An Efficient Parallel Substrate for Typed Feature Structures on Shared Memory Parallel Machines

NINOMIYA Takashi[†], TORISAWA Kentaro[†] and TSUJII Jun'ichi[†‡]
[†]Department of Information Science
Graduate School of Science, University of Tokyo[*]
[‡]CCL, UMIST, U.K.

Abstract

This paper describes an efficient parallel system for processing Typed Feature Structures (TFSs) on shared-memory parallel machines. We call the system Parallel Substrate for TFS (PSTFS). PSTFS is designed for parallel computing environments where a large number of agents are working and communicating with each other. Such agents use PSTFS as their low-level module for solving constraints on TFSs and sending/receiving TFSs to/from other agents in an efficient manner. From a programmers' point of view, PSTFS provides a simple and unified mechanism for building high-level parallel NLP systems. The performance and the flexibility of our PSTFS are shown through the experiments on two different types of parallel HPSG parsers. The speed-up was more than 10 times on both parsers.

1 Introduction

The need for real-time NLP systems has been discussed for the last decade. The difficulty in implementing such a system is that people can not use sophisticated but computationally expensive methodologies. However, if we could provide an efficient tool/environment for developing parallel NLP systems, programmers would have to be less concerned about the issues related to efficiency of the system. This became possible due to recent developments of parallel machines with shared-memory architecture.

We propose an efficient programming environment for developing parallel NLP systems on shared-memory parallel machines, called the Parallel Substrate for Typed Feature Structures (PSTFS). The environment is based on agent-based/object-oriented architecture. In other words, a system based on PSTFS has many computational agents running on different processors in parallel; those agents communicate with each other by using messages including TFSs. Tasks of the whole system, such as parsing or semantic processing, are divided into several pieces which can be simultaneously computed by several agents.

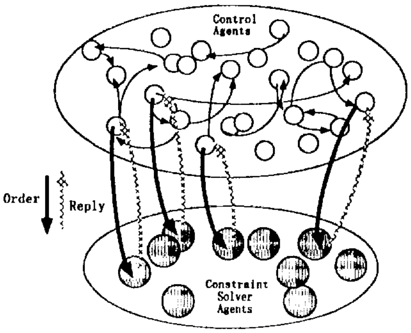


Figure 1: Agent-based System with the PSTFS

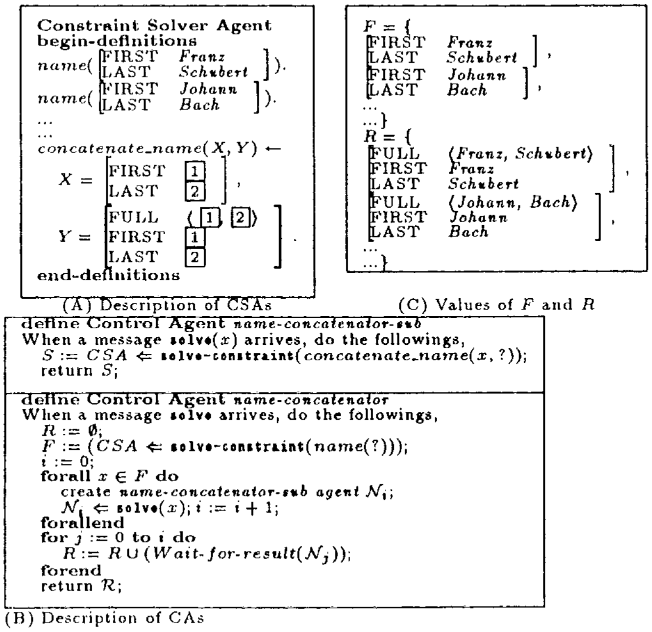
Several parallel NLP systems have been developed previously. But most of them have been neither efficient nor practical enough (Adriaens and Hahn, 1994). On the other hand, our PSTFS provides the following features.

- An efficient communication scheme for messages including Typed Feature Structures (TFSs) (Carpenter, 1992).
- Efficient treatment of TFSs by an abstract machine (Makino et al., 1998).

Another possible way to develop parallel NLP systems with TFSs is to use a full concurrent logic programming language (Clark and Gregory, 1986; Ueda, 1985). However, we have observed that it is necessary to control parallelism in a flexible way to achieve high-performance. (Fixed concurrency in a logic programming language does not provide sufficient flexibility.) Our agent-based architecture is suitable for accomplishing such flexibility in parallelism.

