An Agent-based Parallel HPSG Parser for Shared-memory Parallel Machines

Takashi Ninomiya,[†] Kentaro Torisawa[†,††] and Jun'ichi Tsujii[†,†††]

We describe an agent-based parallel HPSG parser that operates on shared-memory parallel machines. It efficiently parses real-world corpora by using a wide-coverage HPSG grammar. The efficiency is due to the use of a parallel parsing algorithm and the efficient treatment of feature structures. The parsing algorithm is based on the CKY algorithm, in which resolving constraints between a mother and her daughters is regarded as an atomic operation. The CKY algorithm features data distribution and granularity of parallelism. The keys to the efficient treatment of feature structures are i) transferring them through shared-memory, ii) copying them on demand, and iii) writing/reading them simultaneously onto/from memory. Being parallel, our parser is more efficient than sequential parsers. The average parsing time per sentence for the EDR Japanese corpus was 78 msec and its speed-up reaches 13.2 when 50 processors were used.

KeyWords: *parsing, parallel parsing, concurrent object, agent, HPSG*

1 Introduction

This paper describes an agent-based parallel HPSG parser. Its efficiency is sufficient for practical use in that it can analyze real-world corpora by using a wide-coverage grammar and it works efficiently in terms of both analysis time and speed-up.

The HPSG formalism (Pollard and Sag. 1994) is the most widely accepted unification-based grammar theory in the area of computational linguistics. It is gaining ground as a non-transformational alternative to the Chomskyan grammar theory, a formal and theoretically proper linguistic theory. It has attracted much interest among NLP researchers, mainly because it is mathematically well-defined (Carpenter 1992) and is justified by detailed linguistic explanations, i.e., on both mathematic and linguistic grounds. Due to its being mathematically well defined, the HPSG formalism is suitable for computer-based, algorithmic processing used to develop efficient systems. Due to its being linguistically justified, systems based on it produce in-depth syntactic and semantic analyses, making them suitable for use over a wide range of NLP domains, particularly where precise and accurate interpretation is important.

† Department of Information Science, Graduate School of Science, University of Tokyo
†† Information and Human Behavior, PRESTO, Japan Science and Technology Corporation
† † † CCL, UMIST

Studies over the past decade of HPSG-based processing technology by various NLP researchers (Torisawa, Nishida, Miyao, and Tsujii 2000; Kiefer, Krieger, Carroll, and Malouf 1999; Copestake, Flickinger, Malouf, Riehemann, and Sag 1995) have led to drastic improvements in HPSG-based systems in terms of efficiency, accuracy, coverage, and depth of syntactic and semantic analyses. For example, HPSG-based parsing systems developed by our group (Tateisi, Torisawa, Miyao, and Tsujii 1998; Mitsuishi, Torisawa, and Tsujii 1998; Makino, Yoshida, Torisawa, and Tsujii 1998; Torisawa et al. 2000; Miyao, Makino, Torisawa, and T-sujii 2000) have improved the efficiency, coverage, and accuracy of HPSG-based systems to the point where they can parse the EDR Japanese corpus in less than 500 msec per sentence with 98.7% coverage and 88.6% bunsetsu-dependency accuracy. Other groups have shown that HPSG-based dialog-translation systems can precisely interpret a sentence by using semantic representations (Uszkoreit, Backofen, Busemann, Diagne, Hinkelman, Kasper, Kiefer, Krieger, Netter, Neumann, Oepen, and Spackman 1994; Krieger and Schaefer 1994; Kasper, Kiefer, Krieger, Rupp, and Worm 1999; van Noord, Bouma, Koeling, and Nederhof 1999). The system developed by the DFKI group (Uszkoreit et al. 1994; Flickinger 2000) provided correct syntactic and semantic analyses for 83% of 8,520 well-formed English utterances found in the transcriptions of 175 person-to-person dialogs.

Further advances in HPSG-based systems require advances in the efficiency of parsing techniques. If we could develop a more efficient parser, we would be able to apply more sophisticated techniques to the HPSG framework and achieve higher accuracy, wider coverage, and deeper syntactic and semantic analyses, thereby acquiring more precise interpretations. Though several efficient HPSG-based systems have already been developed, much more efficient parsers are needed.

The goal of this article is to achieve an efficient parallel HPSG parser. Recent improvements in shared-memory parallel machines have been drastic, and such machines will become standard in computing environments. By exploiting this parallelism for parsing systems, we will be able to apply more sophisticated techniques to the HPSG framework.

We have taken two steps in this direction.

(1) We have designed an agent-based parallel programming environment.

(2) We have designed a parallel parsing algorithm.

We propose an efficient programming environment implemented on shared-memory parallel machines for developing parallel NLP systems based on typed feature structures (TFSs; the TFS (Carpenter 1992) is a basic unit of the HPSG formalism). We call it the " parallel substrate for TFS (PSTFS)" environment. It has many computational agents running on dif-

ferent processors in parallel; these agents communicate with each other using messages. The system tasks such as parsing and semantic processing are divided into several pieces, which are simultaneously computed by several agents. We use an agent-based architecture (Agha 1986; Yonezawa and Ohsawa 1988; Taura 1997) so that we can develop an efficient parallel HPSG parser with ease. Using this architecture, we can divide the development of a parallel HPSG parser into the parsing algorithm itself and the processing of the TFSs. Another way to develop parallel NLP systems with TFSs is to use a fully concurrent logic programming language (Clark and Gregory 1986; Ueda 1985). However, we have noted that parallelism should be controlled in a flexible way with deep analyses and consideration to achieve high performance. (The fixed concurrency in logic programming does not provide sufficient flexibility.) The PSTFS environment is suitable for achieving such flexibility.

As the basis of our parallel HPSG parsing algorithm, we chose the CKY algorithm (Kasami 1965; Younger 1967). A parallel CKY algorithm is desirable from the viewpoints of speed-up, data distribution, and memory efficiency. Several parallel parsing algorithms have been developed, but most of them are neither efficient nor practical enough (Adriaens and Hahn 1994; Nijholt 1994; Grishman and Chitrao 1988; Thompson 1994). The efficiency of our algorithm was shown through experiments.

We describe how to write programs in the PSTFS environment and the mechanism used to achieve efficiency in Section 2. Section 3 describes our CKY-style parallel HPSG parsing algorithm, and its performance is shown in Section 4 through a series of experiments. The performance and time complexity of our parser is discussed in Section 5.

2 Parallel Substrate for Typed Feature Structures (P-STFS)

The PSTFS is an efficient parallel substrate for TFS processing and provides an agent-based programming environment. An agent is a unit of parallelism, encapsulation, data distribution, and mutual exclusion. Each agent runs in parallel and synchronizes itself with other agents by sending and receiving messages.

From the programmer's viewpoint, PSTFS has two types of agents:

- constraint solver agents (CSAs).
- control agents (CAs)

The CSAs are carefully designed to attain efficient communication for passing messages containing TFSs and efficient processing for the TFSs. The CAs have overall control of a sys-

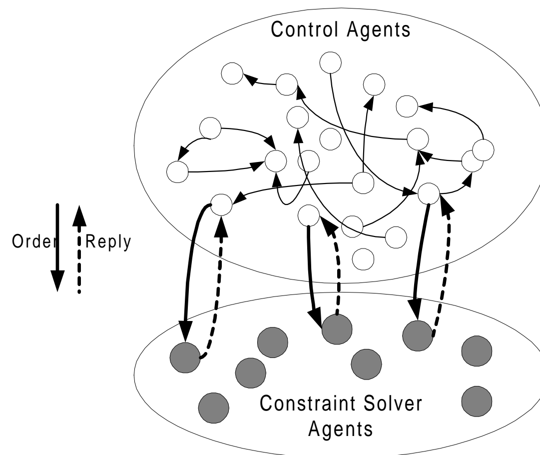


Fig. 1 Overview of PSTFS environment

tem, including control of parallelism, and they behave as masters of the CSAs (see Figure 1).

When a CA needs to process TFSs, the TFSs are transferred to CSAs by sending messages containing the TFSs, and then the CSAs process them according to the messages. Note that the CAs can neither modify nor generate TFSs by themselves. Suppose that one is trying to implement a parsing system based on PSTFS. The CAs correspond to an abstracted parsing algorithm, and the CSAs correspond to the application of phrase structure rules. By "abstracted parsing algorithm" we mean a high-level description of a parsing algorithm in which the application of phrase structure rules is regarded as an atomic operation or a subroutine.

The keys to achieving efficiency for processing and passing TFSs are i) having the CSAs work independently in parallel, ii) transferring the TFSs as IDs and actually transferring the actual images of the TFSs only after the IDs have reached the CSAs requiring the TFSs, and iii) transferring the TFSs via shared-memory.

