Parallel Parsing Strategies in Natural Language Processing

Anton Nijholt

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

ABSTRACT

We present a concise survey of approaches to the context-free parsing problem of natural languages in parallel environments. The discussion includes parsing schemes which use more than one traditional parser, schemes where separate processes are assigned to the 'non-deterministic' choices during parsing, schemes where the number of processes depends on the length of the sentence being parsed, and schemes where the number of processes depends on the grammar size rather than on the input length. In addition we discuss a connectionist approach to the parsing problem.

1. Introduction

In the early 1970's papers appeared in which ideas on parallel compiling for programming languages and parallel executing of computer programs were investigated. In these papers parallel lexical analysis, syntactic analysis (parsing) and code generation were discussed. At that time various multi-processor computers were introduced (CDC 6500, 7600, STAR, ILLIAC IV, etc.) and the first attempts were made to construct compilers which used more than one processor when compiling programs. Slowly, with the advent of new parallel architectures and the ubiquitous application of VLSI, interest increased and presently research on parallel compiling and executing is widespread. Although more slowly, a similar change of orientation occurred in the field of natural language processing. However, unlike the compiler construction environment with its generally accepted theories, in natural language processing no generally advocated – and accepted – theory of natural language analysis and understanding is available. Therefore it is not only the desire to exploit parallelism for the improvement of speed but it is also the assumption that human sentence processing is of an inherently parallel nature which makes computer linguists and cognitive scientists turn to parallel approaches for their problems.

Parallel parsing methods have been introduced in the areas of theoretical computer science, compiler construction and natural language processing. In the area of compiler construction these methods sometimes refer to the properties of programming languages, e.g. the existence of special keywords, the frequent occurrence of arithmetic expressions, etc. Sometimes the parsing methods that have been introduced were closely related to existing and well-known serial parsing methods, such as LL-, LR-, and precedence parsing. Parallel parsing has often been looked upon as deterministic parsing of sentences with more than just a single serial parser. However, with the massively parallel architectures that have been designed and constructed, together with the possibility to design special-purpose chips for parsing and compiling in mind, also the well-known methods for general context-free parsing have been re-investigated in order to see whether they allow parallel implementations. Typical results in this area are $O(n)$-time parallel parsing algorithms based on the Earley or the Cocke-Younger-Kasami parsing methods. In order to study complexity results for parallel recognition and parsing of context-free languages theoretical computer scientists have introduced parallel machine models and special subclasses of the context-free languages (bracket languages, input-driven languages). Methods that have been introduced in this area aim at obtaining lower bounds for time and/or space complexity and are not necessarily useful from a more practical point of view. A typical result in this area tells us that context-free language recognition can be done in $O(\log^2 n)$ time using n^6 processors, where n is the length of the input string.

In the area of natural language processing many kinds of approaches and results can be distinguished. While some researchers aim at cognitive simulation, others are satisfied with high performance language systems. The first-mentioned researchers may ultimately ask for numbers of processors and connections between processors that approximate the number of neurons and interconnections in the human brain. They model human language processing with connectionist models and therefore they are interested in massive parallelism and methods which allow low degradation in the face of local errors. In connectionist and related approaches to parsing and natural language analysis the traditional methods of language analysis are often replaced by strongly interactive distributed processing of word senses, case roles and semantic markers. A more modest use of parallelism may also be useful. For any system which has to understand natural language sentences it is necessary to distinguish different levels of analysis (see e.g. Nijholt[1988], where we distinguish the morphological, the lexical, the syntactic, the semantic, the referential and the behavioral level) and at each level a different kind of knowledge has to be invoked. Therefore we can distinguish different tasks: the application of morphological knowledge, the application of lexical knowledge, etc. It is not necessarily the case that the application of one type of knowledge is under control of the application of any other type of knowledge. These tasks may interact and at times they can be performed simultaneously. Therefore processors which can work in parallel and which can communicate with each other may be assigned to these tasks in order to perform this interplay of multiple sources of knowledge. Finally, and independent of a parallel nature that can be recognized in the domain of language processing, since operating in parallel with a collection of processors can achieve substantial speed-ups, designers and implementers of natural language processing systems will consider the application of available parallel processing power for any task or subtask which allows that application.

In this paper various approaches to the problem of parallel parsing will be surveyed. We will discuss examples of parsing schemes which use more than one traditional parser, schemes where 'non-deterministic' choices during parsing lead to separate processes, schemes where the number of processes depends on the length of the sentence being parsed, and schemes where the number of processes depends on the grammar size rather than on the input length. Our aim is not to give a complete survey of methods that have been introduced in the area of parallel parsing. Rather we present some approaches that use ideas that seem to be characteristic for many of the parallel parsing methods that have been introduced.

2. From One to Many Traditional Serial Parsers

Introduction

As mentioned in the introduction, many algorithms for parallel parsing have been proposed. Concentrating on the ideas that underlie these methods, some of them will be discussed here. For an annotated bibliography containing references to other methods see Nijholt et al[1989]. Since we will frequently refer to *LR-parsing* a few words will be spent on this algorithm. The class of LR-grammars is a subclass of the class of context-free grammars. Each LR-grammar generates a *deterministic* context-free languages and each deterministic context-free language can be generated by an LR-grammar. From an LR-grammar an LR-parser can be constructed. The LR-parser consists of an LR-table and an LR-routine which consults the table to decide the actions that have to be performed on a pushdown stack and on the input. The pushdown stack will contain symbols denoting the *state* of the parser. As an example, consider the following context-free grammar.

1. S $\rightarrow$ NP VP	4. PP $\rightarrow$ *prep NP
2. S $\rightarrow$ S PP	5. VP $\rightarrow$ *v NP
3. NP $\rightarrow$ *det *n	

With the LR-construction method the LR-table of Fig. 1 will be obtained from this grammar. It is assumed that each input string to be parsed will have an endmarker which consists of the $-sign.

An entry in the table of the form 'sh n' indicates the action 'shift state n on the stack and advance the input pointer'; entry 're n' indicates the action 'reduce the stack using rule n'. The

state	*det	*n	*v	*prep	$	NP	PP	VP	S
0	sh3					2			1
1				sh5	acc		4		
2			sh6					7	
3		sh8							
4				re2	re2				
5	sh3					9			
6	sh3					10			
7				re1	re1				
8			re3	re3	re3				
9				re4	re4				
10				re5	re5				

Fig. 1 LR-parsing table for the example grammar.

entry 'acc' indicates that the input string is accepted. The right part of the table is used to decide the state the parser has to enter after a reduce action. In a reduce action states are popped from the stack. The number of states that are popped is equal to the length of the right hand side of the rule that has to be used in the reduction. With the state which becomes the topmost symbol of the stack (0–10) and with the nonterminal of the left hand side of the rule which is used in the reduction (*S*, *NP*, *VP*, or *PP*) the right part of the table tells the parser what state to push next on the stack. In Fig. 2 the usual configuration of an LR-parser is shown.