The next section discusses PSTFS from a programmers' point of view. Section 3 describes the PSTFS architecture in detail. Section 4 describes the performance of PSTFS on our HPSG parsers.

[*] This research is partially founded by the project of JSPS(JSPS-RFTF96P00502).



```
Constraint Solver Agent
begin-definitions
name( [FIRST   Franz   ] ).
      [LAST    Schubert]
name( [FIRST   Johann  ] ).
      [LAST    Bach    ]
...
...
concatenate_name(X, Y) ←
   X = [FIRST   [1]] ,
       [LAST    [2]]
       [FULL    ⟨[1], [2]⟩]
   Y = [FIRST   [1]       ] .
       [LAST    [2]       ]
end-definitions
```

(A) Description of CSAs

```
F = {
[FIRST   Franz   ] ,
[LAST    Schubert]
[FIRST   Johann  ] ,
[LAST    Bach    ]
...
...}
R = {
[FULL    ⟨Franz, Schubert⟩]
[FIRST   Franz            ] ,
[LAST    Schubert         ]
[FULL    ⟨Johann, Bach⟩   ]
[FIRST   Johann           ] ,
[LAST    Bach             ]
...
...}
```

(C) Values of F and R

```
define Control Agent name-concatenator-sub
When a message solve(x) arrives, do the followings,
  S := CSA ⇐ solve-constraint(concatenate_name(x, ?));
  return S;
```

```
define Control Agent name-concatenator
When a message solve arrives, do the followings,
  R := ∅;
  F := (CSA ⇐ solve-constraint(name(?)));
  i := 0;
  forall x ∈ F do
    create name-concatenator-sub agent N_i;
    N_i ⇐ solve(x); i := i + 1;
  forallend
  for j := 0 to i do
    R := R ∪ (Wait-for-result(N_j));
  forend
  return R;
```

(B) Description of CAs

Figure 2: Example *concatenate_name*

2 Programmers' View

From a programmers' point of view, the PSTFS mechanism is quite simple and natural, which is due to careful design for accomplishing high-performance and ease of programming.

Systems to be constructed on our PSTFS will include two different types of agents:

- Control Agents (CAs)
- Constraint Solver Agents (CSAs)

As illustrated in Figure 1, CAs have overall control of a system, including control of parallelism, and they behave as masters of CSAs. CSAs modify TFSs according to the orders from CAs. Note that CAs can neither modify nor generate TFSs by themselves.

PSTFS has been implemented by combining two existing programming languages: the concurrent object-oriented programming language ABCL/*f* (Taura, 1997) and the sequential programming language LiLFeS (Makino et al., 1998). CAs can be written in ABCL/*f*, while description of CSAs can be mainly written in LiLFeS.

Figure 2 shows an example of a part of the PSTFS code. The task of this code is to concatenate the first and the second name in a given list. One of the CAs is called *name-concatenator*. This specific CA gathers pairs of the first and last name by asking a CSA with the message `solve-constraint`(*'name*(?)*'*). When the CSA receives this message, the argument *'name*(?)*'* is treated as a Prolog query in LiLFeS[1], according to the program of a CSA ((A) of Figure 2). There are several facts with the predicate *'name'*. When the goal *'name*(?)*'* is processed by a CSA, all the possible answers defined by these facts are returned. The obtained pairs are stored in the variable F in the *name-concatenator* ((C) in Figure 2).

The next behavior of the *name-concatenator* agent is to create CAs (*name-concatenator-subs*) and to send the message **solve** with a TFS to each created CA running in parallel. The message contains one of the TFSs in F. Each *name-concatenator-sub* asks a CSA to concatenate FIRST and LAST in a TFS. Then each CSA concatenates them using the definite clause *concatenate_name* given in (A) of Figure 2. The result is returned to the *name-concatenator-sub* which had asked to do the job. Note that the *name-concatenator-sub* can ask any of the existing CSAs. All CSAs can basically perform concatenation in parallel and independent way. Then, the *name-concatenator* waits for the *name-concatenator-sub* to return concatenated names, and puts the return values into the variable R.

The CA *name-concatenator* controls the overall process. It controls parallelism by creating CAs and sending messages to them. On the other hand, all the operations on TFSs are performed by CSAs when they are asked by CAs.