Programming in the PSTFS environment is described in Section 2.1, and the PSTFS architecture is described in Section 2.2.

2.1 Programming in PSTFS environment

The programming language for PSTFS is quite simple and natural; it was carefully designed to provide both high-performance and ease of programming.

PSTFS was implemented by combining two existing programming languages: the actor-based concurrent programming language ABCL/*f* (Taura 1997) and the TFS-based sequential programming language LiLFeS (Makino et al. 1998). Descriptions of CAs are written in AB-

define-CSA-begin

$$name(\begin{bmatrix}\text{FIRST} & \textit{Franz} \\ \text{LAST} & \textit{Schubert}\end{bmatrix}).$$

$$name(\begin{bmatrix}\text{FIRST} & \textit{Johann} \\ \text{LAST} & \textit{Bach}\end{bmatrix}).$$

...

...

$$concatenate_name(X, Y) \leftarrow X = \begin{bmatrix}\text{FIRST} & \boxed{1} \\ \text{LAST} & \boxed{2}\end{bmatrix}, \quad Y = \begin{bmatrix}\text{FULL} & \langle \boxed{1}, \boxed{2} \rangle \\ \text{FIRST} & \boxed{1} \\ \text{LAST} & \boxed{2}\end{bmatrix}.$$

Fig. 2 Example *concatenate_name*: description of CSA

define-CA *name-concatenator* $\mathcal{NC}$
When message "`active`" arrives, **do**
  $R := \emptyset$;
  $F :=$ **wait-for-result** ($CSA \Leftarrow$ "$\{\varphi|\texttt{name}(\varphi)\}$"); ..... (A)
  $Tasks := \emptyset$;
  forall $\varphi \in F$ **do**
    $r := (\textit{random-integer})$ mod (# of processors);
    $Tasks := Tasks \cup (\mathcal{CSA}_r \Leftarrow$ "$\{\psi|\texttt{concatenate_name}(\varphi, \psi)\}$"); ..... (B)
  end-forall
  $R :=$ **wait-for-result** ($Tasks$);
  return R;

Synchronization between agents is done using two functions, $\Leftarrow$ and **wait-for-result**.
$\Leftarrow$ is a function with two arguments. "*agent* $\Leftarrow$ *message*" means to assign a task to the *agent* by sending a *message*; this function returns an ID tag that identifies the task, e.g., $Tasks$ in this example is a set of tags.

"**wait-for-result** *task-IDs*" means waiting for the results of the tasks that are identified by *task-IDs*.

Fig. 3 Example *concatenate_name*: description of CA

CL/*f*, while descriptions of CSAs are mainly written in LiLFeS.

Figures 2 and 3 show examples of PSTFS coding. The task is to concatenate the first and last names in a given list. The CA in this example (Figure. 3) is called a *name-concatenator*. It gathers pairs of first and last names by sending a CSA the message "$\{\varphi|name(\varphi)\}$" (See (A) in Figure 3). When the CSA receives this message, it regards it as a Prolog-like query in LiLFeS[1], and process it according to the code of a CSA (Figure 2). There are several facts with the predicate $'name'$. When the message "$\{\varphi|name(\varphi)\}$" is processed by a CSA, all possible answers defined by these facts are returned. The obtained pairs are stored in variable

1 LiLFeS supports definite clause programs, the TFS version of Horn clauses.

$$F = \{ \begin{bmatrix} \text{FIRST} & \textit{Franz} \\ \text{LAST} & \textit{Schubert} \end{bmatrix}, \begin{bmatrix} \text{FIRST} & \textit{Johann} \\ \text{LAST} & \textit{Bach} \end{bmatrix}, \ldots\ldots \}$$

$$R = \{ \begin{bmatrix} \text{FULL} & \langle \textit{Franz, Schubert} \rangle \\ \text{FIRST} & \textit{Franz} \\ \text{LAST} & \textit{Schubert} \end{bmatrix}, \begin{bmatrix} \text{FULL} & \langle \textit{Johann, Bach} \rangle \\ \text{FIRST} & \textit{Johann} \\ \text{LAST} & \textit{Bach} \end{bmatrix}, \ldots\ldots \}$$

Fig. 4 Example *concatenate_name*: values of F and R

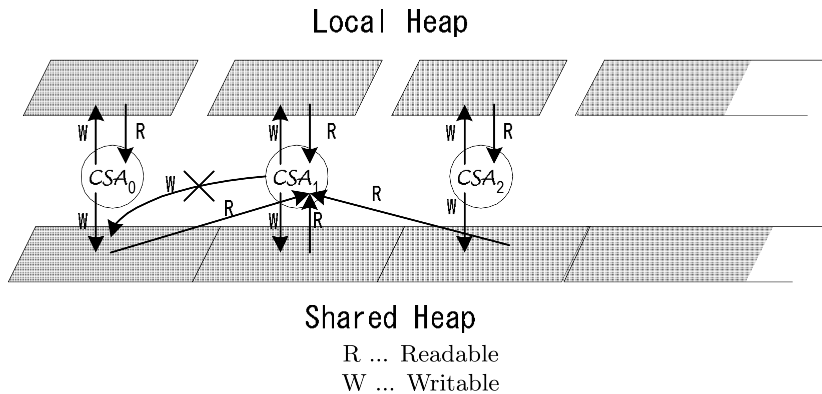
F in the *name-concatenator* (Figure 4).

The *name-concatenator* agent next sends the message `concatenate_name` with a TFS to CSAs (See (B) in Figure 3). The message contains one of the TFSs in F. Each CSA concatenates the value of FIRST with the value of LAST in the received TFS by using the definite clause *concatenate_name* given in Figure 2. The CSAs can basically perform concatenation in parallel and independently. The result is returned to the *name-concatenator* that requested the job. The *name-concatenator* places the returned values into variable R.

The CA *name-concatenator* controls the overall process. It controls parallelism by sending the messages. The operations on the TFSs are performed by the CSAs when asked to do so by a CA.

This distinction between CAs and CSAs is a minor factor of writing a sequential parser, but it has a major impact in a parallel environment. For instance, suppose that several distinct agents evoke applications of phrase structure rules against identical data simultaneously, and the applications perform destructive operations on the data. This can cause an anomaly because the agents will modify the original data in an unpredictable order, so there is no way to maintain consistency. To avoid this problem, one has to determine what is an atomic operation and how to provide a method to prevent anomalies when atomic operations are evoked by several agents. In our framework, any action taken by a CSA is viewed as an atomic operation, so no anomaly can occur, even if CSAs concurrently perform operations on identical data. This is done by copying the TFSs, which does not require any destructive operation. The details are described in Section 2.2.

The other implication of the distinction between CAs and CSAs is that agents can efficiently communicate in a natural way. During parsing in HPSG, TFSs with hundreds of nodes can be generated. Encoding such TFSs in a message and sending them efficiently are not trivial operations. PSTFS provides a communication scheme that enables efficient sending and receiving of such TFSs. This is made possible by the distinction between agents. More precisely, since CAs cannot modify a TFS, they do not have to have an actual image of a TFS.



Each CSA is assigned a portion of the shared heap, and CSAs can write TFSs only to the assigned portion. Though the writable portion is limited, each CSA can read any portion of the shared heap.

Fig. 5 Architecture of PSTFS

When a CSA returns the results to the CA, it only has to send the IDs of the TFSs. Only when the ID is transferred to other CSAs and they try to modify the TFS with the ID, does the actual transfer of the TFS's actual image occur. Since the transfer is carried out only between CSAs, it can be directly and efficiently performed by using a low-level representation of the TFSs used in the CSAs. Note that if CAs modified TFSs directly, this communication scheme could not be used.

2.2 Constraint Solver Agent: PSTFS Architecture

This section explains the architecture of PSTFS, focusing on the execution mechanism of CSAs[2]. A CSA is implemented by modifying the LiLFeS abstract machine (LiAM), which is an abstract machine for TFSs, originally designed for executing LiLFeS programs (Makino et al. 1998).

The important constraint in designing the architecture of PSTFS is that TFSs generated by CSAs must be preserved unmodified. This is because they are used by several agents simultaneously. If a TFS were modified by a CSA and if other agents did not know this, the expected results would not been obtained. Note that unification, which is a major operation on TFSs, is a destructive operation[3]. If many agents try to unify identical TFSs simultaneously without any mechanism, the modifications would occur simultaneously. Our execution mechanism solves this problem by letting CSAs copy the TFSs each time they try to modify the TFSs. Though this may not look efficient at first, it is performed efficiently by using

2 See (Taura 1997) for further details about CAs
3 To be precise, it is difficult to unify a TFS efficiently without destructive operation

shared-memory mechanisms and our copying methods.