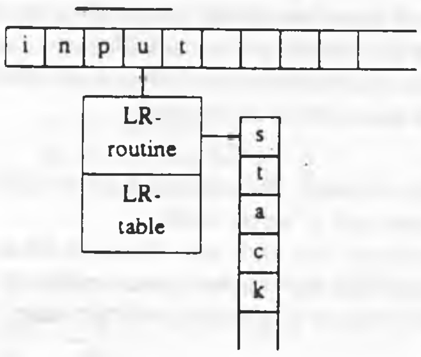

Fig. 2 LR-parser.

More than One Serial Parser

Having more than one processor, why not use two parsers? One of them can be used to process the input from left to right, the other can be used to process the input from right to left. Each parser can be assigned part of the input. When the parsers meet the complete parse tree has to be constructed from the partial parse trees delivered by the two parsers. Obviously, this idea is not new. We can find it in Tseytlin and Yushchenko[1977] and it appears again in Loka[1984]. Let $G = (N, \Sigma, P, S)$ be a context-free grammar. For any string $\alpha \in V^*$ let α^R denote the reversal of α. Let $G^R = (N, \Sigma, P^R, S)$ be the context-free grammar which is obtained from G by defining $P^R = \{i.\, A \rightarrow \alpha^R \mid i.\, A \rightarrow \alpha \in P\}$. It is not difficult to see that, when we start a left-to-right top-down construction of a parse tree with respect to G at the leftmost symbol of a string w and a bottom-up right-to-left construction of a parse tree with respect to G^R at the rightmost symbol of w, then – assuming the grammar is unambiguous – the resulting partial parse trees can be tied together and a parse tree of w with respect to G is obtained. If the grammar is ambiguous all partial trees have to be produced before the correct combinations can be made. Similarly, we can start with a bottom-up parser at the left end of the string and combine it with a top-down parser starting from the right end of the string. Especially when the grammar G allows a combination of a deterministic top-down (or LL-) parser and a deterministic bottom-up (or LR-) parser this might be a useful idea. However, in general we can not expect that if G is an LL-grammar, then G^R is an LR-grammar and conversely.

Rather than having one or two parsers operating at the far left or the far right of the input, we would like to see a number of parsers, where the number depends on the 'parallelism' the input string allows, working along the length of the input string. If there is a natural way to segment a string, then each segment can have its own parser. Examples of this strategy are the methods described in Lincoln[1970], Mickunas and Schell[1975], Fischer[1975], Carlisle and Friesen[1985] and Lozinskii and Nirenburg[1986]. Here we confine ourselves to an explanation of Fischer's method. Fischer introduces 'synchronous parsing machines' (SPM) that LR-parse part of the input string. Each of the SPM's is a serial LR-parser which is able to parse any sentence of the grammar in the usual way from left to right. However, at least in theory, Fischer's method allows any symbol in the input string as the starting point of each SPM. For practical applications one may think of starting at keywords denoting the start of a procedure, a block, or even a statement. One obvious problem that emerges is, when we let a serial LR-parser start somewhere in the input string, in what state should it start? The solution is to let each SPM carry a set of states, guaranteed to include the correct one. In addition, for each of these states the SPM carries a pushdown stack on which the next actions are to be performed. An outline of the parsing algorithm follows.

For convenience we assume that the LR-parser is an LR(0) parser. No look-ahead is necessary to decide a shift or a reduce action. In the algorithm M denotes the LR-parsing table and for any state s, $R(s)$ denotes the set consisting of the rule which has to be used in making a reduction in state s. By definition, $R(s) = \{0\}$ if no reduction has to be made in state s.

(1) *Initialization.*
Start one SPM at the far left of the input string. This SPM has a single stack and it only contains s_0, the initial state. Start a number of other SPM's. Suppose we want to start an SPM immediately to the left of some symbol a. In the LR-parse table M we can find which states have a non-empty entry for symbol a. For each of these states the SPM which will be started, possesses a stack containing this state only. Hence, the SPM is started with just those states that can validly scan the next symbol in the string.

(2) *Scan the next symbol.*
Let a be the symbol to be scanned. For each stack of the SPM, if state s is on top, then
(a) if $M(s, a) = \text{sh } s'$, then push s' on the stack;
(b) if $M(s, a) = \emptyset$, then delete this stack from the set of stacks this SPM carries.
In the latter case the stack has been shown to be invalid. While scanning the next input symbols the number of stacks that an SPM carries will decrease.

(3) *Reduce?*
Let $Q = \{s_1, \cdots, s_n\}$ be the set of top states of the stacks of the SPM under consideration. Define

$$R(Q) = \bigcup_{s \in Q} R(s).$$

(a) if $R(Q) = \{0\}$, then go to step (2); in this case the top states of the stacks agree that no reduction is indicated;
(b) if $R(Q) = \{i\}$, $i \neq 0$, and $i = A \rightarrow \gamma_i$, then, if the stacks of the SPM are deep enough to pop off $|\gamma_i|$ states and not be empty, then do reduction i;
(c) otherwise, if we have insufficient stack depth or not all top states agree on the same reduction, we stop this SPM (for the time being) and, if possible, we start a new SPM to the immediate right.

An SPM which has been stopped can be restarted. If an SPM is about to scan a symbol already scanned by an SPM to its immediate right, then a merge of the two SPM's will be attempted. The following two situations have to be distinguished:

- If the left SPM contains a single stack with top state s, then s is the correct state to be in and we can select from the stacks of the right SPM the stack with bottom state s. Pop s from the left stack and then concatenate the two. All other stacks can be discarded and the newly obtained SPM can continue parsing.
- If the left SPM contains more than one stack, then it is stopped. It has to wait until it is restarted by an SPM to its left. Notice that the leftmost SPM always has one stack and it will always have sufficient stack depth. Therefore there will always be an SPM coming from the left which can restart a waiting SPM.