Suppose that one is trying to implement a parsing system based on PSTFS. The distinction between CAs and CSAs roughly corresponds to the distinction between an abstract parsing schema and application of phrase structure rules. Here, a parsing schema means 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. This distinction is a minor factor in writing a sequential parser, but it has a major impact on a parallel environment.

For instance, suppose that several distinct agents evoke applications of phrase structure rules against the same data simultaneously, and the applications are accompanied with destructive operations on the data. This can cause an anomaly, since the agents will modify the original data in unpredictable order and there is no way to keep consistency. In order to avoid this anomaly, one has to determine what is an atomic operation and provide a method to prevent the anomaly when atomic operations are evoked by several agents. In our framework, any action taken by CSAs is viewed as such an atomic operation and it is guaranteed that no anomaly occurs even if CSAs concurrently

[1] LiLFeS supports definite clause programs, a TFS version of Horn clauses.

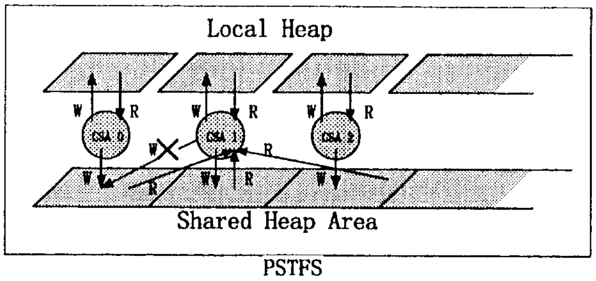


Figure 3: Inside of the PSTFS

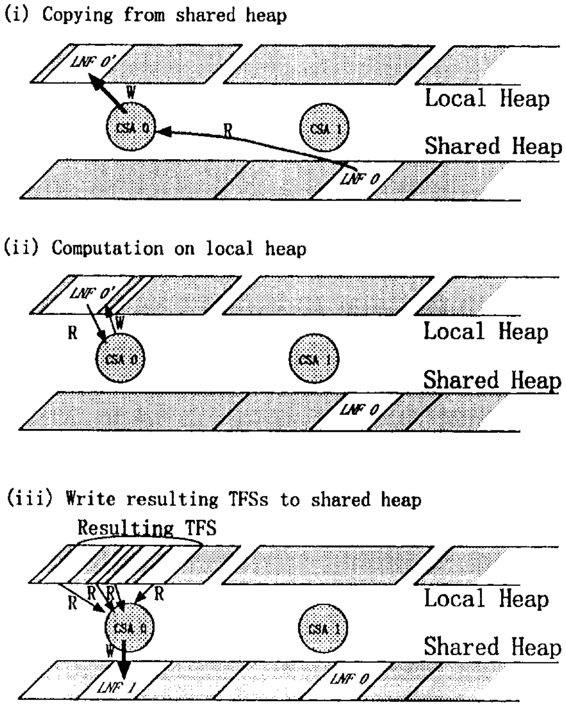
perform operations on the same data. This can be done by introducing copying of TFSs, which does not require any destructive operations. The details are described in the next section.

The other implication of the distinction between CAs and CSAs is that this enables efficient communication between agents in a natural way. During parsing in HPSG, it is possible that TFSs with hundreds of nodes can be generated. Encoding such TFSs in a message and sending them in an efficient way are not trivial. PSTFS provides a communication scheme that enables efficient sending/receiving of such TFSs. This becomes possible because of the distinction of agents. In other words, since CAs cannot modify a TFS, CAs do not have to have a real image of TFSs. When CSAs return the results of computations to CAs, the CSAs send only an ID of a TFS. Only when the ID is passed to other CSAs and they try to modify a TFS with the ID, the actual transfer of the TFS's real image occurs. Since the transfer is carried out only between CSAs, it can be directly performed using a low level representation of TFSs used in CSAs in an efficient manner. Note that if CAs were to modify TFSs directly, this scheme could not have been used.

3 Architecture

This section explains the inner structure of PSTFS focusing on the execution mechanism of CSAs (See (Taura, 1997) for further detail on CAs). A CSA is implemented by modifying the abstract machine for TFSs (i.e., LiAM), originally designed for executing LiLFeS (Makino et al., 1998).