A CSA uses two different types of memory areas as its heap (Figure 5):

- Local heap
- Shared heap

The local heap is used for temporary operations during computation inside the CSA. A CSA can neither read nor write to the local heaps of other CSAs. The shared heap is used as a medium of communication between CSAs, and it is implemented on shared memory inside parallel machines. When a CSA completes a computation on a TFS, it writes the result to the shared heap. Since the shared heap can be read by any CSA, each CSA can read the results of operations by other CSAs. However, we limit the portion of the shared heap to which the CSA can write. Other CSAs cannot write to that portion.

Next, we look at the steps performed by a CSA when it receives a message from a CA. Note that the message contains only the IDs of the TFSs as described in Section 2.1. The IDs are given as pointers on the shared heap. See Figure 6.

(i) A CA sends a message to a CSA.

(ii) The CSA copies the TFSs to which the IDs in the message point from the shared heap to the local heap of the CSA.

(iii) The CSA processes a Prolog-like query by using LiAM on the local heap.

(iv) If the query receives an answer, the answer is copied to the portion of the shared heap writable by the CSA. The CSA evokes backtracking in LiAM, keeping the IDs of the copied TFSs, and goes to Step (iii). If there is no answer, it goes to Step (v).

(v) The CSA sends a message with the IDs of the resulting TFSs back to the requesting CA.

Note that the results of the computation become readable by other CSAs after step (v).

To sum up, this procedure has the following desirable features.

Simultaneous copying A TFS on a shared heap can be copied by several CSAs simultaneously. This is due to our shared memory mechanism and the property of LiAM that copying does not have any side-effect on TFSs[4].

Simultaneous/Safe writing CSAs can write on their own shared heap without the danger of accidental modification by other CSAs.

Demand-driven copying As described in Section 2.1, the transfer of actual images of

4 Actually, this is not trivial. A TFS is a graph structure. Therefore, during a copy operation, marking the traversed region in a TFS is required to detect structure sharing. Note that marking is a destructive operation. In our approach, TFSs are stored into a continuous region on a shared heap during copying in Step (iv). TFSs stored in a continuous region can be copied efficiently without any side-effect because such TFSs can be copied with a simple loop procedure.

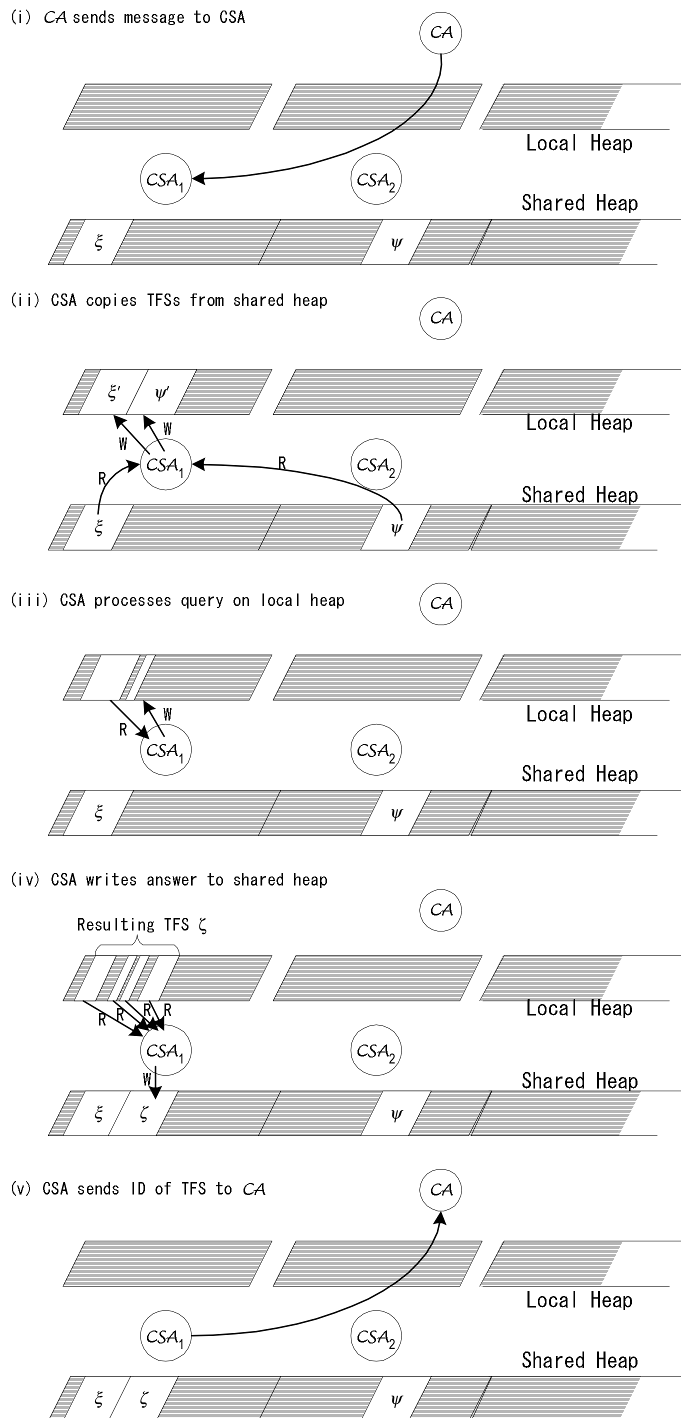


Fig. 6 PSTFS mechanism

TFSs is performed only after the IDs of the TFSs reach the CSAs requiring the TFSs. Redundant copying/sending of actual TFS images is reduced, and the transfer is performed efficiently by mechanisms originally provided by LiAM[5].

With efficient data transfer in shared-memory machines, these features reduce the overhead of parallelization.

The PSTFS mechanism seems to require a lot of memory space on the shared heap because the TFSs to be processed in parallel must be copied to the shared heap first. However, in many cases of developing NLP systems, such a problem doesn't arise because we have to keep the resulting TFSs to support non-determinism even in the sequential NLP systems. For instance, in a chart parsing for a unification-based grammar, intermediate parse trees must be preserved untouched. In the case of our parallel HPSG parser, the heap size required by the parallel parser and that required by the sequential parser are the same. In general, destructive operations on the results are done after copying them. The copying of TFSs in the above steps achieves such mechanisms naturally, though it was originally designed for efficient support of data sharing and destructive operations on shared heaps.

3 Parsing Algorithms

In this section we describe our parallel HPSG parsing algorithm by using CAs and CSAs, i.e., we describe our algorithm in a PSTFS code. First, we explain a sequential CKY-style HPSG parsing algorithm (Haas 1987), which our parallel HPSG parsing algorithm is based on. Our agent-based parallel HPSG parsing algorithm is described using mainly CAs because the parallelism of CSAs is controlled by CAs.

3.1 CKY-style Sequential Parsing Algorithm for HPSG

To simplify our discussion, we assume that an HPSG grammar consists of a *Lexicon* and a *RuleSchemata*. A *Lexicon* is a finite set of lexical entries in the form $(\omega \rightarrow w)$, where ω is a TFS and w is a word. A *RuleSchemata* is a finite set of *RuleSchema*, which correspond to a rewriting rule in CFG and represents the structural relation and constraints between a mother and her daughters in a parse tree. Our parsing algorithm for HPSG is based on the CKY algorithm for CFG. In a CKY-style algorithm, *RuleSchema* ρ is given in the form

$$\rho = \begin{bmatrix} MOTHER & \gamma \\ DTRS & \langle \alpha, \beta \rangle \end{bmatrix}$$

5 LiAM has a function to store TFSs into a continuous region on a heap. Such TFSs can be copied efficiently with a simple loop procedure.

```
procedure initialize ()
  for j := 2 to n do
    for i := j − 2 to 0 do
      S_{i,j} := ∅;
    end-for
  end-for
```

```
procedure parse ()
  forall 1 ≤ j ≤ n do
    S_{j−1,j} := {ω|ω → w_j};
  end-forall

  for j := 2 to n do ...................................... (loop A)
    for i := j − 2 to 0 do .............................. (loop B)
      for k := i + 1 to j − 1 do ..................... (loop C)
        foreach φ ∈ S_{i,k} do .......................... (loop D)
          foreach ψ ∈ S_{k,j} do ...................... (loop E)
            foreach ρ ∈ RuleSchemata do .. (loop F)
              W := rule-schema(ρ, φ, ψ);
              S_{i,j} := S_{i,j} ∪ W;
            end-foreach
          end-foreach
        end-foreach
      end-for
    end-for
  end-for
```

Fig. 7 Sequential CKY-style HPSG parsing algorithms

, where ρ is a TFS, the value γ followed by the feature $MOTHER$ corresponds to a mother, and the value $\langle\alpha, \beta\rangle$ followed by the feature $DTRS$ corresponds to her daughters (α corresponds to a left daughter, and β corresponds to a right daughter).