In step (3c) we started a new SPM immediate to the right of the stopped SPM. What set of states and associated stacks should it be started in? We cannot, as was done in the initialization, simply take those states which allow a scan of the next input symbol. To the left of this new SPM reductions may have been done (or will be done) and therefore other states should be considered in order to guarantee that the correct state is included. Hence, if in step (3) $|R(Q)| > 1$, then for each s in Q, provided $R(s) = \{0\}$, we add s to the set of states of the new SPM and in case $R(s) = \{i\}$ we add to the set of states that have to be carried by the new SPM also the states that can become topmost after a reduction using production rule i (perhaps followed by other reductions).

This concludes our explanation of Fischer's method. For more details and extensions of these ideas the reader is referred to Fischer[1975].

'Solving' Parsing Conflicts by Parallelism?

To allow more efficient parsing methods restrictions on the class of general context-free grammars have been introduced. These restrictions have led to, among others, the classes of LL-, LR- and precedence grammars and associated LL-, LR- and precedence parsing techniques. The LR-technique uses, as discussed in the previous section, an LR-parsing table which is constructed from the LR-grammar.

If the grammar from which the table is constructed is not an LR-grammar, then the table will contain conflict entries. In case of a conflict entry the parser has to choose. One decision may turn out to be wrong or both (or more) possibilities may be correct but only one may be chosen. The entry may allow reduction of a production rule but at the same time it may allow shifting of the next input symbol onto the stack. A conflict entry may also allow reductions according to different production rules. Consider the following example grammar G:

1. S → NP VP	5. NP → NP PP
2. S → S PP	6. PP → *prep NP
3. NP → *n	7. VP → *v NP
4. NP → *det *n	

The parsing table for this grammar, taken from Tomita[1985], is shown in Fig. 3.

state	*det	*n	*v	*prep	$	NP	PP	VP	S
0	sh3	sh4				2			1
1				sh6	acc		5		
2			sh7	sh6			9	8	
3		sh10							
4			re3	re3	re3				
5				re2	re2				
6	sh3	sh4				11			
7	sh3	sh4				12			
8				re1	re1				
9			re5	re5	re5				
10			re4	re4	re4				
11			re6	re6, sh6	re6		9		
12				re7, sh6	re7		9		

Fig. 3 LR-parsing table for grammar G.

Tomita's answer to the problem of LR-parsing of general context-free grammars is 'pseudo-parallelism'. Each time during parsing the parser encounters a multiple entry, the parsing process is split into as many processes as there are entries. Splitting is done by replicating the stack as many times as necessary and then continue parsing with the actions of the entry separately. The processes are 'synchronized' on the shift action. Any process that encounters a shift action waits until the other processes also encounter a shift action. Therefore all processes look at the same input word of the sentence.

Obviously, this LR-directed breadth-first parsing may lead to a large number of non-interacting stacks. So it may occur that during parts of a sentence all processes behave in exactly the same way. Both the amount of computation and the amount of space can be reduced considerably by unifying processes by combining their stacks into a so-called 'graph-structured' stack. Tomita does not suggest a parallel implementation of the algorithm. Rather his techniques for improving efficiency are aimed at efficient serial processing of sentences. Nevertheless, we can ask whether a parallel implementation might be useful. Obviously, Tomita's method is not a 'parallel-designed' algorithm. There is a master routine (the LR-parser) which maintains a data structure (the graph-structured stack) and each word that is read by the LR-parser is required for each process (or stack). In a parallel imolementation nothing is gained when we weave a list of stacks into a graph-structured stack. In tact, when this is done, Tomita's method becomes closely related to Earley's method (see section 4) and it seems more natural – although the number of processes may become too large – to consider parallel versions of this algorithm since it is not restricted in advance by the use of a stack. When we want to stay close to Tomita's ideas, then we rather think of a more straightforward parallel implementation in which each LR conflict causes the creation of a new LR-parser which receives a copy of the stack and a copy of the remaining input (if it is already available) and then continues parsing without ever communicating with the other LR-parsers that work on the same string. On a transputer network, for example, each transputer may act as an LR-parser. However, due to its restrictions on interconnection patterns, sending stacks and strings through the network may become a time-consuming process. When a parser encounters a conflict the network should be searched for a free transputer whereas the stack and the remainder of the input should be passed through the network to this transputer. This will cause other processes to slow down and one may expect that only a limited 'degree of non-LR-ness' will allow an appropriate application of these ideas. Moreover, one may expect serious problems when on-line parsing of the input is required.

3. Translating Grammar Rules into Process Configurations

A simple 'object-oriented' parallel parsing method for ε-free and cycle-free context-free grammars has been introduced by Yonezawa and Ohsawa[1988]. The method resembles the well-known Cocke-Younger-Kasami parsing method, but does not require that the grammars are in Chomsky Normal Form (CNF). Consider again our example grammar G:

1. S → NP VP	5. NP → NP PP
2. S → S PP	6. PP → *prep NP
3. NP → *n	7. VP → *v NP
4. NP → *det *n	

The parsing table for this grammar, taken from Tomita[1985], is shown in This set of rules will be viewed as a network of computing agents working concurrently. Each occurrence of a (pre-)terminal or a nonterminal symbol in the grammar rules corresponds with an agent with modest processing power and internal memory. The agents communicate with one another by passing subtrees of possible parse trees. The topology of the network is obtained as follows. Rule 1 yields the network fragment depicted in Fig. 4.

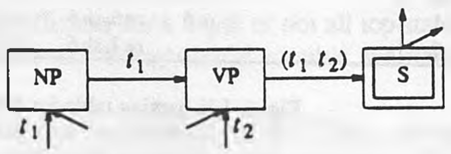

Fig. 4 From rules to configuration.

In the figure we have three agents, one for NP, one for VP and a 'double' agent for S. Suppose the NP-agent has received a subtree t_1. It passes t_1 to the VP-agent. Suppose this agent has received a subtree t_2. It checks whether they can be put together (the 'boundary adjacency test') and, if this test succeeds, it passes $(t_1\ t_2)$ to the S-agent. This agent constructs the parse tree $(S\ (t_1\ t_2))$ and distributes the result to all computing agents in the network which correspond with an occurrence of S in a right hand side of a rule. The complete network for the rules of G is shown in Fig. 5. As can be seen in the network, there is only one of these S-agents. For this agent $(S\ (t_1\ t_2))$ plays the same role as t_1 did for the NP-agent. If the boundary adjacency test is not successful, then the VP-agent stores the trees until it has a pair of trees which satisfies the test.