The important constraint in designing the execution mechanism for CSAs is that TFSs generated by CSAs must be kept unmodified. This is because the TFSs must be used with several agents in parallel. If the TFS had been modified by a CSA and if other agents did not know the fact, the expected results could not have been obtained. Note that unification, which is a major operation on TFSs, is a destructive operation, and modifications are likely to occur while executing CSAs. Our execution mechanism handles this problem by letting CSAs copy TFSs generated by other CSAs at each time. Though this may not look like an efficient way at first glance, it has been performed efficiently by shared memory mechanisms and our copying methods.


(i) Copying from shared heap

(ii) Computation on local heap

(iii) Write resulting TFSs to shared heap

Figure 4: Operation steps on PSTFS

A CSA uses two different types of memory areas as its heap:

- shared heap
- local heap

A local heap is used for temporary operations during the computation inside a CSA. A CSA cannot read/write local heap of other CSAs. A shared heap is used as a medium of communication between CSAs, and it is realized on a shared memory. When a CSA completes a computation on TFSs, it writes the result on a shared heap. Since the shared heap can be read by any CSAs, each CSA can read the result performed by any other CSAs. However, the portion of a shared heap that the CSA can write to is limited. Any other CSA cannot write on that portion.

Next, we look at the steps performed by a CSA when it is asked by CAs with a message.

Note that the message only contains the IDs of the TFSs as described in the previous section. The IDs are realized as pointers on the shared heap.

1. Copy TFSs pointed at by the IDs in the message from the shared heap to the local heap of the CSA. ((i) in Figure 4.)
2. Process a query using LiAM and the local heap. ((ii) in Figure 4.)
3. If a query has an answer, the result is copied to the portion of the shared heap writable by the CSA. Keep IDs on the copied TFSs. If there is no answer for the query, go to Step 5. ((iii) in Figure 4.)
4. Evoke backtracking in LiAM and go to Step 2.
5. Send the message, including the kept IDs, back to the CA that had asked the task.

Note that, in step 3, the results of the computation becomes readable by other CSAs. This procedure has the following desirable features.

Simultaneous Copying An identical 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[2].

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 the previous section, the transfer of real images of TFSs is performed only after the IDs of the TFSs reach to the CSAs requiring the TFSs. Redundant copying/sending of the TFSs' real image is reduced, and the transfer is performed efficiently by mechanisms originally provided by LiAM.

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

Note that copying in the procedures makes it possible to support non-determinism in NLP systems. For instance, during parsing, intermediate parse trees must be kept. In a chart parsing for a unification-based grammar, generated edges are kept untouched, and destructive operations on the results must be done after copying them. The copying of TFSs in the above steps realizes such mechanisms in a natural way, as it is designed for efficient support for data sharing and destructive operations on shared heaps by parallel agents.

4 Application and Performance Evaluation

This section describes two different types of HPSG parsers implemented on PSTFS. One is designed for our Japanese grammar and the algorithm is a parallel version of the CKY algorithm (Kasami, 1965). The other is a parser for an ALE-style Grammar (Carpenter and Penn, 1994). The algorithms of both parsers are based on parallel parsing algorithms for CFG (Ninomiya et al., 1997; Nijholt, 1994; Grishman and Chitrao, 1988; Thompson, 1994). Descriptions of both parsers are concise. Both of them are written in less than 1,000 lines. This shows that our PSTFS can be easily used. With the high performance of the parsers, this shows the feasibility and flexibility of our PSTFS.

For simplicity of discussion, we assume that HPSG consists of lexical entries and rule schemata. Lexical entries can be regarded as TFSs assigned to each word. A rule schema is a rule in the form of $z \rightarrow abc \cdots$ where z, a, b, c are TFSs.

4.1 Parallel CKY-style HPSG Parsing Algorithm

A sequential CKY parser for CFG uses a data structure called a *triangular table*. Let $F_{i,j}$ denote a cell in the triangular table. Each cell $F_{i,j}$ has a set of the non-terminal symbols in CFG that can generate the word sequence from the $i+1$-th word to the j-th word in an input sentence. The sequential CKY algorithm computes each $F_{i,j}$ according to a certain order.