Given $w_1 w_2 \cdots w_n$ as a sentence and $i, j (0 \leq i < j \leq n)$, we define $S_{i,j}$ as follows:

$$
\begin{aligned}
S_{i-1,i} &\equiv \{\omega | (\omega \rightarrow w_i) \in Lexicon\} \\
S_{i,j} &\equiv \{\gamma | \exists k \delta \epsilon \varphi \psi \rho, i < k < j, \rho \in RuleSchemata, \\
&\quad \varphi \in S_{i,k}, \psi \in S_{k,j}, \rho \sqcup \begin{bmatrix} DTRS & \langle\varphi, \psi\rangle \end{bmatrix} = \begin{bmatrix} MOTHER & \gamma \\ DTRS & \langle\delta, \epsilon\rangle \end{bmatrix} \}
\end{aligned}
$$

Intuitively, $S_{i,j}$ corresponds to the set of the sub-parse-trees whose leaves are $w_{i+1}, \cdots, w_j$.

A sequential version of this algorithm is shown in Figure 7. "$rule\text{-}schema(\rho, \varphi, \psi)$" returns all possible γ computed by unifying α with φ and β with ψ for $\rho = \begin{bmatrix} MOTHER & \gamma \\ DTRS & \langle\alpha, \beta\rangle \end{bmatrix}$.

Conventionally, $S_{i,j}$ is represented as a member of a two-dimensional table called a CKY table. In the example shown in Figure 8, the numbers i, j in each cell correspond to i, j of $S_{i,j}$. The table corresponds to a parse tree whose root is an element of $S_{0,n}$. $S_{i,j}$ for all

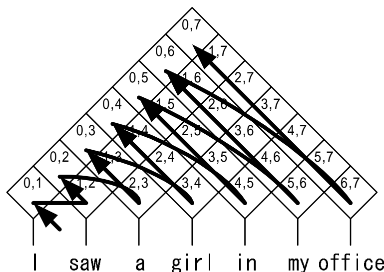


Fig. 8 CKY table; sequential parsing algorithm proceeds along the arrows.

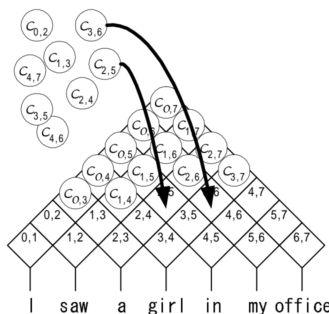


Fig. 9 Correspondence between CKY table and $\mathcal{C}_{i,j}$

$0 \leq i < j \leq n$ can be computed in a bottom-up manner toward the arrows. Parsing completes when computation of $S_{0,n}$ completes.

3.2 CKY-style Parallel Parsing Algorithm for HPSG

We describe our CKY-style parallel HPSG parsing algorithm in an agent-based programming language style. The algorithm is shown in Figure 10 and Figure 11.

In our algorithm, all $S_{i,j}$ are computed in parallel. Parsing starts by creating a CA called $\mathcal{PARSER}$. $\mathcal{PARSER}$ creates CAs, which we call *cell-agents*, and distributes them to processors on parallel machines (Figure 9). For convenience of explanation, we transcribe *cell-agents* as $\mathcal{C}_{i,j}(0 \leq i < j \leq n)$.

Each $\mathcal{C}_{i,j}$ computes $S_{i,j}$ in parallel. To be more precise, $\mathcal{C}_{i,j}(j-i=1)$ looks up the word in *Lexicon* and obtains lexical entries. $\mathcal{C}_{i,j}(j-i>1)$ waits for the messages containing $S_{i,k}$ and $S_{k,j}$ for all $k(i<k<j)$ from other *cell-agents*.

When $\mathcal{C}_{i,j}$ receives $S_{i,k}$ and $S_{k,j}$ for an arbitrary k, it computes TFSs by applying rule

define-CA *parser-agent* $\mathcal{PARSER}$
When message `parse`$(w_1, w_2, \ldots, w_n)$ arrives, **do**
  forall $i(0 < i \leq n)$ **do**
    create-agents *cell-agent* $\mathcal{C}_{i-1,i}$;
    $\mathcal{C}_{i-1,i} \Leftarrow$ "`active-as-leaf`"
  end-forall
  forall $i, j(0 \leq i, j - i > 1, j \leq n)$ **do**
    create-agents *cell-agent* $\mathcal{C}_{i,j}$;
    $\mathcal{C}_{i,j} \Leftarrow$ "`active-as-phrase`"
  end-forall
  wait-for-result $S_{0,n}$.

Fig. 10 CKY-style parallel parsing algorithm for HPSG: $\mathcal{PARSER}$

schemata to each pair of members of $S_{i,k}$ and $S_{k,j}$. The computed TFSs are considered to be mothers of members of $S_{i,k}$ and $S_{k,j}$, and they are added to $S_{i,j}$. Note that these applications of rule schemata are done in parallel by several CSAs[6].

Finally, when the computation of $S_{i,j}$ (by using $S_{i,k}$ and $S_{k,j}$ for all $k(i < k < j)$) completes, $\mathcal{C}_{i,j}$ distributes $S_{i,j}$ to other agents waiting for $S_{i,j}$.

Parsing completes when the computation of $S_{0,n}$ completes.

To perform this parallelization, we must synchronize the computation of $S_{i,j}$. In our algorithm, the following *Dependency* always holds.

Dependency Computation of $T_{i,k,j}$ must start after computation of both $S_{i,k}$ and $S_{k,j}$ completes, where $T_{i,k,j}$ is a subset of $S_{i,j}$ and is defined as

$$T_{i,k,j} \equiv \{\gamma | \exists \alpha\beta\varphi\psi\rho, \rho \in RuleSchemata,$$
$$\alpha \in S_{i,k}, \beta \in S_{k,j}, \rho \sqcup \begin{bmatrix} DTRS & \langle \alpha, \beta \rangle \end{bmatrix} = \begin{bmatrix} MOTHER & \gamma \\ DTRS & \langle \varphi, \psi \rangle \end{bmatrix} \}$$

As long as *Dependency* holds, each $S_{i,j}$ can be computed in parallel in any order[7]. In our algorithm, *Dependency* always holds because i) each cell-agent computes $T_{i,k,j}$ after the arrival of both $S_{i,k}$ and $S_{k,j}$, as illustrated in Figure 11(A), and ii) each cell-agent distributes $S_{i,j}$ to the agents requesting $S_{i,j}$ after completing the computation of $S_{i,j}$.

We call the loops from the outermost loop to the innermost loop "**loop A, B, C, D, E, F**" in Figure 7. The parallelization of our algorithm corresponds to the parallelization of loops A, B, D and E, and the inside of loop C is processed in the order that computation of both

6 CSAs cannot be added dynamically in our implementation. Therefore, to achieve maximum parallelism, we assigned a CSA to each processor. Each $C_{i,j}$ asks the CSA on a randomly selected processor to apply rule schemata.

7 Of course, we can give finer grained dependency. For example, we can start computation of $T_{i,k,j}$ immediately after one of the members of $S_{i,k}$ and one of the members of $S_{k,j}$ are computed. But an algorithm that follows such a dependency condition greatly increases the number of messages passed. This increases the overhead for message passing, so complex data-flow control is required.

```
define-CA cell-agent C_{i,j}
initial-values : S_{i,j} := ∅; NTASK := j − i − 1;

When message "active-as-phrase" arrives, do
  forall k(i < k < j) do
    C_{i,k} ⇐ "request S_{i,k}";
    C_{k,j} ⇐ "request S_{k,j}";
  end-forall

When both S_{i,k} and S_{k,j} for some k arrive, do ....... (A)
  Tasks := ∅;
  forall φ ∈ S_{i,k} do
    forall ψ ∈ S_{k,j} do
      r := (random-integer) mod (# of processors);
      Tasks := Tasks ∪ (CSA_r ⇐ "{ξ|rule-schema(φ, ψ, ξ)}");
    end-forall
  end-forall
  S_{i,j} := S_{i,j} ∪ wait-for-result (Tasks);
  NTASK := NTASK − 1;
  if NTASK = 0 then
    send S_{i,j} to all agents requesting S_{i,j};
  end-if

When message "active-as-leaf" arrives, do
  r := (random-integer) mod (# of processors);
  S_{i,j} := wait-for-result (CSA_r ⇐ "{ω|lexical-entry(w_j, ω)}");
  send S_{i,j} to all agents requesting S_{i,j};
```

Fig. 11 A CKY-style parallel parsing algorithm for HPSG: $\mathcal{C}_{i,j}$

$S_{i,k}$ and $S_{k,j}$ completes.

