Object-Oriented Parallel Parsing for Context-Free Grammars

Akinori Yonezawa
Ichiro Ohsawa

Department of Information Science
Tokyo Institute of Technology
Ookayama, Meguro-ku
Tokyo 152, Japan

yonezawa%is.titech.junet%utokyo-relay.csnet@relay.cs.net
ohsawa%is.titech.junet%utokyo-relay.csnet@relay.cs.net

Abstract

This paper describes a new parallel parsing scheme for context-free grammars and our experience of implementing this scheme, and it also reports the result of our simulation for running the parsing program on a massive parallel processor.

In our basic parsing scheme, a set of context free-grammar rules is represented by a network of processor-like computing agents each having its local memory. Each computing agent in the network corresponds to an occurrence of a non-terminal or terminal symbol appearing in the grammar rules. Computing agents in the network work concurrently and communicate with one another by passing messages which are partial parse trees.

This scheme is shown to be fast (`O(n*h)` time for the first complete parse tree, where n is the length of an input sentence and h is the height of the parse tree) and useful in various modes of parsing such as on-line parsing, overlap parsing, on-line unparsing, pipe-lining to semantics processing, etc. Performance evaluation for implementing this scheme on a massive parallel machine is conducted by distributed event simulation using the Time Warp mechanism /Jefferson85/.

Our parsing scheme is implemented in a programming language called ABCL/1 which is designed for object-oriented concurrent programming and used for various concurrent programming /Yonezawa86/. The program is currently runing on standard single-cpu machines such as SUN3s and Symbolics Lisp machines (by simulated parallelism).

In our experiment and simulation, a set of about 250 context-free grammar rules specifying a subset of English is represented by the corresponding network of objects (i.e., computing agents) and about 1100 concurrently executable objects are involved.

1 Introduction

This paper presents a new approach to parsing for context-free grammars, which is conceptually very simple. The significance of our approach is supported by recent trends in computer-related fields. In computational linguistics, much attention has been drawn to parsing of context-free grammars owing to the progress of context-free based grammatical frameworks for natural languages such as LFG /Kaplan82/, GPSG /Gazdar85/. Furthermore, many practical natural language interface systems are based on context-free (phrase structure) grammars. In computer architecture and programming, exploitation of parallelism has be actively pursued; innovative computer architectures utilizing a large number of processors /Gottlieb83/ /Seitz85/ have been developed and accordingly new methodologies for concurrent programming /Agha86/ /Gelernter86/ /Yonezawa87/ have been actively studied.

In our basic parsing scheme, a given set of context-free grammar rules is viewed as a network of terminal and non-terminal symbols, and a corresponding network of processor-like computing agents with internal memory (or simple processors) is constructed. The node set of the network has a direct one-to-one correspondence to the set of occurrences of symbols appearing in the grammar rules and the link topology of the network is directly derived from the structure of the set of grammar rules. Our parsing scheme produces all the possible parse trees for a given input string without duplication.

Since the notion of *objects* in object-oriented concurrent programming /Yonezawa87/ naturally fits the computing agents composing the network, this parsing scheme has been implemented in an object-oriented language for concurrent programming ABCL/1/Yonezawa86/ by representing each computing agent in the network as an *object* of ABCL/1.

2 The Basic Scheme

2.1 A Symbol as a Computing Agent

Our approach is basically bottom-up. Suppose we have a context free grammar rule such as:

```
VP --> V NP                    (1)
```

In bottom-up parsing, a usual interpretation of this kind of rule is:

> In a substring of an input string, if its first half portion can be reduced to a category (terminal/non-terminal symbol) V and subsequently, if its second half portion can be reduced to a category VP, then the whole substring can be reduced to a category VP.

This interpretation is implicitly based upon the following two assumptions about parsing process:

- a single computing agent (processor or process) is working on the input string, and
- non-terminal or terminal symbols such as VP, V, and NP are viewed as passive tokens or data.

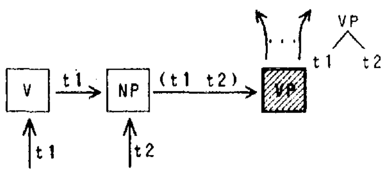

Figure 1:

Instead, we will take a radically different approach, in which

- more than one, actually, a number of computing agents are allowed to work concurrently, each performing a rather simple task,
- for each occurrence of a non-terminal or terminal symbol in grammar rules, a computing agent is assumed,
- such a computing agent receives data (messages), manipulates and stores data in its local memory, and also can send data (messages) asynchronously to other computing agents that correspond to non-terminal or terminal symbols, and
- data to be passed around among such computing agents are partial parse trees.