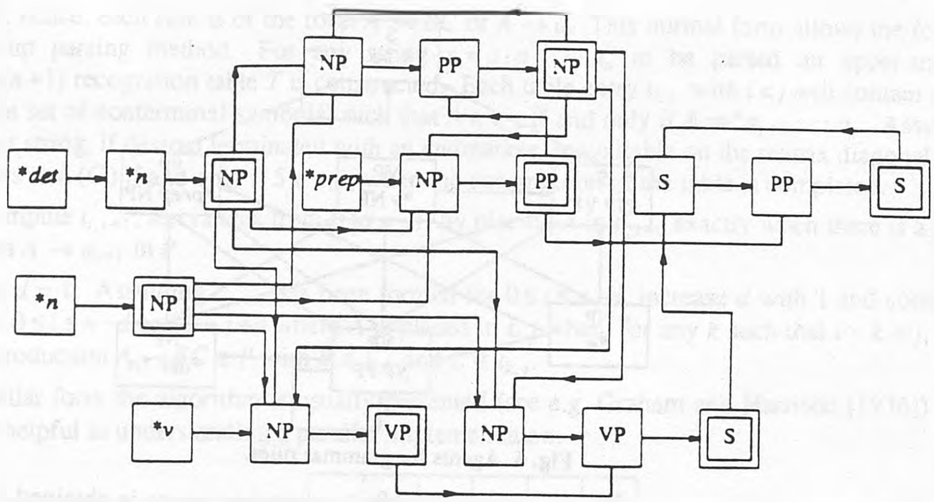

Fig. 5 Computing agents for grammar *G*.

As an example, consider the sentence *The man saw a girl with a telescope*. For this particular sentence we do not want to construct from a subtree t_1 for *a telescope* and from a subtree t_2 for *saw the girl* a subtree for *a telescope saw a girl*, although the rule $S \rightarrow NP\ VP$ permits this construction. Therefore, words to be sent into the network are provided with tags representing positional information and during construction of a subtree this information is inherited from its constituents. For our example sentence the input should look as

(0 1 the)(1 2 man)(2 3 saw)(3 4 a)(4 5 girl)(5 6 with)(6 7 a)(7 8 telescope).

Combination of tokens and trees according to the grammar rules and the positional information can yield a subtree *(3 5 (NP ((*det a)(*n girl))))* but not a subtree in which *(0 1 the)* and *(4 5 girl)* are combined. Each word accompanied with its tags is distributed to the agents for its (pre-)terminal(s) by a *manager agent* which has this information available.

If the context-free grammar which underlies the network is ambiguous, then all possible parse trees for a given input sentence will be constructed. It is possible to pipe-line constructed subtrees to semantic processing agents which filter the trees so that only semantically valid subtrees are distributed to other agents. Another useful extension is the capability to unparse a sentence when the user of a system based on this method erases ('backspaces to') previously typed words. This can be realized by letting the agents send anti-messages that cancel the effects of earlier messages. It should be noted that the parsing of a sentence does not have to be finished before a next sentence is fed into the network. By attaching another tag to the words it becomes possible to distinguish the subtrees from one sentence from those of an other sentence. The method as explained here has been implemented in the object-oriented concurrent language ABCL/1. For the experiment a context-free English grammar which gave rise to 1124 computing agents has been used. Sentences with a length between 10 and 30 words and a parse tree height between .0 and 20 were used for input. Parallelism was simulated by time-slicing. From this simulation it followed that a parse tree is produced from the network in $O(n \times h)$ time, where n is the length of the input string and h is the height of the parse tree. Obviously, simple examples of grammars and their sentences can be given which cause an explosion in the number of adjacency tests and also in the number of subtrees that will be stored without ever being used. Constructs which lead to such explosions do not usually occur in context-free descriptions of natural language.

There are several ways in which the number of computing agents can be reduced. For example, instead of the three double *NP*-agents of Fig. 5 it is possible to use one double *NP*-agent with the same function but with an increase of parse trees that have to be constructed and distributed. The same can be done for the two *S*-agents. A next step is to eliminate all double agents and give their tasks to the agents which correspond with the rightmost symbol of a grammar rule. It is also possible to have one computing agent for each grammar rule. In this way we obtain the configuration of Fig. 6. It will be clear what has to be done by the different agents.

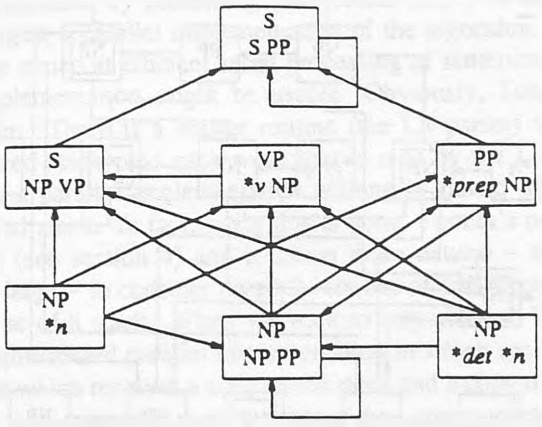

Fig. 6 Agents for grammar rules.

Another configuration with a reduced number of computing agents is obtained if we have an agent for each nonterminal symbol of the grammar. For our example grammar we have four agents, the *S*-, the *NP*-, the *VP*-, and the *PP*-agent. We may also introduce agents for the pre-terminals or even for each word which can occur in an input sentence. We confine ourselves to agents for the nonterminal symbols and discuss their roles. In Fig. 7 we have displayed the configuration of computing agents which will be obtained from the example grammar.

The communication between the agents of this network is as follows.

(1) The *S*-agent sends subtrees with root *S* to itself; it receives subtrees from itself, the *PP*-agent, the *NP*-agent, and the *VP*-agent.

(2) The *NP*-agent sends subtrees with root *NP* to itself, the *S*-agent, the *VP*-agent and the *PP*-agent; it receives subtrees from itself and from the *PP*-agent; moreover, input comes from the manager agent.

(3) The *VP*-agent sends subtrees with root *VP* to the *S*-agent; it receives subtrees from the *NP*-agent; moreover, input comes from the manager agent.

(4) The *PP*-agent sends subtrees with root *PP* to the *S*-agent and to the *NP*-agent; it receives subtrees from the *NP*-agent; moreover, it receives input from the manager agent.