Our parallel parsing algorithm is an HPSG version of the parallel CKY parsing algorithm for CFG (Ninomiya, Torisawa, Taura, and Tsujii 1997). The differences between these two algorithms are as follows. 1) The parallel HPSG parsing algorithm deals with the TFSs, but the parallel CFG parsing algorithm deals with the nonterminals. Owing to the PSTFS environment, we can neglect the difference between TFSs and nonterminals in the description of the parsing algorithm. 2) These two algorithms are different in the number of loops which are parallelized. We parallelized four loops (loop A, B, D and E in Figure 7) in the parallel HPSG parsing algorithm, but in the parallel CFG parsing algorithm, only two loops (loop A and B) are parallelized. In general, parallel processing of too much fine-grained tasks decreases the system performance because the ratio of "the overhead caused by the parallel processing of each task" to "the execution time of each task" increases. The key to achieving the efficiency

in parallel processing is to find a good parallelism where the overhead and the execution time of each task are balanced. In the parallel CFG parsing algorithm, the execution time for the CFG rule application is extremely short, and hence it is difficult to parallelize 2 more loops (loop D and E) because the ratio of the overhead increases greater than the balanced ratio. In the parallel HPSG parsing algorithm, parallelization of two loops was insufficient because the execution time of the HPSG rule application is longer than that of the CFG rule application. We have implemented and experimented parallelization of two loops (loop A and B), four loops (loop A, B, D and E) and five loops (loop A, B, D, E and F) in the parallel HPSG parsing algorithm, and we observed that the parallelization of four loops was the most efficient.

Another parallel CKY algorithm was proposed by Nijholt (Nijholt 1994). The most significant differences between ours and his are as follows. 1) Nijholt's algorithm is based on data-flow computation, and the output of $S_{i,j}$ is passed only from one cell to the adjacent cells in the two-dimensional table, i.e., $S_{i,j}$ is passed to the processor that computes $S_{i-1,j}$ and to the processor that computes $S_{i,j+1}$, whereas in our algorithm, $S_{i,j}$ is passed directly to $S_{i,k}(j < k \leq n)$ and $S_{k,j}(0 \leq k < i)$. 2) In Nijholt's algorithm, $S_{i,j}$ is passed and received in a statically predicted order, whereas in our algorithm, $S_{i,k}$ is passed when its computation completes, and $T_{i,k,j}$ is processed as soon as computation of $S_{k,j}$ is completed. From these viewpoints, our algorithm is more efficient than Nijholt's.

4 Performance Evaluation

We tested our CKY-style parallel HPSG parser on a shared-memory parallel machine, a SUN Ultra Enterprise 10000 consisting of 64 nodes (each node is a 250-MHz UltraSparc processor) and 8-GB shared memory. The grammar we used is an underspecified Japanese HPSG grammar (Mitsuishi et al. 1998) developed in September 1998. It consists of 6 ID-schemata and 39 lexical entries (most of them are functional words) and 41 lexical-entry-templates (assigned to parts of speech). This grammar has wide coverage and high accuracy for real-world texts[8]. The corpus consists of randomly selected 881 sentences from the EDR Japanese corpus (the average sentence length is 20.8)[9].

Table 1 shows the average parsing time, along with that for a parser written in LiLFeS. As

8 This grammar can generate parse trees for 82% of 10,000 sentences from the EDR Japanese corpus; its bunsetsu dependency accuracy is 78%. The current version of the grammar was developed in October 1999. It consists of 134 lexical entries and 53 lexical-entry-templates and can generate parse trees for 98.7% of 2,024 sentences from the EDR Japanese corpus. Its bunsetsu dependency accuracy with a statistical model is 88.6% (Kanayama, Torisawa, Mitsuishi, and Tsujii 2000)

9 We chose 1,000 randomly selected sentences from the EDR Japanese corpus; the 881 sentences we used were all parsable by the grammar.

Number of Processors	Avg. Parsing Time (msec)	
	PSTFS	LiLFeS
1	1,029.9	991
10	229.5	N/A
20	123.0	N/A
30	95.8	N/A
40	82.4	N/A
50	78.0	N/A

Table 1 Average parsing time per sentence in EDR Japanese corpus

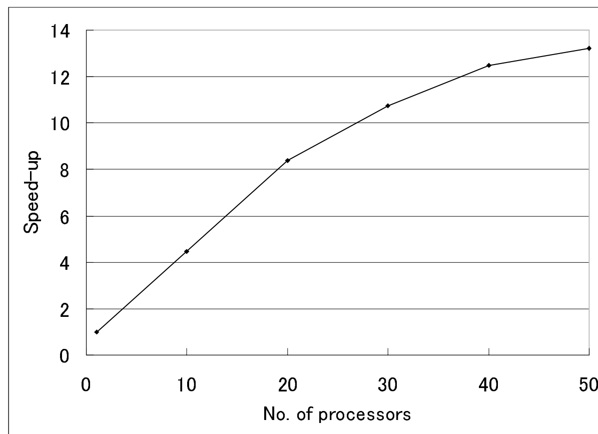


Fig. 12 Speed-up of parsing time on a parallel CKY-style HPSG parser

shown in Figure 12, the maximum speed-up reached 13.2. The average parsing time was 78 msec per sentence[10][11].

The parsing time of our parser reached the level required by real-time applications, though we used computationally expensive grammar formalisms, i.e., HPSG with reasonable coverage and accuracy. This shows the feasibility of our parallel HPSG parser and PSTFS. In addition, our parallel HPSG parser was considerably more efficient than the sequential HPSG parser written in LiLFeS[12]. As Makino et al. reported (Makino et al. 1998), a sequential parser writ-

10 We couldn't measure parsing time with 60 processors because performance was degraded extremely. In general, when the number of processes is near or more than the number of existing processors, context switching between processes occurs frequently on shared-memory parallel machines. We believe the cause for the inefficiency when we used 60 processors lies in such context switches. We thus excluded experiments with 60 processors so that we could evaluate the performance of our algorithm precisely.

11 Morphological analysis time is excluded from this parsing time. The average morphological analysis time was 36 msec, so the overall parsing time was around 114 msec.

12 At present, our group has a much more efficient sequential parser called the TNT parser (Nishida, Torisawa, and Tsujii 1999). Unfortunately, because of chronological inconsistencies in systems and grammars, we cannot

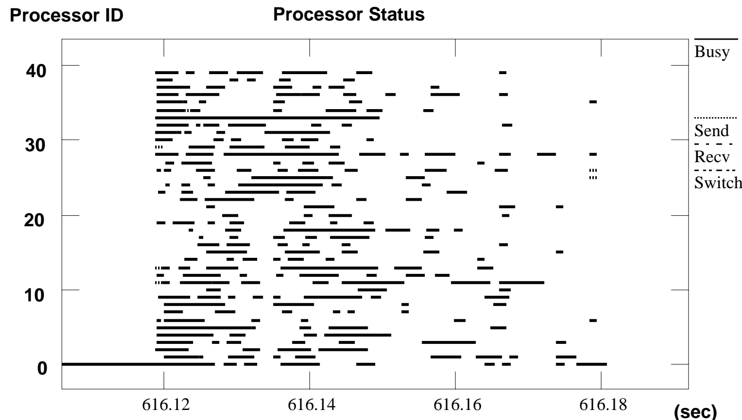


Fig. 13 Processors status

ten in LiLFeS is more efficient than other existing parsers (Carpenter and Penn 1994; Erbach 1994); we can thus say that our parser is more efficient than other existing parsers.

However, as shown in Figure 12, speed-up was not proportional to the number of processors. We think this is because parallelism is not fully extracted in our parsing algorithm. Figure 13 shows processor status during parsing of a Japanese sentence by our parallel parser. The black lines indicate busy periods. One can see that many processors were frequently idle.

This idle time does not suggest that parallel NLP systems are inefficient. Instead, it suggests that parallel NLP systems have many possibilities. If we introduce semantic processing, for instance, overall processing time may not change because the idle time can be used for the semantic processing. Another possibility is the use of parallel NLP systems as servers. Even if we feed several sentences at a time, throughput will not change because the idle time can be used for parsing different sentences.

5 Discussion

We discuss how our parallel HPSG parser works from the following viewpoints, i) the region where *cell-agents* are working in the CKY table at any given moment, and ii) the worst-case time complexity, theoretically and empirically.

5.1 Active Region in CKY Table

We consider the region where $S_{i,j}$ is being computed in a CKY table at any given moment.

compare performance between our parallel parser and the TNT parser.

