# Views on Parallel Parsing: A preliminary survey

*Anton Nijholt*†

Faculteit Informatica, Universiteit Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands

*ABSTRACT*

A preliminary survey is presented of approaches to the parsing problem in parallel environments. The discussion includes parsing schemes which use more than one traditional parser, schemes where 'non-deterministic' choices during parsing are assigned 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.

## 1. Introduction

In the early 1970's papers appeared in which ideas on parallel compiling and executing of programs were investigated. In these papers parallel lexical analysis, syntactic analysis (parsing) and code generation were discussed. At that time various multiprocessor computers were introduced (CDC6500, 7600, STAR, ILLIAC IV, etc.) and the first attempts were made to construct compilers which used more than one processor when compiling programs (Lincoln[1970], Ellis[1971], and Zosel[1973]). In 1975 this activity led to a conference on *Programming Languages and Compilers for Parallel and Vector Machines* (see Donegan and Katz[1975] for parsing techniques for vector machines). Slowly, with the coming of new parallel architectures and the advent of VLSI, interest increased and presently interest in 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 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

† With help from Bart Van Acker and Bart De Wolf, Faculteit Wetenschappen, Vrije Universiteit Brussel, Belgium.

cognitive scientists turn to parallel approaches for their problems. Hence, also in natural language processing environments there is presently a widespread interest in parallel processing techniques for natural language analysis.

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 were introduced were closely related to existing and well known serial parsing methods, such as LL-, LR-, and precedence parsing. Parallel parsing was often looked upon as deterministic parsing of sentences with more than one serial parser. However, with the massively parallel architectures that have been designed and constructed, and with the possibility to design special-purpose chips for parsing and compiling, also the well known methods for general context-free parsing have again been investigated in order to see whether they allow parallel implementations. Typical results in this area are $O(n)$-time parallel parsing methods 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 connections between them in the human brain (that is, an order of $10^{11}$ neurons and $10^3$–$10^4$ connections each). They model human language processing with connectionist models and therefore they are interested in massive parallelism and low degradation in the face of local errors. In connectionist 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 (see e.g., Cottrell and Small[1984], Waltz and Pollack[1985] and Small[1987]). The second group of language researchers, those interested in designing and building high performance language systems, are interested in parallelism as well. 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 an 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 can be assigned to these tasks.

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 are assigned 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.

## 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[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 → NP VP
2. S → S NP
3. NP → *det *n
4. PP → *prep NP
5. VP → *v NP

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 'r *n*' indicates the action 'reduce the stack using rule *n*'. The 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 go 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)

| 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.

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.

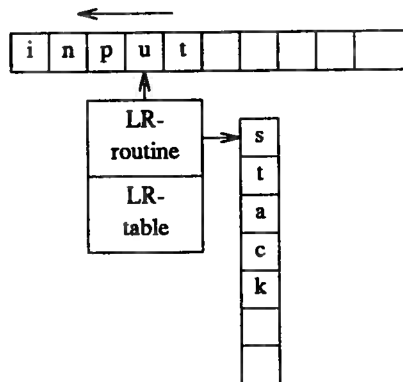


**Fig. 2** LR-parser.

## 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 $\alpha^R$ denote the reversal of $\alpha$. Let $G^R=(N,\Sigma,P^R,S)$ be the context-free grammar which is obtained from $G$ by defining $P^R=\{i.\,A\to\alpha^R \mid i.\,A\to\alpha\in P\}$. 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 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 that $G^R$ 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. In theory the starting point of each SPM may be any symbol in the input string. 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 is encountered 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 only one stack and it only contains $s_0$, the start 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 is started has 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 $M(s, a)=s'$ then push $s'$ on the stack;
(b) if $M(s, a)=\varnothing$ 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 is 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 can not, 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 $Q$, if $R(s)=\{0\}$, then add $s$ to the set of states of the new SPM and if $R(s)=\{i\}$ 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. With 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
2. S → S NP
3. NP → *n
4. NP → *det *n
5. NP → NP PP
6. PP → *prep NP
7. VP → *v NP

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. 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 implementation nothing is gained when we weave a list of stacks into a graph-structured stack. In fact, 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 high – to consider parallel versions of this algorithm since it is not in advance restricted 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 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 and stack and 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
2. S → S NP
3. NP → *n
4. NP → *det *n
5. NP → NP PP
6. PP → *prep NP
7. VP → *v NP

This set of rules will be viewed as a network of concurrently working computing agents. 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.

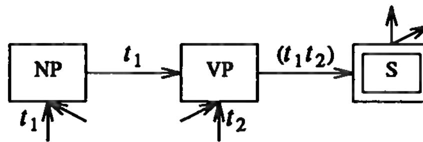


**Fig. 4** From rules to configuration.

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.