Our algorithm for a parallel CKY-style parser for HPSG computes each $F_{i,j}$ in parallel. Note that $F_{i,j}$ contains TFSs covering the word sequence from the $i+1$-th word to the j-th word, not non-terminals. We consider only the rule schemata with a form of $z \rightarrow ab$ where z, a, b are TFSs. Parsing is started by a CA called $\mathcal{PARSER}$. $\mathcal{PARSER}$ creates *cell-agents* $\mathcal{C}_{i,j}(0 \leq i < j \leq n)$ and distributes them to processors on a parallel machine (Figure 5). Each $\mathcal{C}_{i,j}$ computes $F_{i,j}$ in parallel. More precisely, $\mathcal{C}_{i,j}(j-i=1)$ looks up a dictionary and obtains lexical entries. $\mathcal{C}_{i,j}(j-i>1)$ waits for the messages including $F_{i,k}$ and $F_{k,j}$ for all $k(i<k<j)$ from other *cell-agents*. When $\mathcal{C}_{i,j}$ receives $F_{i,k}$ and $F_{k,j}$ for an arbitrary k, $\mathcal{C}_{i,j}$ computes TFSs by applying rule schemata to each members of

[2] Actually, this is not trivial. Copying in Step 3 normalizes TFSs and stores the TFSs into a continuous region on a shared heap. TFSs stored in such a way can be copied without any side-effect.

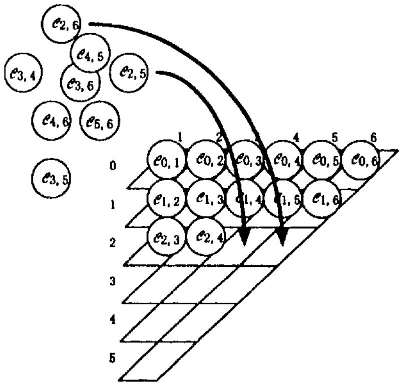


Figure 5: Correspondence between CKY matrix and agents: $C_{i,j}$ correspond to the element of a CKY triangular matrix

$F_{i,k}$ and $F_{k,j}$. The computed TFSs are considered to be mothers of members of $F_{i,k}$ and $F_{k,j}$ and they are added to $F_{i,j}$. Note that these applications of rule schemata are done in parallel in several CSAs[3]. Finally, when computation of $F_{i,j}$ (using $F_{i,k}$ and $F_{k,j}$ for all $k(i < k < j)$) is completed, $C_{i,j}$ distributes $F_{i,j}$ to other agents waiting for $F_{i,j}$. Parsing is completed when the computation of $F_{0,n}$ is completed.

We have done a series of experiments on a shared-memory parallel machine, SUN Ultra Enterprise 10000 consisting of 64 nodes (each node is a 250 MHz UltraSparc) and 6 GByte shared memory. The corpus consists of 879 random sentences from the EDR Japanese corpus written in Japanese (average length of sentences is 20.8)[4]. The grammar we used is an underspecified Japanese HPSG grammar (Mitsuishi et al., 1998) consisting of 6 ID-schemata and 39 lexical entries (assigned to functional words) and 41 lexical-entry-templates (assigned to parts of speech). This grammar has wide coverage and high accuracy for real-world texts[5].

Table 1 shows the result and comparison with a parser written in LiLFeS. Figure 6 shows its speed-up. From the Figure 6, we observe that the maximum speedup reaches up to 12.4 times. The average parsing time is 85 msec per sentence[6].

Number of Processors	Avg. of Parsing Time(msec) PSTFS	LiLFeS
1	1057	991
10	248	
20	138	
30	106	
40	93	
50	85	
60	135	

Table 1: Average parsing time per sentence

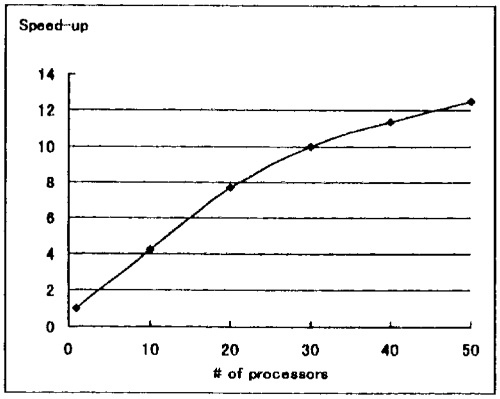


Figure 6: Speed-up of parsing time on parallel CKY-style HPSG parser

4.2 Chart-based Parallel HPSG Parsing Algorithm for ALE Grammar

Next, we developed a parallel chart-based HPSG parser for an ALE-style grammar. The algorithm is based on a chart schema on which each agent throws active edges and inactive edges containing a TFS. When we regard the rule schemata as a set of rewriting rules in CFG, this algorithm is exactly the same as the Thompson's algorithm (Thompson, 1994) and similar to PAX (Matsumoto, 1987). The main difference between the chart-based parser and our CKY-style parser is that the ALE-style parser supports a n-branching tree.