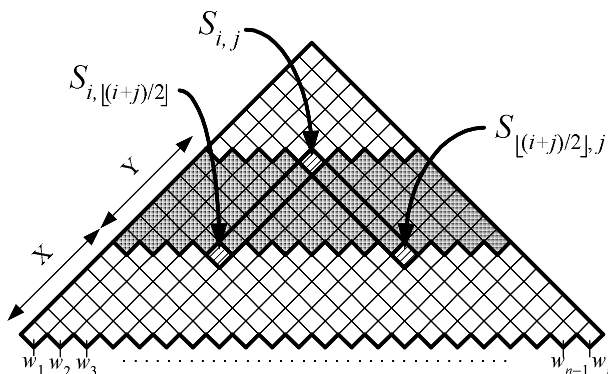


Fig. 14 $S_{i,j}$ being computed in CKY table

The start and complete times of the computation of $S_{i,j}$ have the following properties:

Start Time Property Computation of $S_{i,j}$ can start after one of the pairs (for some $k(i < k < j)$, $\langle S_{i,k}, S_{k,j}\rangle$) becomes available.

Complete Time Property Computation of $S_{i,j}$ completes when the computation of $T_{i,k,j}$ for all $k(i < k < j)$ completes.[13]

We also define the word *pair* and *available*.

Pair We call $\langle S_{i,k}, S_{k,j}\rangle$ a pair for $S_{i,j}$.

Available When the computation of both $S_{i,k}$ and $S_{k,j}$ has completed, "the pair $\langle S_{i,k}, S_{k,j}\rangle$ is available for computing $S_{i,j}$".

For reasons of symmetry and independency of $S_{i,j}(j-i=C)$, which are arranged on the same horizontal line in a CKY table, computation of $S_{i,j}(j-i=C)$ will complete at the same time if the amount of computation for each $S_{i,j}$ is equal (an ideal condition). In such a condition, from the complete-time property, the computation of $S_{i,j}$ proceeds from $S_{i,j}(j-i=1)$ to $S_{0,n}$, line by line, in the CKY table. The first pair to be available for computation of $S_{i,j}$ is $\langle S_{i,k}\ S_{k,j}\rangle$ with k close to $(i+j)/2$. The region where $S_{i,j}$ is being computed at any given moment is the shaded portion of Figure 14; the width of the shaded portion (the length of Y in the figure) is the same as the width of the portion where $S_{i,j}$ has been computed (the length of X in the figure). For that reason, the shaded portion widens as the parsing proceeds, so the number of processes running in parallel increases as the parsing proceeds.

5.2 Theoretical Time Complexity

In this section, we explain the worst-case time complexity of our HPSG parsing algorithm.

13 The definition of $S_{i,j}$ is given in Section 3.1 and the definition of $T_{i,k,j}$ is given in Section 3.2.

Let $f(n)$ be the worst-case time complexity to compute $T_{i,k,j}$ for a sentence with length n. The time complexity of our parallel HPSG parsing algorithm is less than $2(n-1)f(n)+D$ (D is constant) for a sentence with length n under the following assumptions, while that of a sequential algorithm is less than $\frac{1}{6}n(n^2-1)f(n)+D$[14].

Assumption 5.1 *The number of processors is infinite.*

Assumption 5.2 *There is no overhead for parallel processing or distributing messages.*

Definition 5.1

$$\begin{aligned}
S_d &\equiv \{S_{i,j} | j-i=d\} \\
A_{i,j}(t) &\equiv \textit{set of available pairs } \langle S_{i,k}, S_{k,j}\rangle \textit{ for computing } S_{i,j} \textit{ until time } t. \\
C_{i,j}(t) &\equiv \textit{set of } T_{i,k,j} \textit{ whose computation has completed until time } t. \\
t_d &\equiv \textit{time when computation of all members of } S_d \textit{ completes.}
\end{aligned}$$

Lemma 5.1 *For all $S_{i,j}(j-i=d, d$ is even$)$,*

$$|A_{i,j}(t_x)| \geq \begin{cases} 0 & x < \frac{d}{2} \\ 2x-d+1 & \frac{d}{2} \leq x < d \\ d-1 & d \leq x \end{cases}$$

Proof When $x < \frac{d}{2}$ and $d \leq x$, it is easy to see that the lemma above holds. In the case of $\frac{d}{2} \leq x < d$, we provide inductive proof.

Basis When $x = \frac{d}{2}$, computation of $S_{i,i+\frac{d}{2}}$ and $S_{i+\frac{d}{2},j}$ has completed. Therefore, $|A_{i,j}(t_{\frac{d}{2}})|$ is greater than 1.

Induction Assume that when $x = \frac{d}{2}+h$, $A_{i,j}(t_x) \supset \{\langle S_{i,i+\frac{d}{2}\pm e}, S_{i+\frac{d}{2}\pm e,j}\rangle | 0 \leq e \leq h\}$. That is, $|A_{i,j}(t_x)| \geq 2h+1$.

When $x = \frac{d}{2}+h+1$, $\langle S_{i,i+\frac{d}{2}\pm(h+1)}, S_{i+\frac{d}{2}\pm(h+1),j}\rangle \in A_{i,j}(t_x)$ from the definition of t_x. Therefore, $A_{i,j} \supset \{\langle S_{i,i+\frac{d}{2}\pm e}, S_{i+\frac{d}{2}\pm e,j}\rangle | 0 \leq e \leq h+1\}$. Therefore, $|A_{i,j}(t_x)| \geq 2(h+1)+1$.

□

Lemma 5.2 *For all $S_{i,j}(j-i=d)$,*

$$|A_{i,j}(t_x)| \geq \begin{cases} 0 & x < \frac{d}{2} \\ 2x-d+1 & \frac{d}{2} \leq x < d \\ d-1 & d \leq x \end{cases}$$

14 Cf. The time complexity of a sequential CFG parser based on the CKY algorithm is $\frac{1}{6}n(n^2-1)C+D = O(n^3)$ because $f(n)$ is constant. In the case of HPSG, $f(n)$ is exponential.

Proof In the same way as the proof of Lemma 5.1, we can prove that, for all $S_{i,j} (j - i = d, d \text{ is odd})$,

$$|A_{i,j}(t_x)| \geq \begin{cases} 0 & x < \frac{d+1}{2} \\ 2x - d + 1 & \frac{d+1}{2} \leq x < d \\ d - 1 & d \leq x \end{cases}$$

$$= \begin{cases} 0 & x < \frac{d+1}{2} - \frac{1}{2} \\ 2x - d + 1 & \frac{d+1}{2} - \frac{1}{2} \leq x < d \\ d - 1 & d \leq x \end{cases}$$

From Lemma 5.1 and above, we proved Lemma 5.2. □

Lemma 5.3 *Let T_g be $2(g-1)f(n) + D$. $t_g \leq T_g$. For all $C_{i,j}(T_g)$ s.t. $(d = j - i)$,*

$$|C_{i,j}(T_g)| \geq \begin{cases} 0 & g < \frac{d}{2} + 1 \\ 2g - d - 1 & \frac{d}{2} + 1 \leq g < d \\ d - 1 & d \leq g \end{cases}$$

Proof

Basis Until t_1, lexical entries are looked up for each word completely in parallel. Therefore, $t_1 \leq D = T_1$. $|C_{i,j}(T_1)| = 0$.

Induction We assume that for some x, $t_x \leq T_x$, and for all $C_{i,j}(T_x)$ s.t. $(d = j - i)$,

$$|C_{i,j}(T_x)| \geq \begin{cases} 0 & x < \frac{d}{2} \\ 0 & \frac{d}{2} \leq x < \frac{d}{2} + 1 \\ 2x - d - 1 & \frac{d}{2} + 1 \leq x < d \\ d - 1 & d \leq x \end{cases}$$

From Lemma 5.2 and the assumption $t_x < T_x$,

$$|A_{i,j}(T_x)| \geq \begin{cases} 0 & x < \frac{d}{2} \\ 2x - d + 1 & \frac{d}{2} \leq x < \frac{d}{2} + 1 \\ 2x - d + 1 & \frac{d}{2} + 1 \leq x < d \\ d - 1 & d \leq x \end{cases}$$

Note that in the case of $(\frac{d}{2} \leq x < \frac{d}{2} + 1)$, $|A_{i,j}(T_x)| (= 2x - d + 1)$ is 1 or 2. Therefore,

$$\inf(|A_{i,j}(T_x)|) = \inf(|C_{i,j}(T_x)|) + \begin{cases} 0 & x < \frac{d}{2} \\ 1 \text{ or } 2 & \frac{d}{2} \leq x < \frac{d}{2} + 1 \\ 2 & \frac{d}{2} + 1 \leq x < d \\ 0 & d \leq x \end{cases}$$

This means that the difference between the lower bound of $|A_{i,j}(T_x)|$ and the lower bound of $|C_{i,j}(T_x)|$ is less than 2. Hence the lower bound of $|C_{i,j}(T_x + 2f(n))|$ is equal to the lower bound of $|A_{i,j}(T_x)|$.