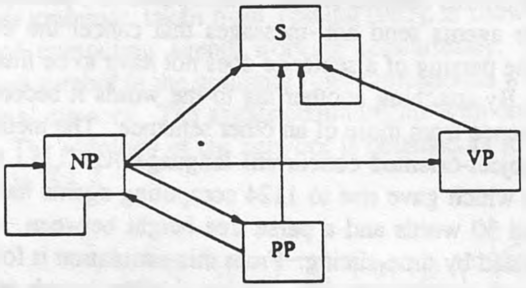

Fig. 7 Agents for nonterminal symbols.

The task of each of these nonterminal agents is to check whether the subtrees it receives can be put together according to the grammar rules with the nonterminal as left-hand side and according to positional information that is carried along with the subtrees. If possible, a tree with the nonterminal as root is constructed, otherwise the agent checks other trees or waits until trees are available.

4. From Sentence Words to Processes

Cocke-Younger-Kasami's Algorithm

Traditional parsing methods for context-free grammars have been re-investigated in order to see whether they can be adapted to a parallel processing view. In Chu and Fu[1982] parallel aspects of the tabular Cocke-Younger-Kasami algorithm have been discussed. The input grammar should be in CNF, hence, each rule is of the form $A \rightarrow BC$ or $A \rightarrow a$. This normal form allows the following bottom-up parsing method. For any string $x = a_1 a_2 \cdots a_n$ to be parsed an upper-triangular $(n+1)\times(n+1)$ recognition table T is constructed. Each table entry $t_{i,j}$ with $i<j$ will contain a subset of N (the set of nonterminal symbols) such that $A \in t_{i,j}$ if and only if $A \Rightarrow^* a_{i+1} \cdots a_j$. Assume that the input string, if desired terminated with an endmarker, is available on the matrix diagonal. String x belongs to $L(G)$ if and only if $S \in t_{0,n}$ when the construction of the table is completed.

(1) Compute $t_{i,i+1}$, as i ranges from 0 to $n-1$, by placing A in $t_{i,i+1}$ exactly when there is a production $A \rightarrow a_{i+1}$ in P.

(2) Set $d = 1$. Assuming $t_{i,i+d}$ has been formed for $0 \le i \le n-d$, increase d with 1 and compute $t_{i,j}$ for $0 \le i \le n-d$ and $j = i+d$ where A is placed in $t_{i,j}$ when, for any k such that $i<k<j$, there is a production $A \rightarrow BC \in P$ with $B \in t_{i,k}$ and $C \in t_{k,j}$.

In a similar form the algorithm is usually presented (see e.g. Graham and Harrison [1976]). Fig. 8 may be helpful in understanding a parallel implementation.

	0,1	0,2	0,3	0,4	0,5
		1,2	1,3	1,4	1,5
			2,3	2,4	2,5
				3,4	3,5
					4,5

Fig. 8 The upper-triangular CYK-table.

Notice that after step (1) the computation of the entries is done diagonal by diagonal until entry $t_{0,n}$ has been completed. For each entry of a diagonal only elements of preceding diagonals are used to compute its value. More specifically, in order to see whether a nonterminal should be included in an element $t_{i,j}$ it is necessary to compare $t_{i,k}$ and $t_{k,j}$, with k between i and j. The amount of storage that is required by this method is proportional to n^2 and the number of elementary operations is proportional to n^3. Unlike Yonezawa and Oshawa's algorithm where positional information needs an explicit representation, here it is in fact available (due to the CNF of the grammar) in the indices of the table elements. For example, in $t_{1,4}$ we can find the nonterminals which generate the substring of the input between positions 1 and 4. The algorithm can be extended in order to produce parse trees.

From the recognition table we can conclude a two-dimensional configuration of processes. For each entry $t_{i,j}$ of the upper-triangular table there is a process $P_{i,j}$ which receives table elements (i.e., sets of nonterminals) from processes $P_{i,j-1}$ and $P_{i+1,j}$. Process $P_{i,j}$ transmits the table elements it receives from $P_{i,j-1}$ to $P_{i,j+1}$ and the elements it receives from $P_{i+1,j}$ to $P_{i-1,j}$. Process $P_{i,j}$ transmits the table element it has constructed to processes $P_{i-1,j}$ and $P_{i,j+1}$. Fig. 9 shows the interconnection structure for $n = 5$. As soon as a table element has been computed, it is sent to its right and upstairs neighbor. Each process should be provided with a coding of the production rules of the grammar. Clearly, each process requires $O(n)$ time. It is not difficult to see that like similar algorithms suitable for VLSI-implementation, e.g. systolic algorithms for matrix multiplication or transitive closure computation (see Guibas et al[1979] and many others) the required parsing time is also $O(n)$. In Chu and Fu[1982] a VLSI design for this algorithm is presented (see also Tan[1983]).

Earley's Algorithm

The second algorithm we discuss in this section is the well-known Earley's method. It is not essentially different from the CYK algorithm. Since the method maintains information in the table entries about the righthand sides of the productions that are being recognized, the condition that the grammar should be in CNF is not necessary. For general context-free grammars Earley parsing takes $O(n^3)$ time. This time can be reduced to $O(n^2)$ or $O(n)$ for special subclasses of context-free

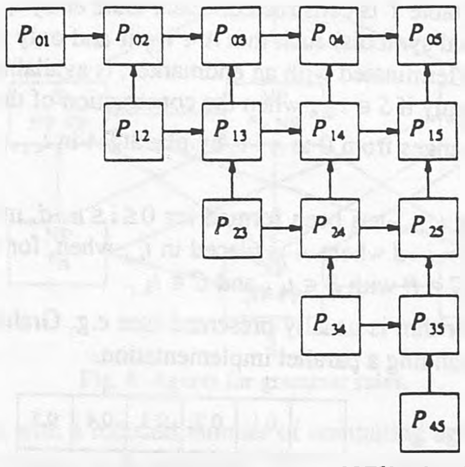

Fig. 9 Process configuration for CYK's algorithm.