Suppose that the computing agent which acts for the V symbol in Rule (1) has received a (token that represents a) partial parse tree t1. Also suppose that the computing agent which acts for the NP symbol in Rule (1) has received a partial parse tree t2. If the terminal symbol which is the right boundary of t1 is, in the original input string, *adjacent* to the terminal symbol which is the left boundary of t2, then t1 and t2 can be put together and they can form a larger partial parse tree which corresponds to the VP symbol in Rule (1). See Figure 1.

For example, let us consider an input string:

I saw a girl with a telescope.

If t1 is a parse tree constructed from 'saw' and t2 is a parse tree constructed from 'a girl', then the right boundary of t1 is *adjacent* to the left boundary of t2. But if t2 is a parse tree constructed from 'a telescope', then t1 and t2 are not adjacent and a larger parse tree cannot be constructed from them.

Now, which computing agent should check the boundary adjacency, and which one should perform the tree-constructing task? In our scheme, it is natural that the computing agent acting for the NP symbol does the boundary checking because, in many simple cases, the NP agent often receives t2 after the V agent receives t1 (due to the left-to-right nature of on-line processing). In order for the NP agent to be able to perform this task, the V agent must send t1 to the NP agent. Upon receiving t1 from the V agent, the NP agent checks the boundary adjacency between t1 and t2 if it has already received t2. If t2 has not arrived yet, the NP agent has to postpone the boundary checking until t2 arrives and t1 will be stored in the NP agent's local memory. If the two boundaries are not adjacent, the NP agent stores t1 in its local memory for future references. Later on when the NP agent receives subsequently arriving partial parse trees, their left boundary will be checked againt the right boundary of t1.

When the adjacency test succeeds, the NP agent concatenates t1 and t2 and sends them to the computing agent acting for the non-terminal symbol VP in Rule (1). The VP agent constructs, out of t1 and t2, a partial parse tree with the root-node tag being the non-terminal symbol 'VP.' This newly constructed partial parse tree is then *distributed* by the VP agent to *all* the computing agents each of which acts for an occurrence of symbol VP in the right-hand side of a rule. This distributed tree in turn plays a role of data (messages) to the computing agents in exactly the same way as t1 and t2 play roles of data to the V and NP agents above.

This is the basic idea of our parsing scheme. It is very simple. It is the matter of course that every single computing agent acting for a non-terminal or terminal symbol can work independently, *in parallel* and *asynchronously*. Rule (1) is represented as the computing agent network illustrated in Figure 1. (This is part of a larger network.) Boxes and arrows denote computing agents and flows of trees, respectively.

2.2 A Set of Rules as a Network of Computing Agents

It should be clear from the previous subsection that a set of context-free grammar rules (even a singleton grammar) is represented as a network of computing agents each of which acts for an occurrence of a non-terminal or terminal symbol in a grammar rule. More precisely, the correspondence between the set of computing agents and the set of occurrences of symbols in the set of grammar rules is one-to-one; for each occurrence of a symbol in a rule, there is one distinct computing agent. For example, the following set of rules (including Rule (1)) is represented as the network depicted in Figure 2.

```
S  --> NP VP          (2)
S  --> S PP           (3)
NP --> DET N          (4)
PP --> PREP NP        (5)
```

A white box corresponds to the computing agent acting for a symbol in the right-hand side of a grammar rule and a dark box corresponds to the computing agent acting for the non-terminal symbol in the left-hand side of a grammar rule. Note that the dark box labeled with 'NP' (at the bottom of the figure) is linked to three boxes labeled with 'NP.' This means that a partial parse tree constructed by the computing agent acting for the left symbol NP in Rule (4) is distributed to the three computing agents acting for the three occurrences of symbol NP in Rules (1), (2), and (5). Note that Rule (3) is left-recursive, which is represented as the feed-back link in Figure 2.

