Δ and the set of right-incomplete items Γ by

$$\Delta \overset{\text{def}}{=} \{[i,A,j] \mid A\in N,\ 0\leqslant i<j\leqslant n\},$$

$$\Gamma \overset{\text{def}}{=} \{[i,A,j;B] \mid A,B\in N,\ 0\leqslant i<j\leqslant n\},$$

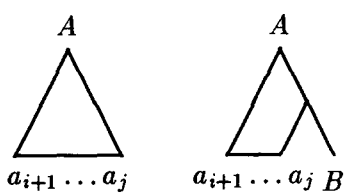

Fig. 2. Graphical representation of a complete item and a right-incomplete item.

where n is the length of the sentence to be recognised. Furthermore, we define the subsets of Δ and Γ with a fixed right place marker:

$$\Delta_j \stackrel{\text{def}}{=} \{[i,A,j] \mid A \in N \wedge 0 \leqslant i < j\},$$

$$\Gamma_j \stackrel{\text{def}}{=} \{[i,A,j;B] \mid A,B \in N \wedge 0 \leqslant i < j\}.$$

Definition 2.2. A complete item $[i,A,j]$ is called *recognisable* if $A \Rightarrow^* a_{i+1} \ldots a_j$; a right-incomplete item $[i,A,j;B]$ is called *recognisable* if $A \Rightarrow^* a_{i+1} \ldots a_j B$.

3. The recognition algorithm

The purpose of the algorithm is to compute item sets $I \subset \Delta$ and $J \subset \Gamma$ such that all recognisable items (and only those) are in I and J. A sentence $a_1 \ldots a_n$ is correct, obviously, iff $[0,S,n] \in I$ when the algorithm has finished. We use subsets of I and J with a fixed right place marker, $I_j \stackrel{\text{def}}{=} I \cup \Delta_j$ and $J_j \stackrel{\text{def}}{=} I \cup \Gamma_j$. When the jth word is read, I_j and J_j are computed from $I_1, \ldots, I_{j-1}$, $J_1, \ldots, J_{j-1}$ and the terminal a_j.

For each step in the algorithm we define a function that can be computed in constant time using up to $O(n^2)$ processors on a WRAM. In the following definitions we write $\wp(X)$ to denote the power set of an arbitrary set X.

Definition 3.1. The function $init: \Sigma \times \mathbb{N} \to \wp(\Delta)$ is defined by

$$init(a,j) \stackrel{\text{def}}{=} \{[j-1,A,j] \in \Delta \mid A \to a \in P\}.$$

That is, *init* delivers *pre-terminal* items – spanning substrings of one terminal symbol – for a particular word at a particular position in the sentence (see Fig. 3).

Definition 3.2. The function $recognise: \wp(\Gamma) \times \wp(\Delta) \times \mathbb{N} \to \wp(\Delta)$ is defined by

$$recognise(J,I,k) \stackrel{\text{def}}{=} \{[i,A,j] \in \Delta \mid [i,A,k;B] \in J \wedge [k,B,j] \in I\}.$$

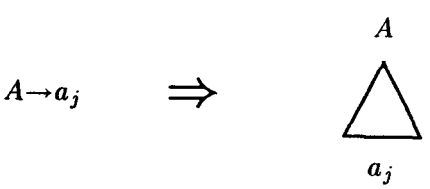

Fig. 3. *init*.

Recognise delivers new items that can be combined from a right-incomplete and a complete item. If $[i, A, k; B]$ and $[k, B, j]$ have been recognised, then clearly $[i, A, j]$ can be recognised as well; see Fig. 4. For technical reasons, the function is defined for a parameter k and free ranging i and j.

Definition 3.3. The function $propose: \wp(\Delta) \to \wp(\Gamma)$ is defined by

$$propose(I) \stackrel{\text{def}}{=} \{[i, A, j; C] \in \Gamma \mid \exists B \in N: [i, B, j] \in I \wedge A \to BC \in P\}.$$

That is, *propose* extends complete items to right-incomplete items. If $[i, B, j] \in I$ and $A \to BC \in P$, then $[i, A, j; C]$ is proposed as an item that, potentially, can be completed later on (see Fig. 5).

Definition 3.4. The function $combine: \wp(\Gamma) \times \mathbb{N} \to \wp(\Gamma)$ is defined by

$$combine(J, j) \stackrel{\text{def}}{=} \{[i, A, j; C] \in \Gamma \mid \exists k \in \mathbb{N} \exists B \in N: [i, A, k; B] \in J \wedge [k, B, j; C] \in J\}.$$

Combine combines right-incomplete items $[i, A, k; B]$ and $[k, B, j; C]$ into $[i, A, j; C]$, for a parameter j and free ranging i and k. See Fig. 6.