grammars. Many versions of Earley's method exist. In Graham and Harrison[1976] the following tabular version can be found. For any string $x = a_1 a_2 \cdots a_n$ to be parsed an upper-triangular $(n+1)\times(n+1)$ recognition table T is constructed. Each table entry $t_{i,j}$ will contain a set of items, i.e., a set of elements of the form $A \rightarrow \alpha \cdot \beta$ (a dotted rule), where $A \rightarrow \alpha\beta$ is a production rule from the grammar and the dot $\cdot$ is a symbol not in $N \cup \Sigma$. The computation of the table entries goes column by column. The following two functions will be useful. Function PREDICT:$N \rightarrow 2^D$, where $D = \{A \rightarrow \alpha \cdot \beta \mid A \rightarrow \alpha\beta \in P\}$, is defined as

$$\text{PREDICT}(A) = \{B \rightarrow \alpha \cdot \beta \mid B \rightarrow \alpha\beta \in P,\ \alpha \Rightarrow^* \varepsilon \text{ and } \exists\ \gamma \in V^* \text{ with } A \Rightarrow^* B\gamma\}.$$

Function PRED:$2^N \rightarrow 2^D$ is defined as

$$\text{PRED}(X) = \bigcup_{A \in X} \text{PREDICT}(A).$$

Initially, $t_{0,0} = \text{PRED}(\{S\})$ and all other table entries are empty. Suppose we want to compute the elements of column j, $j > 0$. In order to compute $t_{i,j}$ with $i \neq j$ assume that all elements of the columns of the upper-triangular table to the left of column j have already been computed and in column j the elements $t_{k,j}$ for $i < k < j$ have been computed.

(1) Add $B \rightarrow \alpha a \beta \cdot \gamma$ to $t_{i,j}$ if $B \rightarrow \alpha \cdot a\beta\gamma \in t_{i,j-1}$, $a = a_j$ and $\beta \Rightarrow^* \varepsilon$.

(2) Add $B \rightarrow \alpha A \beta \cdot \gamma$ to $t_{i,j}$, if, for any k such that $i < k < j$, $B \rightarrow \alpha \cdot A\beta\gamma \in t_{i,k}$, $A \rightarrow \omega \cdot \in t_{k,j}$ and $\beta \Rightarrow^* \varepsilon$.

(3) Add $B \rightarrow \alpha A \beta \cdot \gamma$ to $t_{i,j}$ if $B \rightarrow \alpha \cdot A\beta\gamma \in t_{i,i}$, $\beta \Rightarrow^* \varepsilon$ and there exists $C \in N$ such that $A \Rightarrow^* C$ and $C \rightarrow \omega \cdot \in t_{i,j}$.

After all elements $t_{i,j}$ with $0 \leq i \leq j-1$ of column j have been computed then it is possible to compute $t_{j,j}$.

(4) Let $X_j = \{A \in N \mid B \rightarrow \alpha \cdot A\beta \in t_{i,j}, 0 \leq i \leq j-1\}$. Then $t_{j,j} = \text{PRED}(X_j)$.

It is not difficult to see that $A \rightarrow \alpha \cdot \beta \in t_{i,j}$ if and only if there exists $\gamma \in V^*$ such that $S \Rightarrow^* a_1 \cdots a_i A \gamma$ and $\alpha \Rightarrow^* a_{i+1} \cdots a_j$. Hence, in $t_{0,n}$ we can read whether the sentence was correct. The algorithm can be extended in order to produce parse trees.†

Various parallel implementations of Earley's algorithm have been suggested in the literature (see e.g. Chiang and Fu[1982], Tan[1983] and Sijstermans[1986]). The algorithms differ mainly in details on the handling of ε-rules, preprocessing, the representation of data and circuit and layout design. The main problem in a parallel implementation of the previous algorithm is the computation of the diagonal elements $t_{i,i}$, for $0 \leq i \leq n$. The solution is simple. Initially all elements $t_{i,i}$, $0 \leq i \leq n$,

† When Earley's algorithm was introduced, it was compared with the exponential time methods in which successively every path was followed whenever a non-deterministic choice occurred. Since in Earley's algorithm a 'simultaneous' following of paths can be recognized, it was sometimes considered as a parallel implementation of the earlier depth-first algorithms (see e.g. Lang[1971]).

are set equal to PREDICT(N), where N is the set of nonterminal symbols. The other entries are defined according to the steps (1), (2) and (3). As a consequence, we now have $A \rightarrow \alpha \cdot \beta \in t_{i,j}$ if and only if $\alpha \Rightarrow^* a_{i+1} \cdots a_j$. In spite of weakening the conditions on the contents of the table entries the completed table can still be used to determine whether an input sentence was correct. Moreover, computation of the elements can be done diagonal by diagonal, similar to the CYK algorithm.

(1) Set $t_{i,i}$ equal to PREDICT(N), $0 \leq i \leq n$.

(2) Set $d = 0$. Assuming $t_{i,i+d}$ has been formed for $0 \leq i \leq n-d$, increase d with 1 and compute $t_{i,j}$ for $0 \leq i \leq n-d$ and $j = i+d$ according to:

(2.1) Add $B \rightarrow \alpha a \beta \cdot \gamma$ to $t_{i,j}$ if $B \rightarrow \alpha \cdot a \beta \gamma \in t_{i,j-1}$, $a = a_j$ and $\beta \Rightarrow^* \varepsilon$.

(2.2) Add $B \rightarrow \alpha A \beta \cdot \gamma$ to $t_{i,j}$ if, for any k such that $i < k < j$, $B \rightarrow \alpha \cdot A \beta \gamma \in t_{i,k}$, $A \rightarrow \omega \cdot \in t_{k,j}$ and $\beta \Rightarrow^* \varepsilon$.

(2.3) Add $B \rightarrow \alpha A \beta \cdot \gamma$ to $t_{i,j}$ if $B \rightarrow \alpha \cdot A \beta \gamma \in t_{i,i}$, $\beta \Rightarrow^* \varepsilon$ and there exists $C \in N$ such that $A \Rightarrow^* C$ and $C \rightarrow \omega \cdot \in t_{i,j}$.

VLSI designs or process configurations which implement this algorithm in such a way that it takes $O(n)$ time (with $O(n^2)$ cells or processes can be found in Chiang and Fu[1982], Tan[1983] and Sijstermans[1986] (see also Fig. 9 and its explanation).

5. Connectionist Parsing Algorithms

Only few authors have considered parsing in connectionist networks. It is possible to distinguish a dynamic programming approach based on the CYK algorithm (Fanty[1985]), a Boltzmann machine approach (Selman and Hirst[1985,1987]) and an interactive relaxation approach (Howells[1980]). We confine ourselves to an explanation of Fanty's method since it fits rather naturally in the framework of parsing strategies we have considered in the previous sections. A connectionist Earley parsing algorithm can be found in the full version of the present paper.