2.3 Three Types of Computing Agents

[1]As the reader might have already noticed, there are three types of computing agents: Type-1 corresponds to the left symbol in a grammar rule, Type-2 corresponds to the left-corner (i.e. left-most) right symbol, and Type-3 corresponds to other right symbols. (If a grammar rule has more than two right symbols, each of the right symbols except the left-corner symbol is represented as a Type-3 agent.) For example, in Rule (1), VP is Type-1, V is Type-2, and NP is Type-3.

[1]This subsection may be skipped if the idea of the scheme is already clear.

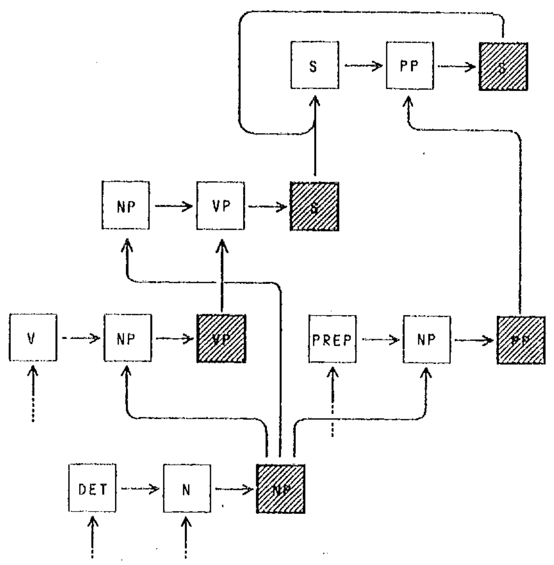

Figure 2:

A Type-1 computing agent A1 receives a concatenation of parse trees from the Type-3 agent acting for the right-most right symbol (e.g., NP for the case of Rule (1)) and constructs a new parse tree with its root node being the non-terminal symbol that A1 acts for and distributes it to all the Type-2 or Type-3 agents acting for the occurrences of the same non-terminal symbol (e.g., 'NP' in the above case).

A Type-2 computing agent A2 receives a partial parse tree from some computing agent that is acting for the occurrence of the same symbol as A2 acts for, and simply passes it to the computing agent acting for the symbol occurrence which is right-adjacent to the symbol occurrence that A2 is acting for. In the case of Rule (1), a Type-2 agent acting for V simply passes the received partial parse tree to the computing agent acting for NP. In the case where a grammar rule has just one right symbol as in

```
NP --> N,                (6)
```

a Type-2 agent acting for N sends a partial parse tree to the Type-1 agent acting for NP.

A Type-3 computing agent has two kinds of sources of parse trees to receive: one from Type-1 agents and the other from the Type-2 or Type-3 agent acting for its left-adjacent symbol occurrence. In the case of Rule (1), the Type-3 agent acting for NP receives partial parse trees from Type-1 agents acting for occurrences of symbol NP in other rules and also from the Type-2 agent acting for V in Rule (1). Upon receiving a partial parse tree t1 from one of the sources, a Type-3 agent A3 checks to see if it has already received, from the other kind of source, a partial parse tree which clears the boundary adjacency test against t1. If such a parse tree t2 has already arrived at A3, then A3 concatenates t1 and t2 and passes them to the computing agent acting for the symbol occurrence which is right-adjacent to the symbol occurrence A3 acting for. If no such parse tree has arrived yet, A3 stores t1 in its local memory for the future use. In the case where no right-adjacent symbol exits in the grammar rule, (which means that the symbol occurrence A3 is acting for is the right-most right symbol in the grammar rule), A3 sends the concatenated trees to the Type-1 computing agent acting for the left symbol of the grammar rule.

2.4 Terminal Symbols as Computing Agents

It should be noted that in our basic scheme we do not make any distinction between non-terminal symbols and terminal symbols. In fact, this uniform treatment contributes to the conceptual simplicity of our parsing scheme. We do not have to make a special treatment for grammar rules such as:

```
NP --> NP and NP          (7)
```

where a lower case symbol 'and' is a terminal symbol. The uniformity implies that a word of a natural language, say 'fly' in English, which has more than one grammatical category should be described as follow:

```
V --> fly                 (8)
N --> fly                 (9)
```

where Rules (8) and (9) indicate that a word 'fly' can be a *verb* or *noun*. The two rules are represented by two Type-1 agents acting for V and N, and two Type-2 agents acting for the two occurrences of 'fly' in Rules (8) and (9). Thus, in our parsing scheme, the grammatical categories of each word in the whole vocabulary in use are described by grammar rules with a single right symbol. This means that conceptually, one or more computing agents exist for each word. (Those who might worry about the number of computing agents acting for words should read Subsection 4.2.)