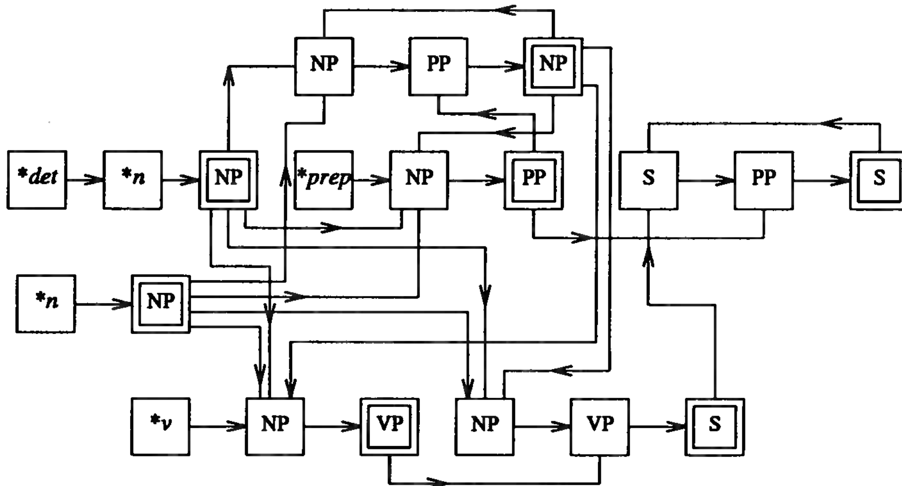


**Fig. 5** Computing agents for grammar *G*.

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. An other useful extension is the capability to unparse a sentence when the user of a system based on this method 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 10 and 20 were used for input. Parallelism was simulated by time-slicing. From this simulation it was concluded 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 are normally not seen in context-free descriptions of natural language. As an example, consider the method described above for a context-free grammar with rules $S \rightarrow aS$ and $S \rightarrow a$ only.

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.

Another configuration with a reduced number of computing agents is obtained if we have an agent for each nonterminal symbol of the grammar. For the example grammar we have four agents, the *S*-, the *NP*-, the *VP*-, and the *PP*-agent. If we want we can 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

Fig. 6 Agents for grammar rules.

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 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 a strictly upper-triangular $(n+1)\times(n+1)$ recognition table $T$ is constructed. Each table entry $t_{i,j}$ will contain a subset of $N$ (the set of nonterminal symbols) such that $A \in t_{i,j}$ if and only if $A \Rightarrow^* a_{i+1} \cdots a_j$. String $x \in L(G)$ if and only if $S \in t_{0,n}$ when 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 $i+1 \le k \le j-1$, there is a production $A \rightarrow BC \in P$ with $B \in t_{i,k}$ and $C \in t_{k,i+j}$.

In this 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 Strictly 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.

From the recognition table we can conclude a two-dimensional configuration of processes. For each entry $t_{i,j}$ of the strictly 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 process structure for $n=5$. As soon as a table element is computed or available 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 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]).

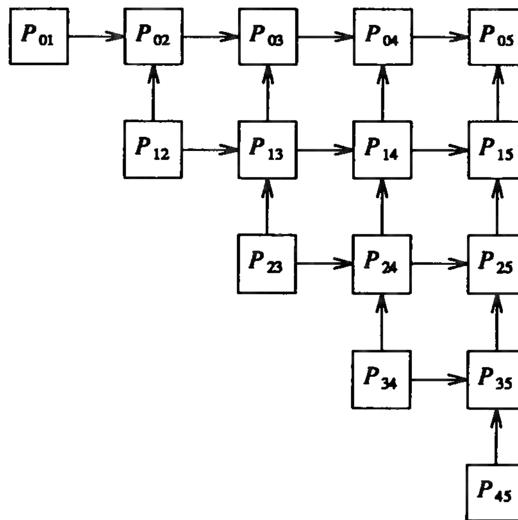


Fig. 9 Process configuration for CYK's algorithm.

## Earley's Algorithm

The second algorithm we discuss in this section is the well known Earley's method. 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 the general context-free 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 PRED:$N \rightarrow 2^D$, where $D = \{A \rightarrow \alpha \cdot \beta \mid A \rightarrow \alpha\beta \in P\}$, is defined as

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

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

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

Initially, $t_{0,0}$=PREDICT($\{S\}$) 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 $j-1 \leq k \leq i+1$ 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 $j-1 \leq k \leq i+1$, $B \rightarrow \alpha \cdot A\beta\gamma \in t_{i,k}$, $A \rightarrow \omega \cdot \in t_{k,j}$ 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 $t_{j,j}$=PREDICT($X_j$).

Fig. 10 illustrates the four steps by showing which elements provide the information for computing $t_{i,j}$ and $t_{j,j}$. 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 $t_{i,i}$, $0 \leq i \leq n$, 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

---

† 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]).