Fanty's strategy is that of the CYK parser. The nodes that will be part of the connectionist network are organized according to the positions of the entries of the upper-triangular recognition table. For convenience we first assume that the grammar is in CNF. The table's diagonal will be used for representing the input symbols. This representation will be explained later. For each nonterminal symbol each entry in the table which is not on the diagonal will represent a configuration of nodes. These nodes allow top-down and bottom-up passing of activity. We first explain the bottom-up pass. Consider a particular entry, say $t_{i,j}$ with $j-i \geq 2$, of the upper-triangular matrix. In the traditional algorithm a nonterminal symbol X is added to the set of nonterminal symbols associated with the entry if there are symbols $Y \in t_{i,k}$ and $Z \in t_{k,j}$ such that $X \rightarrow YZ$ is in P. In the connectionist adaptation of the algorithm we already have a node for each nonterminal symbol in entry $t_{i,j}$. Therefore, rather than adding a symbol, here node X at position $t_{i,j}$ is made active if node Y at position $t_{i,k}$ and node Z at position $t_{k,j}$ are active. In general there will be more ways to have a realization of the production $X \rightarrow YZ$ at position $t_{i,j}$. For example, a node for X at entry $t_{1,5}$ can be made active for a production $X \rightarrow YZ$ if there is an active node for Y at $t_{1,2}$ and for Z at $t_{2,5}$, or for Y at $t_{1,3}$ and for Z at $t_{3,5}$, or for Y at $t_{1,4}$ and for Z at $t_{4,5}$. This separation is realized with the help of match nodes in the configuration of each entry of the table. The use of match nodes is illustrated in Fig. 10 for a node for X at position t_{15} of a CYK-table. Here we have shown the three match nodes, one for each possible realization of $X \rightarrow YZ$, for this node at this particular position. For these match nodes to become active all of their inputs must be on. The node for X becomes active when at least one of its inputs (coming from its match nodes) is on. In the figure only match nodes for separate realizations of the same production are included. Obviously, match nodes should also be included at this position for all possible realizations of the other productions with lefthand side X. In this way all the inputs that can make the node for X at this particular position active can be received in a proper way. Observe that if during the recognition of a sentence in an entry more than one match node for a nonterminal is active then the sentence is ambiguous.

In our explanation the assumption $j-i \geq 2$ for entry $t_{i,j}$ was made. Since the grammar is in CNF we have realizations of productions of the form $X \rightarrow a$ in the entries $t_{i,j}$ with $j-i = 1$. In these entries no match nodes are needed since in each entry there can be only one realization of a production with a given lefthand side. We assume that there is a node for each terminal symbol in each

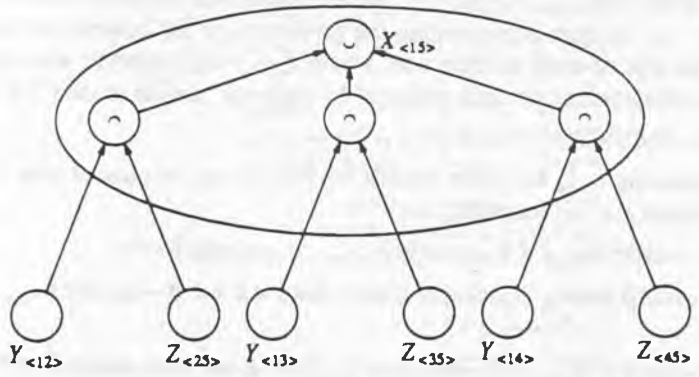

Fig. 10 Bottom-up passing of activity.

position at the diagonal of the matrix. Parsing starts by activating the nodes which correspond with the input symbols. Then activation passes bottom-up through the network, first with realizations of productions of the form $X \to a$, next with realizations of productions of the form $X \to YZ$. The input is accepted as soon as the node for the start symbol in the topmost entry of the column of the last input symbol becomes active.

Until now we have discussed a network which accepts (or rejects) an input string. In order to obtain a representation of the parse tree or parse trees a second, top-down, pass of activity is necessary. To perform this top-down pass we assume that each node mentioned so far consists of a bottom-up and a top-down unit. The bottom-up units are used as explained above. In Fig. 11 both bottom-up and top-down passing of activity is illustrated in a configuration of nodes for an entry $t_{i,j}$ with $j-i \geq 2$. Each node is represented as consisting of a leftmost or bottom-up and a rightmost or top-down unit.

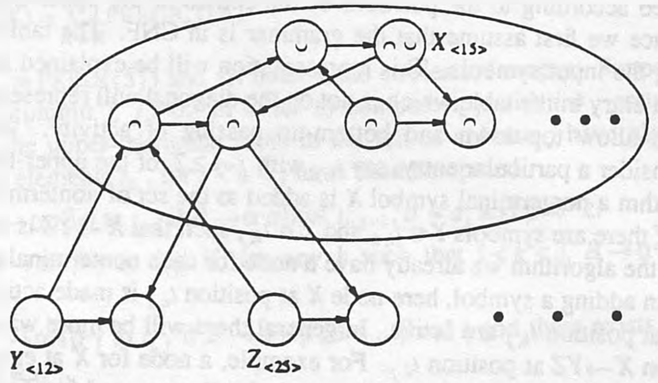

Fig. 11 Top-down and bottom-up passing of activity.

A top-down unit becomes active when it receives input from its bottom-up counterpart and at least one external source. In order to activate the top-down unit of the node for the start symbol in the upper right corner of the table we assume that it receives input from its bottom-up counterpart and from the node at position $t_{n,n}$, where n is the length of the input, which is used to represent endmarker \$ of the input and which is made active when parsing starts. Hence, when the input is recognized this unit becomes active and it passes activity top-down. All top-down units which receive this activation and which receive activation from their bottom-up counterparts become active. In this way activity is passed down to the terminal nodes and the active top-down nodes of the network represent the parse tree(s). The parse in the connectionist network completes in $O(n)$ time.

Above our assumption was that grammars are in CNF. This is not a necessary condition, but it facilitates the present discussion. See Fanty[1985] or the full version of this paper for possible relaxations of this condition and the consequences for the time complexity.