2.5 Input to the Network

In our parsing scheme, a given set of grammar rules is compiled as a network of computing agents in the manner described above. Then, how is an input string fed into the network of computing agents? We assume that an input string is a sequence of words (namely, terminal symbols).

In feeding an input string into the network, two things has to be taken into account. One is: for each word in an input string, appropriate computing agents, to which the word should be sent, must be found. Of course, such computing agents are ones that act for the occurrences of the word in the grammar rules. Notice that there can be more than one such computing agent for each word, due to multiplicity of grammatical category and the multiple occurrences of the same symbol in grammar rules. Since the set of appropriate computing agents can be known in compiling a given set of grammar rules, such information should be kept in a special computing agent which does the managerial work for the network. Let us call it the *manager agent*. The manager agent receives an input string and sends (or distributes) each word in the input string to the corresponding agents in the network in the on-line manner.

The other thing needed to be considered in feeding the input is: the information about the order of words appearing in an input string must be provided to computing agents in the network in an appropriate manner. This is because Type-3 computing agents need such information to perform the boundary adjacency test. For this, each word to be sent (or distributed) to computing agents in the network should be accompanied with its positional information in the input string. Suppose an input string is `I saw a girl with a telescope`. Then a word `girl` should be sent with the pair of its starting position and its ending position. The

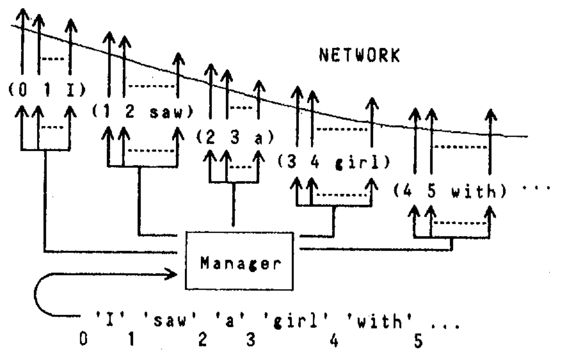

Figure 3:

actual form of data (message) for the word `girl` may look like `(3 4 girl)`. See Figure 3. This data form convention is adopted in dealing with more general parse trees. (In fact, a single word (terminal symbol) is also the simplest case of parse tree.)

2.6 How Partial Parse Trees Flow

To get a more concrete feeling of how symbols are processed in the network, let us look at the flows of words `a` and `girl` in the initial phase. (See Figure 4) Assuming that the following rules are compiled in addition to Rules (1) through (5),

```
DET --> a       (10)
N   --> girl    (11)
```

the manager agent sends `(2 3 a)` and `(3 4 girl)` to the Type-2 computing agent acting for `a` in Rule (10) and the Type-2 computing agent acting for `girl` in Rule (11), respectively. They are in turn sent to a Type-1 agent Det1 acting for `DET` in Rule (10) and a Type-1 agent N1 acting for N in Rule (11), respectively. These Type-1 agents construct a parse tree with its root node label being `DET` or `N`. Then the parse tree constructed by Det1 is sent to a Type-2 agent Det2 acting for `DET` in Rule (4). Similarly, the parse tree constructed by N1 is sent to a Type-*3* agent N2 acting for `N` in Rule (4). In both cases, the positional information is accompanied. That is, the actual data forms to be sent are `(2 3 (DET a))` and `(3 4 (N girl))`.

Agent Det2 simply passes the parse tree to agent N2. N2 performs the boundary adjacency test between `(2 3 (DET a))` and `(3 4 (N girl))` and finds the test to be ok. Since the test is ok, N2 concatenates the two data forms, constructing a new single data form:

`(2 4 (DET a) (N girl))`

This new data form is then sent to the Type-1 agent acting for `NP` in Rule (4). This agent constructs a data form of the parse tree for NP, which looks like:

`(2 4 (NP (DET a) (N girl)))`

This data form will be distributed among the Type-2 and Type-3 computing agents acting for symbol `NP` in the network. (See Figure 4.) Finally, when a computing agent acting for `S` receives a message `(0 7 (S ...))`, we can say that a complete parse tree for the input string has been constructed as part of the message.