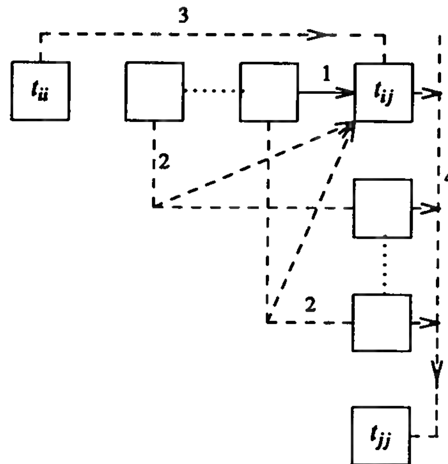


**Fig. 10** Computation of $t_{ij}$ and $t_{jj}$ in Earley's algorithm.

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($N$), $0 \le i \le n$.

(2) Set $d$=0. 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 $j=i+d$ according to:

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

(2.2) Add $B \to \alpha a \beta \cdot \gamma$ to $t_{i,j}$, if, for any $k$ such that $j-1 \le k \le i+1$, $B \to \alpha \cdot A \beta \gamma \in t_{i,k}$, $A \to \omega \cdot \in t_{k,j}$ and $\beta \Rightarrow^* \varepsilon$.

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

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).

## Natural Language Applications

In the field of natural language processing the Earley method is well known. Sometimes closely related methods such as (active) chart parsing (see Winograd[1983]) are used. Because of this close relationship a parallel implementation along the lines sketched above is possible. A similar remark holds for parsing augmented transition networks (ATN's). Earley's algorithm can be modified to transition networks and extended to ATN's (see e.g. Chou and Fu[1975]). It seems worthwhile to explore the possibilities of parallel Earley parsing for natural language applications.

### Transformational Development of Parallel Parsing Algorithms

An interesting view on constructing correct parsing algorithms that are executable on parallel architectures is given by Partsch[1984b]. This view is part of a more general methodology of program development by transformations: develop a program from a formal specification by stepwisely applying correctness-preserving transformation rules (see e.g. Bauer and Wössner[1982]). In Partsch[1984a] this approach is applied to the development of the serial Earley algorithm (see also Jones[1971]) and in Partsch[1984b] the same approach is presented for the development of a parallel version of the Cocke-Younger-Kasami algorithm, and it is remarked that the same development strategy can be used to obtain a parallel version of Earley's algorithm. Without going into details we mention that with the help of some general transformation principles a tabular version of Cocke-Younger-Kasami's algorithm is obtained which is well suited for execution on a vector machine. With $n$ processors the algorithm takes $O(n^2)$ time for context-free grammars in Chomsky Normal Form. If there are less than $n$ processors a combination of parallel and sequential computation of the table elements can be done.

## 5. Conclusions

A survey of some ideas in parallel parsing has been presented. 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. In an extended version of this paper context-free parsing in connectionist networks will be discussed. More references to papers on parallel parsing can be found in Nijholt et al[1989].

## 6. References

Bauer, F.L. and H. Wössner [1982]. *Algorithmic Language and Program Development*. Springer-Verlag, Berlin, 1982.

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.

Donegan, M.K. and S.W. Katzke [1975]. Lexical analysis and parsing techniques for vector machines. In: Proceedings Conference on *Programming Languages and Compilers for Parallel and Vector Machines. SIGPLAN Notices* 10 (1975), No.3, 138-145.

Ellis, C.A. [1971]. Parallel compiling techniques. Proc. *Annual ACM Conf.* 1971, New York, 508-519.

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.

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.

Jones, C.B. [1971]. Formal development of correct algorithms. An example based on Earley's recognizer. Proc. of ACM SIGPLAN Conf. on *Proving Assertions about Programs*, 1971, 150-169.

Kindervater, G.A.P. and J.K. Lenstra [1985]. An introduction to parallelism in combinatorial optimization. In: *Parallel Computers and Computations*, J.K. van Leeuwen and J.K. Lenstra (eds.), CWI-Syllabus 9, Centre for Mathematics and Computer Science, Amsterdam, 1985, 163-184.

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 (January 1984), 57-59.

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

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.

Partsch, H. [1984a]. Structuring transformational developments: a case study based on Earley's recognizer. *Sci. Comput. Program.* 4 (1984), 17-44.

Partsch, H. [1984]. Transformational derivation of parsing algorithms executable on parallel architectures. In: *Programmiersprachen und Programmentwicklung*, 8. Fachtagung, Zürich, März 1984, Informatik Fachberichte 77, U. Ammann (ed.), Springer-Verlag, Berlin, 1984.

Rytter, W. [1987]. On the complexity of parallel parsing of general context-free languages. *Theoretical Computer Science* 47 (1987), 315-322.

Schell Jr., R.M. [1979]. Methods for constructing parallel compilers for use in a multi-processor environment. Ph.D. Thesis, Dept. of Computer Science, Report No. 958, University of Illinois at Urbana-Champaign, February 1979.

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, Juni 1983.

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

Winograd, T. [1983]. *Language as a Cognitive Process*. Vol. 1: Syntax. Addison-Wesley Publishing Company, Reading, Mass., 1983.

Yonezawa, A. and I. Ohsawa [1988]. Object-oriented parallel parsing for context-free grammars. 1988, 773-778.

Zozel, M. [1973]. A parallel approach to compilation. In: *Principles of Programming Languages*, 1973, 59-70.