6. Conclusions

A survey of some ideas in parallel parsing has been presented. In the field of natural language processing the Earley and CYK method are well known. Sometimes closely related methods such as (active) chart parsing are used. Because of this close relationship a parallel implementation along the lines sketched above is possible. Chart parsing (and Earley parsing) can be done with a more modest number of processors if an agenda approach is followed (see e.g. Grisham and Chitrao[1988]). Earley's algorithm can be modified to transition networks and extended to ATN's (see e.g. Chou and Fu[1975]). Therefore it is worthwhile to investigate a similar parallel approach to the parsing of ATN's. No attention has been paid to ideas aimed at improving upper bounds for the recognition and parsing of general context-free languages. An introduction to that area can be found in Chapter 4 of Gibbons and Rytter[1988]. Neither have we been looking here at the connectionist approaches in parsing and natural language processing as they are discussed in the papers of Cottrell and Small[1984], Waltz and Pollack[1985], McClelland and Kawamoto[1986] and Small[1987]. More references to papers on parallel parsing can be found in Nijholt et al[1989].

Acknowledgements.
I am grateful to P.R.J. Asveld for his comments on an earlier version of this paper and to Theo Vosse for drawing my attention to some papers on connectionist parsing. Some discussions with Bart Van Acker and Bart De Wolf have improved my understanding of parallel parsing methods.

7. References

Carlisle, W.H. and D.K. Friesen [1985]. Parallel parsing using Ada. Proceedings 3rd Annual National *Conference on Ada Technology*, March 1985, 103–106.

Chiang, Y.T. and K.S. Fu [1982]. A VLSI architecture for fast context-free language recognition (Earley's algorithm). Proceedings Third International *Conf. on Distributed Comp. Systems*, 1982, 864–869.

Chou, S.M. and K.S. Fu [1975]. Transition networks for pattern recognition. School for Electrical Engineering, Purdue University, West Lafayette, Indiana, TR-EE 75–39, 1975.

Chu, K.-H. and K.S. Fu [1982]. VLSI architectures for high-speed recognition of context-free languages and finite-state languages. Proceedings of the Ninth Annual *Symposium on Computer Architectures, SIGARCH Newsletter* 10 (1982), No.3, 43–49.

Cottrell, G.W. and S.L. Small [1984]. Viewing parsing as word sense discrimination: A connectionist approach. In: *Computational Models of Natural Language Processing*, B.G. Bara and G. Guida (eds.), Elsevier Science Publishers, North-Holland, 1984, 91–119.

Fischer, C.N. [1975]. Parsing context-free languages in parallel environments. Ph.D. Thesis, Tech. Report 75-237, Dept. of Computer Science, Cornell University, 1975.

Gibbons, A. and W. Rytter [1988]. Parallel recognition and parsing of context-free languages. Chapter 4 in *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, 1988.

Graham, S.L. and M.A. Harrison [1976]. Parsing of general context-free languages. *Advances in Computers*, Vol. 14, M. Yovits and M. Rubinoff (eds.), Academic Press, New York, 1976, 76–185.

Grisham, R. and M. Chitrao [1988]. Evaluation of a parallel chart parser. In: Proceedings of the *Second Conference on Applied Natural Language Processing*, Association for Computational Linguistics, 1988, 71–76.

Guibas, L.J., H.T. Kung and C.D. Thompson [1979]. Direct VLSI implementation of combinatorial algorithms. *Proc. Conf. on VLSI*, Caltech, January 1979, 509–526.

Lang, B. [1971]. Parallel non-deterministic bottom-up parsing. In: Proc. *Int. Symposium on Extensible Languages*. Grenoble, 1971, *SIGPLAN Notices* 6, Nr. 12, December 1971.

Lincoln, N. [1970]. Parallel programming techniques for compilers. *SIGPLAN Notices* 5 (1970), No.10, 18–31.

Loka, R.R. [1984]. A note on parallel parsing. *SIGPLAN Notices* 19 (1984), No.1, 57–59.

Lozinskii, E.L. and S. Nirenburg [1986]. Parsing in parallel. *Computer Languages* 11 (1986), 39–51.

McClelland J.L. and A.H. Kawamoto [1986]. Mechanism of sentence processing: assigning roles to constituents of sentences. Chapter 19 in *Parallel Distributed Processing.* Vol.2: *Psychological and Biological Models*, D.E. Rumelhart, J.L. McClelland and the PDP Research Group, The MIT Press, Cambridge, Mass., 1986, 272-325.

Mickunas, M.D. and R.M. Schell [1978]. Parallel compilation in a multiprocessor environment. Proceedings *ACM Annual Conf.*, 1978, 241–246.

Nijholt, A. [1988]. *Computers and Languages: Theory and Practice.* Studies in Computer Science and Artificial Intelligence. North-Holland, Elsevier Science Publishers, Amsterdam, 1988.

Nijholt, A. et al [1989]. An annotated bibliography on parallel parsing. Twente University, Internal Memorandum, in preparation, 1989.

Selman, B. and G. Hirst [1987]. Parsing as an energy minimization problem. Chapter 11 in *Genetic Algorithms and Simulated Annealing.* Research Notes in A.I., Morgan Kaufmann Publishers, Los Altos, California, 1987.

Small, S.L. [1987]. A distributed word-based approach to parsing. In: *Natural Language Parsing Systems.* L. Bolc (ed.), Springer-Verlag, Berlin, 1987, 161–201.

Srikant, Y.N. and P. Shankar [1987]. Parallel parsing of programming languages. *Information Sciences* 43 (1987), 55–83.

Sijstermans, F.W. [1986]. Parallel parsing of context-free languages. Doc. No. 202, Esprit Project 415, Subproject A: Object-oriented language approach, Philips Research Laboratories, Eindhoven, 1986.

Tan, H.D.A. [1983]. VLSI-algoritmen voor herkenning van context-vrije talen in lineaire tijd. Rapport IN 24/83, Stichting Mathematisch Centrum, Amsterdam, Juni 1983.

Tomita, M. [1985]. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, Dordrecht, 1985.

Tseytlin, G.E. and E.L. Yushchenko [1977]. Several aspects of theory of parametric models of languages and parallel syntactic analysis. In: *Methods of Algorithmic Language Implementation.* A. Ershov and C.H.A. Koster (eds.), Lect. Notes Comp. Sci. 47, Springer-Verlag, Berlin, 1977, 231–245.

Waltz, D.L. and J.B. Pollack [1985]. Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science* 9 (1985), 51-74.

Yonezawa, A. and I. Ohsawa [1988]. Object-oriented parallel parsing for context-free grammars. In: *Proceedings of the 12th International Conference on Computational Linguistics (COLING'88)*, Budapest, 1988, 773–778.