It should be reminded that actions taken by computing agents such as Det1, Det2, N1, and N2 are performed all

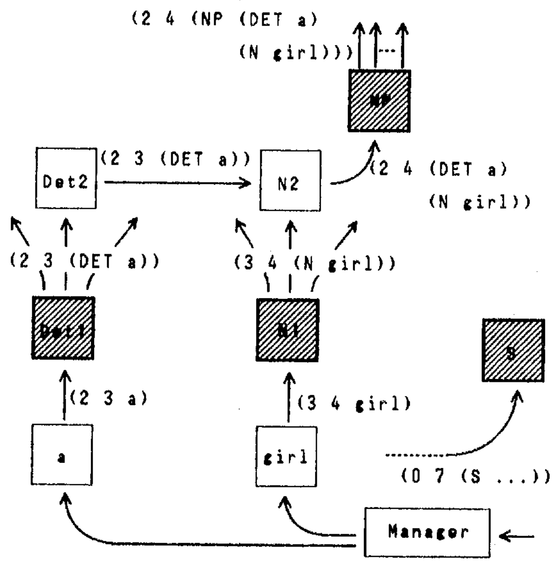

Figure 4:

in parallel. Also note that such computing agents keep being activated as long as data forms continue to arrive, and computing agents acting for `S` receive messages containing (partial) parse trees with the root node label being `S`.

3 Applications

3.1 On-Line Parsing and Overlap Parsing

In starting the parsing process, our scheme does not require the network of computing agents to be fed any token that indicates the end of an input string. That is, an input string can be processed one by one from the beginning in an on-line fashion. Even if feeding an input string to the network is suspended in the middle of the string, partial parse trees can be constructed based on the part of the input string that has been fed so far, and the feeding of the rest of the input string can be resumed at any moment. Thus, our parsing scheme is quite useful in real-time applications such as interpreting telephony (simultaneous interpretation). Notice that our scheme does not require that an input string is fed in the left-to-right manner; words in the input string can be fed in any order as long as the positional information of each word in the input string is accompanied. (cf. Subsection 2.5)

Our parsing scheme has no difficulty even when more than one input string is fed to the network simultaneously as long as different input strings are fed separately. The separation can be easily made by attaching the same tag (or token) to each word in the same input string. Such a tag is copied and inherited to partial parse trees which are constructed from the same input string. When a Type-3 computing agent tests the boundary adjacency between two partial parse trees, the sameness of the tags of the two partial parse trees are checked additionally. This capability of handling the multiple input strings is useful in processing the overlapping utterances by more than two persons engaged in conversation.

This way of handling the multiplicity of input strings is similar to the idea of *color tokens* used in data-flow computer architectures.

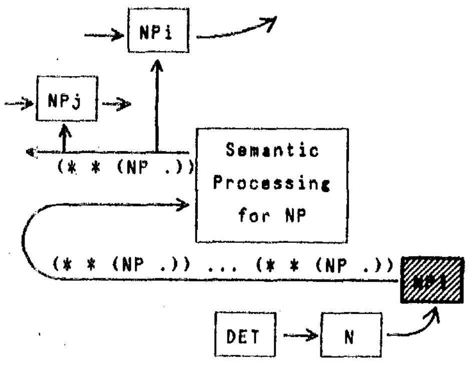

Figure 5:

3.2 Unparsing

Suppose the user is typing an input string on a keyboard and s/he hits the 'backspace' key to correct previously typed words. In the case where these incorrect words have already been fed to the network, our parsing scheme is able to unparse the incorrect portion of the input string and allows the user to retype it. Furthermore, the user can continue to type the rest of the originally intended input string.

This unparsing capability is realized by the use of *anti-messages*. The anti-message /Jefferson85/ of a message M sent to a computing agent A is a message that will be sent to A in order to cancel the effects caused by M. The actual task of cancelling the effects is carried out by A. (Thus A has been programmed beforehand so that it can accept cancelling messages and perform the cancelling task.) If necessary, A must in turn send anti-messages to cancel the effects caused by the messages A itself has already sent. In implementing the unparsing capability, the express-mode message passing in ABCL/1 /Yonezawa86/ is useful, which is a kind of *interrupt*-like high priority message passing.