$$
\begin{aligned}
|C_{i,j}(T_{x+1})| = |C_{i,j}(T_x + 2f(n))| &\geq \begin{cases} 0 & x < \frac{d}{2} \\ 2x - d + 1 & \frac{d}{2} \leq x < \frac{d}{2} + 1 \\ 2x - d + 1 & \frac{d}{2} + 1 \leq x < d \\ d - 1 & d \leq x \end{cases} \\
&= \begin{cases} 0 & x + 1 < \frac{d}{2} \\ 0 & \frac{d}{2} \leq x + 1 < \frac{d}{2} + 1 \\ 2(x+1) - d - 1 & \frac{d}{2} + 1 \leq x + 1 < d \\ d - 1 & d \leq x + 1 \end{cases}
\end{aligned}
$$

Then, for $i, j (j - i = x + 1)$, $|C_{i,j}(T_{x+1})| = (x + 1) - 1$. This means that computation of all S_{x+1} completes until T_{x+1}. Hence, $t_{x+1} \leq T_{x+1}$. □

Lemma 5.4 *The worst-case time complexity of parallel parsing for HPSG is less than $2(n-1)f(n) + D$, where n is the sentence length.*

Proof In Lemma 5.3, the parsing process completes when $g = n$. Therefore, $t_n < 2(n-1)f(n) + D$. □

Theoretical Time Complexity of $f(n)$ Let $g(n)$ be the worst-case time complexity for unifying rule schemata with two daughters and $h(n)$ be the worst-case time complexity for merging the equivalent members in a cell of the CKY table. Considering that (i) $f(n)$ is the time complexity for computing the members of $T_{i,k,j}$, (ii) $T_{i,k,j}$ is computed by applying rule schemata to the members of $S_{i,k}$ and the members of $S_{k,j}$, and (iii) these applications of rule schemata for two daughters can be done in parallel[15], $f(n)$ is $g(n) + h(n)$.

5.3 Empirical Time Complexity

Unfortunately, the worst-case time complexity of both $g(n)$ and $h(n)$ is theoretically exponential, and hence the worst-case time complexity of our parallel HPSG parsing algorithm

15 For example, for $\varphi_1, \varphi_2, \varphi_3 \in S_{i,k}$, $\psi_1 \psi_2 \in S_{k,j}$, a parser applies rule schemata to the following pairs, $\langle \varphi_1, \psi_1 \rangle$, $\langle \varphi_1, \psi_2 \rangle$, $\langle \varphi_2, \psi_1 \rangle$, $\langle \varphi_2, \psi_2 \rangle$, $\langle \varphi_3, \psi_1 \rangle$, $\langle \varphi_3, \psi_2 \rangle$. Our parallel parser can process these unifications in parallel.

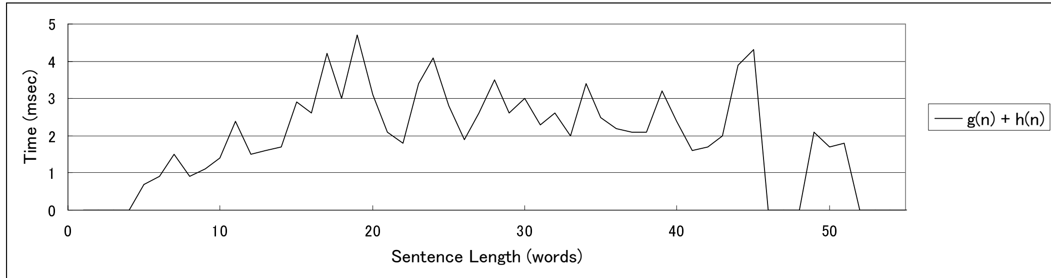


Fig. 15 Empirical time complexity of $f(n)(= g(n) + h(n))$

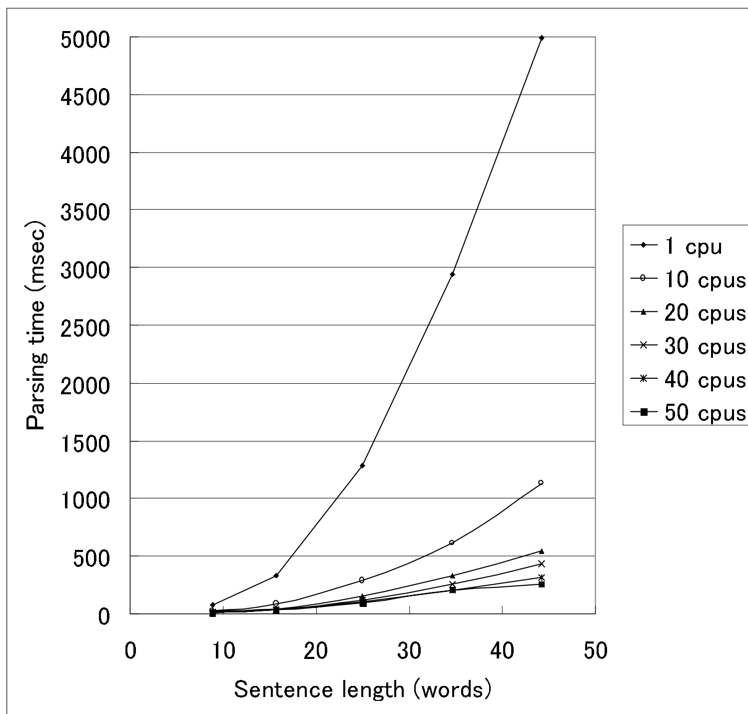


Fig. 16 Parsing time for sentence length n

is also exponential. Though the theoretical worst-case time complexity is exponential, in this section we show that the empirical worst-case time complexity of $f(n)(= g(n) + h(n))$ can be approximated by constant, and we illustrate the parsing time for sentence length n.

Sentence Length (no. of words)	Avg. Sentence Length	Corpus Size (no. of sentences)	Speed-up (*)	Avg. of Parsing Time (msec)					
				1 PE	10 PE	20 PE	30 PE	40 PE	50 PE
1-10	8.9	50	6.38	75.1	22.7	24.3	13.0	12.0	11.8
11-20	15.6	414	10.05	335.0	83.4	44.4	38.1	34.3	33.3
21-30	24.9	293	13.21	1,282.0	287.3	152.9	117.3	104.9	97.1
31-40	34.6	112	14.47	2,940.8	614.2	334.8	254.2	208.0	203.2
41-	44.2	12	19.48	4,994.8	1,129.7	545.1	432.1	310.9	256.4

(*) ... Avg. Parsing Time (1 PE) / Avg. Parsing Time(50 PEs)

Table 2 Average parsing time per sentence

Empirical Time Complexity of $f(n)$ Figure 15 shows the worst-case computing time of $f(n)(= g(n) + h(n))$ for sentence length n[16] Though the empirical value of $f(n)$ seems irregular, it has an upper bound which can be approximated by constant. This empirical time complexity of $f(n)$ implies that the time complexity of parsing $2(n-1)f(n)+D$ can be approximated by $Cn+D$, where C and D are constant[17]. From assumptions 5.1 and 5.2, the more processors we have and the less overhead there is for communications, the closer the parsing time is to linear time.

Empirical Parsing Time Complexity We measured parsing time for each length of sentences. The computing environments, the grammar, and the corpus we used were the same as those used in the experiments described in Section 4. We divided the corpus into several corpora according to the sentence length (Table 2). Table 2 and Figure 16 show the average parsing time for each corpus arranged according to the sentence length. We can see that the more processors we used, the closer the parsing time became to linear time.

6 Conclusion

We have developed an efficient parallel HPSG parser that is practical in terms of both analysis time and speed-up. The key to its efficiency is in our parsing algorithm and the

16 We measured the empirical value of $f(n)$ on a sequential machine, DELL PowerEdge 6350 consisting of 550-MHz PentiumIII processor and 4-GB memory. The grammar used in this experiment was the same grammar used in the experiment in Section 4, but the grammar version was different. The grammar used in this experiment was the current version developed in October 1999 while the grammar used in the experiment in Section 4 was the older version developed in September 1998. The corpus used in this experiment was the same with the corpus used in the experiment in Section 4.

17 The empirical time complexity of $f(n)$ highly depends on the property of the grammar used in the experiment. The grammar we used does not have any semantic features but have only syntactic features. Many of TFSs derived by such a grammar can be merged into one TFS, and therefore the number of TFSs in the CKY table does not grow exponentially. In general, the empirical time complexity of $g(n)$ and $h(n)$ cannot be approximated by constant.

architecture of the PSTFS, a substrate for parallel processing of typed feature structures.

We applied a CKY-style parsing algorithm to the HPSG parser. A parallel CKY algorithm is desirable from the viewpoints of speed-up, distribution of data, and memory efficiency. The main features of PSTFS are efficient communication and a copy scheme for TFSs. This approach is simple without any anomalies, but imposes significant overhead for copying the huge feature structures. We thus developed an efficient way to communicate and copy feature structures on shared-memory parallel machines.