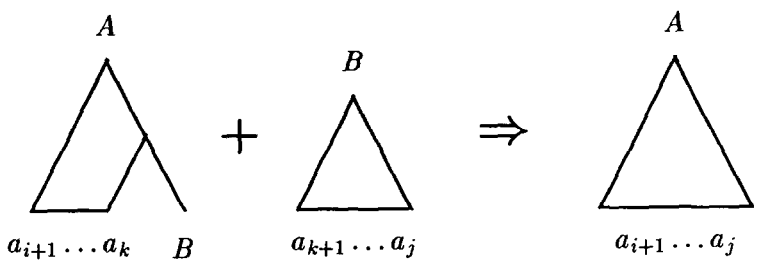

Fig. 4. *recognise.*

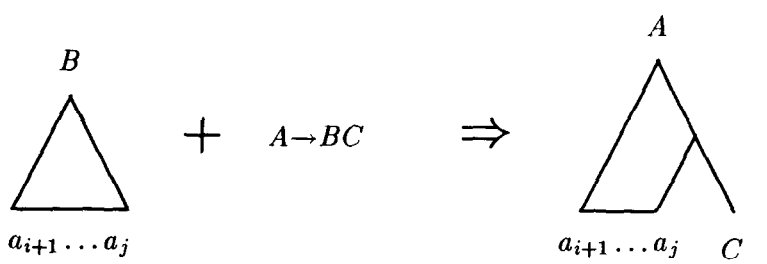

Fig. 5. *propose.*

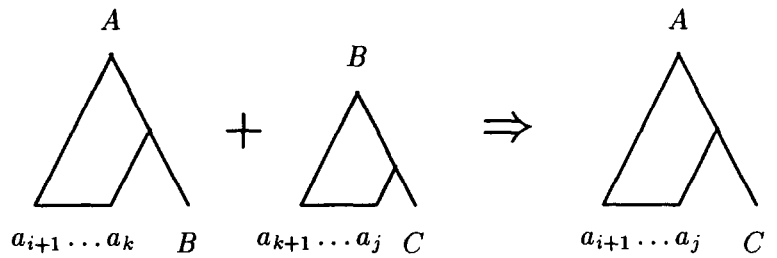

Fig. 6. *combine*.

The algorithm using these functions is as follows:

```
begin
  I:=∅; J:=∅;
  for j:=1 to n do
  begin
    I_j:=init(a_j,j);
    I_j:=I_j∪recognise(J_{j-1},I_j,j-1);
    J_j:=propose(I_j);
    J_j:=J_j∪combine(J,j)
  end
end
```

Note that *combine*(J,j) can be performed in constant time if $O(j^2)$ processors are allocated for each applicable pair of values of i and k. It is possible that several processors recognise an item $[i,A,j;C]$ simultaneously. A set of recognised items can be implemented as a bit string, in which 0 means absence and 1 means presence in the set. Thus, if a parallel write occurs, it is guaranteed that only 1 bits are written. Hence *combine* is implementable on a WRAM in this way. The other functions can be computed in constant time using a linear number of processors on a WRAM.

Theorem 3.5. *After step j in the above algorithm the following statements hold:*
- $[i,A,j] \in I_j$ *if and only if* $[i,A,j]$ *is recognisable,*
- $[i,A,j;B] \in J_j$ *if and only if* $[i,A,j;B]$ *is recognisable.*

Proof. The soundness follows from the definitions of the functions *init*, *recognise*, *propose* and *combine*. We will prove the completeness by simultaneous induction of both statements on j.

Assume that the theorem holds for any k, $1 \leqslant k < j$. We show (I) that $[i,A,j] \in I_j$ if $[i,A,j]$ is recognisable and (II) that $[i,A,j;B] \in J_j$ if $[i,A,j;B]$ is recognisable.

(I) Let $[i,A,j]$ be recognisable. If $i=j-1$ then $[i,A,j] \in init(a_j,j)$. Otherwise $i<j-1$. In that case a derivation

$$A \Rightarrow^* a_{i+1} \ldots a_{j-1} B \Rightarrow a_{i+1} \ldots a_j$$

exists for some $[j-1, B, j] \in init(a_j, j)$. By induction, $[i, A, j-1; B] \in J_{j-1}$, hence $[i, A, j] \in recognise(J_{j-1}, init(a_j, j), j-1)$. Thus $[i, A, j] \in I_j$ after step j.
(II) Let $[i, A, j; C]$ be recognisable, i.e., there is a derivation $A \Rightarrow^* a_{i+1} \ldots a_j C$. Somewhere in this derivation, C must have been produced by some $D \to BC \in P$. Hence, a derivation

$$A \Rightarrow^* a_{i+1} \ldots a_k D \Rightarrow a_{i+1} \ldots a_k BC \Rightarrow^* a_{i+1} \ldots a_j C$$

exists for some (undetermined) k with $i \leqslant k < j$. By (I), we find $[k, B, j] \in I_j$, which implies $[k, D, j; C] \in propose(I_j)$. If $i = k$ then $A = D$ and $[i, A, j; C] \in propose(I_j)$. Otherwise $i < k < j$ and, by induction, $[i, A, k; D] \in J_k$, hence $[i, A, j; C] \in combine(J_k \cup propose(I_j), j)$. Thus, in both cases $[i, A, j; C] \in J_j$ after step j. □

The recognition algorithm uses $O(n^2)$ processors and $O(n^2)$ memory while $O(1)$ time is needed per word.