3.3 Pipe-Lining to Semantic Processing Agents

Our parsing scheme produces all the possible (partial) parse trees for a given input string. In fact, if each Type-1 computing agent in the network stores in its local memory all the parse trees it constructs, all the components of the triangle matrix used in CKY parsing method (i.e., all the possible parse trees) are in fact stored among the Type-1 agents in the network in a distributed manner. If the semantic processing is required, these partial or complete parse trees can be sent to some computing agents which do semantics processing.

Actually, parse trees can be sent to semantic processing agents in a *pipe-lining* manner. Suppose a Type-1 computing agent Np1 is acting for an occurrence of a non-terminal symbol NP. Instead of letting Np1 distribute the parse trees it constructs to Type-2 or Type-3 agents acting for occurrences of the symbol NP, we can let Np1 send the parse trees to the semantics processing agent which checks the semantic validity of the parse trees in the pipe-lining manner. After filtered by the semantic processing agent, only semantically valid parse trees (possibly with semantics information being attached) are distributed to Type-2 or Type-3 computing agents acting for NP. See Figure 5.

These semantic filtering agents can be inserted at any links between Type-1 agents and Type-2 or Type-3 agents. The complete separation of the semantic processing phase from the syntactic processing phase in usual natural language processing systems corresponds to the placing semantic processing agents only after the Type-1 computing agents that act for the non-terminal symbol S that stands for *correct* sentences.

4 Analysis and Discussion

4.1 Implementation and Experiment

Our parsing scheme has been implemented using an object-oriented concurrent language ABCL/1. In this implementation, each computing agent in the network is represented as an ABCL/1 object which becomes active when it receive a message, and data forms containing partial parse trees are represented as messages that are passed around by objects.

The parsing program written in ABCL/1 runs on a standard single-cpu machine (e.g., Symbolics Lisp machines and Sun3s) in which parallelism is simulated by time-slicing. (The code for a simplified version of this program and sample session are given in /Yonezawa87a/.) Using this program, we have been conducting an experiment of our proposed parsing scheme for a context-free English grammar /Tomita86/ with the following characteristics:

- 224 context-free rules for non-terminal symbols (e.g., `NP -> DET N`),
- 445 context-free rules for terminal symbols (e.g., `N -> fly`),
- 94 distinct nonterminal symbols and 679 occurrences, and
- 295 distinct terminal symbols and 445 occurrences.

About 40 input sentences are used for the experiment and they are typically: 10 - 30 in length, and 10 - 20 in height (the height of a correct parse tree).

4.2 The Number of Computing Agents (Objects)

As is obvious from the construction of the network, the number of computing agents is exactly the same as that of the nodes of the network. Since the node set of the network has one-to-one correspondence to the set of symbol occurrences in a given set of grammar rules, the number of computing agents can be very large if the grammar is complex. Thus the number of computing agents (i.e., objects) of the network representing the above English grammar amounts to more than 1100 (more exactly 1124 = 445+679).

Of course, not all these agents can be active simultaneously. The number of all the agents that become active in processing an input string is small compared to that of the computing agents consisting of the network. Since the main task of a Type-1 agent (acting for the left symbol of a grammar rule) is just to distribute a constructed parse tree, this task can be performed by the Type-3 agent which acts for the rightmost right symbol of the grammar rule. Thus all the Type-1 agents can be eliminated. This reduces the number of computing agents considerably. Furthermore, there are number of other ways to reduce the number of computing agents at the sacrifice of both processing speed and the conceptual clarity of the parsing scheme. (We, however, believe that maturity of the technology for exploitation of parallelism will dispel the apprehensions regarding the number of computing agents.)

4.3 Performance Analysis by Distributed Event Simulation

We are interested in the performance of our parsing scheme in the case where the scheme is implemented on a parallal architecture which allocates a single processor for each computing agent (i.e., object) in the network. Since it is not much interesting to theoretically analyze the complexity of our parsing scheme, we have conducted simulation.

The simulation has been done by using a distributed event simulation technique. The very parsing program written in ABCL/1 was reused and slightly modified to form our distributed simulation program. As we mentioned above, the original parsing program is written in such a way that each computing agent in the network is represetnted by a concurrently executable *object* which becomes active when it receives a message. The simulation program preserves the original network structure of objects (i.e., computing agents in the scheme) of the parsing program. The only modifications made to the original parsing program are:

- each object keeps its local time,
- each message passed around by objects additionally contains a *time stamp* indicating the time of the message transmission measured at the *local* time of the object which sent the message,
- each object sends *anti-messages*/Jefferson85/ when it receives a message containing a time stamp indicating an earlier time than the current local time of the object, and
- accordingly, each object can handle an anti-message which requests to cancel the effects made by the original message.