The effectiveness of our parsing algorithm and PSTFS was shown through a series of experiments on parsing Japanese sentences from the EDR corpus. They demonstrated that our parallel HPSG parser has high efficiency in terms of both analysis time and speed-up and is efficient enough for practical use even when it is running on one processor. In addition, the overhead of copying in PSTFS is as small as the overhead on a sequential machine, so PSTFS achieves high efficiency in a parallel environment.

The time complexity of our parsing algorithm is theoretically $2(n-1)f(n)+D$ for sentence length n; empirically this time complexity can be approximated by $Cn + D$.

We are considering the use of our HPSG parser on PSTFS for a speech-recognition system, a natural language interface, and a speech machine translation system.

Acknowledgement

We are indebted to Dr. Kenjiro Taura for letting us use his parallel programming language ABCL/f and for his detailed discussion on the parsing algorithm. We thank Mr. Takaki Makino for letting us use his feature-structure manipulating system LiLFeS.

This research was partially funded by a JSPS project (JSPS-RFTF96P00502), and was also funded by JSPS Research Fellowships for Young Scientists.

Reference

Adriaens and Hahn (Eds.). (1994). *Parallel Natural Language Processing.* Ablex Publishing Corporation, Norwood, New Jersey.

Agha, G. (Ed.). (1986). *Actors: a model of concurrent computation in distributed systems.* The MIT Press, Cambridge, Massachusetts.

Carpenter, B. (1992). *The Logic of Typed Feature Structures.* Cambridge University Press, Cambridge, England.

Carpenter, B. and Penn, G. (1994). "ALE 2.0 User's Guide." Tech. rep., Carnegie Mellon University Laboratory for Computational Linguistics, Pittsburgh, PA.

Clark, K. and Gregory, S. (1986). "Parlog: Parallel programming in logic." *Journal of the ACM Transactions on Programming Languages and Systems*, *8*(1), 1–49.

Copestake, A., Flickinger, D., Malouf, R., Riehemann, S., and Sag, I. (1995). "Translation using Minimal Recursion Semantics." In *Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation (TMI-95)*.

Erbach, G. (1994). *ProFIT 1.05 user's guide*. Computerlinguistik, Universitaet des Saarlandes.

Flickinger, D. (2000). "On building a more efficient grammar by exploiting types." *Journal of Natural Language Engineering*, *6*(1), 15–28. to appear.

Grishman, R. and Chitrao, M. (1988). "Evaluation of a parallel chart parser." In *Proceedings of the Second Conference on Applied Natural Language Processing*, pp. 71–76. Association for Computational Linguistics.

Haas, A. (1987). "Parallel Parsing for Unification Grammars." In *IJCAI '87*, pp. 615–618.

Kanayama, H., Torisawa, K., Mitsuishi, Y., and Tsujii, J. (2000). "A Hybrid Japanese Parser with Hand-crafted Grammar and Statistics." In *COLING 2000*. to appear.

Kasami, T. (1965). "An efficient recognition and syntax algorithm for context-free languages." Tech. rep. AFCRL-65-758, Air Force Cambrige Research Lab., Bedford, Mass.

Kasper, W., Kiefer, B., Krieger, H.-U., Rupp, C., and Worm, K. (1999). "Charting the Depths of Robust Speech Parsing." In *37th Annual Meeting of the Association for Computational Linguistics (ACL'99)*.

Kiefer, B., Krieger, H.-U., Carroll, J., and Malouf, R. (1999). "A Bag of Useful Techniques for Efficient and Robust Parsing." In *37th Annual Meeting of the Association for Computational Linguistics (ACL'99)*.

Krieger, H.-U. and Schaefer, U. (1994). "$\mathcal{TDL}$ — A type description language for constraint-based grammars." In *COLING 94*, pp. 893–899.

Makino, T., Yoshida, M., Torisawa, K., and Tsujii, J. (1998). "LiLFeS — Towards a Practical HPSG Parser." In *COLING-ACL'98 Proceedings*.

Mitsuishi, Y., Torisawa, K., and Tsujii, J. (1998). "HPSG-style Underspecified Japanese Grammar with Wide Coverage." In *COLING-ACL'98 Proceedings*.

Miyao, Y., Makino, T., Torisawa, K., and Tsujii, J. (2000). "The LiLFeS Abstract Machine and Its Evaluation with the LinGO Grammar." *Journal of Natural Language Engineering*, *6*(1), 47–61. to appear.

Nijholt, A. (1994). *Parallel Natural Language Processing*, chap. Parallel Approaches to Context-Free Language Parsing, pp. 135–167. Ablex Publishing Corporation, Norwood, New Jersey.

Ninomiya, T., Torisawa, K., Taura, K., and Tsujii, J. (1997). "A Parallel CKY Parsing Algorithm on Large-Scale Distributed-Memory Parallel Machines." In *PACLING '97*, pp. 223–231.

Nishida, K., Torisawa, K., and Tsujii, J. (1999). "Efficient HPSG Parsing Algorithm with Array Unification." In *Proceedings 5th Natural Language Processing Pacific Rim Symposium 1999 (NLPRS'99)*.

Pollard, C. and Sag., I. A. (1994). *Head-Driven Phrase Structure Grammar.* University of Chicago Press and CSLI Publications, Chicago, IL.

Tateisi, Y., Torisawa, K., Miyao, Y., and Tsujii, J. (1998). "Translating the XTAG English Grammar to HPSG." In *4th Workshop on Tree-adjoining Grammars and Related Frameworks (TAG+)*, pp. 172–175.

Taura, K. (1997). *Efficient and Reusable Implementation of Fine-Grain Multithreading and Garbage Collection on Distributed-Memory Parallel Computers*. Ph.D. thesis, Department of Information Science, University of Tokyo.

Thompson, H. S. (1994). *Parallel Natural Language Processing*, chap. Parallel Parsers for Context-Free Grammars–Two Actual Implementations Compared, pp. 168–187. Ablex Publishing Corporation, Norwood, New Jersey.

Torisawa, K., Nishida, K., Miyao, Y., and Tsujii, J. (2000). "An HPSG Parser with CFG Filtering." *Journal of Natural Language Engineering*, *6*(1), –. to appear.

Ueda, K. (1985). "Guarded Horn Clauses." Tech. rep. TR-103, ICOT.

Uszkoreit, H., Backofen, R., Busemann, S., Diagne, A. K., Hinkelman, E. A., Kasper, W., Kiefer, B., Krieger, H.-U., Netter, K., Neumann, G., Oepen, S., and Spackman, S. P. (1994). "DISCO — an HPSG-based NLP system and its application for appointment scheduling." In *COLING 94*.

van Noord, G., Bouma, G., Koeling, R., and Nederhof, M.-J. (1999). "Robust Grammatical Analysis for Spoken Dialogue Systems." *Journal of Natural Language Engineering*, *5*(1), 45–93.

Yonezawa, A. and Ohsawa, I. (1988). "Object-Oriented Parallel Parsing for Context-Free Grammars." In *COLING 88*, pp. 773–778.

Younger, D. (1967). "Recognition and parsing of context-free languages in time n^3." *Information and Control*, *2*(10), 189–208.

Takashi Ninomiya: Takashi Ninomiya is a Ph.D course student of Graduate School of Science, University of Tokyo, and he is also a research fellow

of the Japan Society for the Promotion of Science. He received the B.S. degree in 1996 and the M.S. degree in 1998 from University of Tokyo. His current research interests are parallel parsing and robust natural language processing.

Kentaro Torisawa: Kentaro Torisawa is a research associate of Graduate School of Science, University of Tokyo. He received the B.S. degree and the M.S. from University of Tokyo in 1992 and 1994 respectively. He is also a researcher in the Information and Human Behavior program at PRESTO, Japan Science and Technology Corporation since 1998. He received the Ph.D from University of Tokyo in 2000. His current research interests are grammar formalisms, grammar learning and automatic knowledge acquisition.

Jun'ichi Tsujii: Jun'ichi Tsujii received the BE and ME degree in electrical engineering and Ph.D from Kyoto University (1971, 1973 and 1978, respectively). He stayed in CNRS, Grenoble, France as invited senior researcher from 1980 to 1981. He was assistant professor and associate professor of Department of Electrical Engineering, Kyoto University from 1973 to 1988 and professor of Computational Linguistics of Centre for Computational Linguistics of UMIST, Manchester, UK from 1988 to 1995. He is professor of Department of Information Science, the University of Tokyo, Japan from 1995. He received the IBM Science Award in 1988. He cuurently holds the presidency of the Association for Natural Language Processing while he is a member of International Committee of Computational Linguistics (ICCL).

(Received May 10, 2000)
(Revised July 31, 2000)
(Accepted October 10, 2000)