4. On-line parsing

In this section it is sketched briefly how a parsing algorithm for arbitrary context-free grammars is constructed, based on the recognition algorithm for CNF grammars. See [10] for more details.

In natural language parsing a syntactic parser should deliver all different parses of a sentence. Usually this is done in the form of a *shared forest* [11, 1], a structure in which different parse trees share common parts. The recogniser is extended to a parser as follows. In addition to the sets of recognised items I and J, a set F representing the shared forest is computed. F contains entries of the form $[i, A, j; p, k]$, with $A \in N$, $p \in P$ and $i, j, k \in \mathbb{N}$. The left part of a forest entry is a recognised complete item, the right part indicates *how* it was recognised. If $[i, A, j; A \to BC, k] \in F$, then $[i, A, j]$ was derived from $[B, i, k]$ and $[C, k, j]$. Productions $A \to a$ are represented by forest entries $[j-1, A, j, A \to a]$.

Each nonleaf node of each parse tree, together with its outgoing edges, is represented by some element of F. Ambiguities are represented by multiple entries $[A, i, j; p, k] \in F$ for a single item $[A, i, j] \in I$. These correspond to *packed nodes* as they are usually called, i.e., nodes in the graph with a *set of pairs* of successors, rather than a single pair of successors.

It should be noted that the shared forest contains nodes that are not part of a parse. *All* sub-parses $A \Rightarrow^* a_{i+1} \ldots a_j$ are represented, also those for which $S \Rightarrow^* a_1 \ldots a_i A a_{j+1} \ldots a_n$ does not hold. In natural language parsing, this is not seen as a problem. The parse forest contains "junk" anyway, as many syntactically valid parses are semantically incorrect.

The algorithm of the previous section can be extended straightforwardly with a function *parse* that computes subsets F_j of F (with fixed second place marker j) in constant time for each j, using $O(n^2)$ processors. Memory usage increases from $O(n^2)$

to $O(n^3)$ while the complexity bounds for time and number of processors remain the same.

Moreover, the algorithm – like any parser for CNF grammars – can be extended to arbitrary context-free grammars using a *bilinear cover* [7]. Productions with three or more non-terminals in the right-hand side are covered by a *sequence* of bilinear productions. With some additional effort, ε-productions can be handled without increasing the complexity bounds.

5. Conclusion

A novel on-line parsing algorithm is presented, related to Rytter's recognition algorithm. It runs in constant time on a WRAM, using $O(n^2)$ processors. Memory space is $O(n^2)$ for recognition and $O(n^3)$ for parsing.

Acknowledgment

I am grateful to Anton Nijholt and an anonymous referee for constructive comments on preliminary versions of this paper.

References

[1] S. Billot and B. Lang, The structure of shared forests in ambiguous parsing, in: *Proc. 27th Ann. Meeting of the Assoc. for Computational Linguistics*, Vancouver (1989) 143–151.

[2] R.P. Brent and L.M. Goldschlager, A parallel algorithm for context-free parsing, *Austral. Comput. Sci. Commun.* **6** (1984) 7.1–7.10.

[3] J. Earley, An efficient context-free parsing algorithm, Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburgh, 1968.

[4] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms* (Cambridge Univ. Press, Cambridge, 1988).

[5] S.L. Graham, M.A. Harrison and W.L. Ruzzo, An improved context-free recognizer, *ACM Trans. Prog. Lang. Syst.* **2** (1980) 415–462.

[6] S.R. Kosaraju, Computations on iterative automata, Ph.D. Thesis, Univ. of Pennsylvania, Philadelphia, 1969.

[7] R. Leermakers, How to cover a grammar? in: *Proc. 27th Ann. Meeting of the Assoc. for Computational Linguistics*, Vancouver (1989) 135–142.

[8] W.L. Ruzzo, On uniform circuit complexity, *J. Comput. System Sci.* **22** (1981) 365–383.

[9] W. Rytter, On the recognition of context-free languages, in: *Proc. 5th Symp. on Fundamentals of Computation Theory*, Lecture Notes in Computer Science, Vol. 208 (Springer, Berlin, 1985) 318–325.

[10] K. Sikkel, On-line parsing in constant time, Memoranda Informatica 91-64, Univ. of Twente, Enschede, The Netherlands, 1991.

[11] M. Tomita, *Efficient Parsing for Natural Language* (Kluwer, Boston, MA, 1985).

[12] D.H. Younger, Recognition and parsing of context-free languages in time n^3, *Inform. and Control* **10** (1967) 189–208.