The initial result of our simulation is that the first complete parse tree is produced from the network in `O(n*h)` time, measuring from the beginning of feeding an input string to the network, where `n` is the length of the input string and `h` is the height of the parse tree (*not the height of the network*). This result was obtained for the context-free English grammar mentioned in Section 4.1. In this simulation we assumed that both processing of a partial parse tree by a single object (i.e., a single computing agent) and a message transmission between two objects (i.e., two computing agents) take a single unit time.

Since all the possible complete parse trees for a given input string are produced from the network in the pipelining manner, the second and subsequent complete trees are expected to be produced in a short interval one by one. We have not yet analyzed the simulation results for these parse trees.

4.4 Generality of the Parsing Scheme

Our parsing scheme can handle the most general class of context free grammars except cyclic grammars. If a set of grammar rules has circularity[2], infinite message passing may take place in the network. To detect or avoid such infinite message passing, a special provision must be made. But fortunately such a provision can be done at the time of compiling the set of grammar rules into the corresponding network of computing agents. As suggested in Subsection 2.2 and Figure 2, left-recursive grammar rules can be handled without any modification to the grammar rules. However, from the nature of bottom-up parsing, our parsing scheme cannot handle an ϵ-rule (a rule that produces a null string). But as is well known/Hopcroft79/, all the ϵ-rules can be eliminated from a given set of grammar rules by transforming the set of rules without changing the generative power of the original set of rules.[3] It should be noted that our scheme can be extended to cope with context-sensitive grammars (or more expressive ones).

[2] A simple example of circular rules is: `A --> B, B --> A, B --> C`.

[3] The original language is assumed to contain no null symbol.

4.5 Previous Work

R.M. Kaplan advocated in /Kaplan73/ that natural language parsing should be conceptualized and implemented as a collection of asynchronous communicating parallel processes. Our work is basically along his line, but our algorithm is completely different from his and is based on finer grain and more massive parallelism than his idea illustrated in /Kaplan73/.

References

[Agha86] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, 1986.

[Gazdar85] G. Gazdar, E. Klein, G. K. Pullum and I. A. Sag, *Generalized Phrase Structure Grammar*, Basic Blackwell Publisher, 1985.

[Gelernter86] D. Gelernter, Domesticating Parallelism, *IEEE Computers*, No. 8, 1986.

[Gottlieb83] A. Gottlieb et al.: The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. Computers*, C-32, No.2, 1983.

[Hopcroft79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[Jefferson85] D. R. Jefferson: Virtual Time, *ACM Trans. Prog. Lang. Syst.*, Vol.7, No.3, 1985.

[Kaplan73] R. M. Kaplan: A Multi-Processing Approach to Natural Language, *Proc. NCC*, 1973.

[Kaplan82] R. M. Kaplan and J. Bresnan: Lexical-Functional Grammar: A Formal System for Grammar Representation, in *The Mental Representation of Grammatical Relations* J. Bresnan (ed.), The MIT Press, 1982.

[Kay67] M. Kay: Experiments with a Powerful Parser, *RM-5452-PR*, The Rand Corporation, 1967.

[Seitz85] C. L. Seitz: The Cosmic Cube, *CACM*, Vol.28, No. 1, 1985.

[Tomita86] M. Tomita, *Efficient Parsing for Natural Language*, Kluwer Academic Publisher, 1986.

[Yonezawa86] A. Yonezawa, J.-P. Briot and E. Shibayama: Object-Oriented Concurrent Programming in ABCL/1, *Proc. 1st ACM Symposium on Object-Oriented Programming, Systems, Languages, and Applications*, 1983.

[Yonezawa87] A. Yonezawa and M. Tokoro (Eds), *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[Yonezawa87a] A. Yonezawa and I. Ohsawa: A New Approach to Parallel Parsing for Context-Free Grammars, *Res. Report*, C-78, Dept. of Info. Sci., Tokyo Inst. of Tech., September 1978.