A parsing process is started by a CA called $\mathcal{PARSER}$. It .creates *word-position agents* $\mathcal{P}_k(0 \leq k \leq n)$, distributes them to parallel processors and waits for them to complete their tasks. The role of the *word-position agent* $\mathcal{P}_k$

[3] CSAs cannot be added dynamically in our implementation. So, to gain the maximum parallelism, we assigned a CSA to each processor. Each $C_{i,j}$ asks the CSA on the same processor to apply rule schemata.

[4] We chose 1000 random sentences from the EDR Japanese corpus, and the used 897 sentences are all the parsable sentences by the grammar.

[5] This grammar can generate parse trees for 82% of 10000 sentences from the EDR Japanese corpus and the dependency accuracy is 78%.

[6] Using 60 processors is worse than with 50 processors. In general, when the number of processes increases to near or more than the number of existing processors, context switch between processes occurs frequently on shared-memory parallel machines (many people can use the machines simultaneously). We believe the cause for the inefficiency when using 60 processors lies in such context switches.

	Short Length Sentences
(1)	kim believes sandy to walk
(2)	a person whom he sees walks
(3)	he is seen
(4)	he persuades her to walk

	Long Length Sentences
(1)	a person who sees kim who sees sandy whom he tries to see walks
(2)	a person who sees kim who sees sandy who sees kim whom he tries to see walks
(3)	a person who sees kim who sees sandy who sees kim who believes her to tend to walk walks

Table 2: Test corpus for parallel ALE-style HPSG parser

is to collect edges adjacent to the position k. A *word-position agent* has its own active edges and inactive edges. An active edge is in the form $\langle i, z \rightarrow A \circ xB \rangle$, where A is a set of TFSs which have already been unified with an existing constituents, B is a set of TFSs which have not been unified yet, and x is the TFS which can be unified with the constituent in an inactive edge whose left-side is in position k. Inactive edges are in the form $\langle k, x, j \rangle$, where k is the left-side position of the constituent x and j is the right-side position of the constituent x. That is, the set of all inactive edges whose left-side position is k are collected by $\mathcal{P}_k$.

In our algorithm, $\mathcal{P}_k$ is always waiting for either an active edge or an inactive edge, and performs the following procedure when receiving an edge.

- When $\mathcal{P}_k$ receives an active edge $\langle i, z \rightarrow A \circ xB \rangle$, $\mathcal{P}_k$ preserve the edge and tries to find the unifiable constituent with x from the set of inactive edges that $\mathcal{P}_k$ has already received. If the unification succeeds, a new active edge $\langle i, z \rightarrow Ax \circ B \rangle$ is created. If the dot in the new active edge reaches to the end of RHS (i.e. $B = \emptyset$), a new inactive edge is created and is sent to $\mathcal{P}_i$. Otherwise the new active edge is sent to $\mathcal{P}_j$.
- When $\mathcal{P}_k$ receives an inactive edge $\langle k, x, j \rangle$, $\mathcal{P}_k$ preserves the edge and tries to find the unifiable constituent on the right side of the dot from the set of active edges that $\mathcal{P}_k$ has already received. If the unification succeeds, a new active edge $\langle i, z \rightarrow Ax \circ B \rangle$ is created. If the dot in the new active edge reaches to the end of RHS (i.e. $B = \emptyset$), a new inactive edge is created and is sent to $\mathcal{P}_i$. Otherwise the new active edge is sent to $\mathcal{P}_j$.

As long as *word-position-agents* follow these behavior, they can run in parallel without any other restriction.

We have done a series of experiments in the same machine settings as the experiments with the CKY-style HPSG parser. We measured both its speed up and real parsing time, and we compared our parallel parser with the ALE system and a sequential parser on LiLFeS. The grammar we used is a sample HPSG grammar attached to ALE system[7], which has 7 schemata and 62 lexical entries. The test corpus we used in this experiment is shown in the Table 2. Results and comparison with other sequential parsing systems are given in Table 3. Its speedup is shown in Figure 7. From the figure, we observe that the maximum speedup reaches up to 10.9 times and its parsing time is 1776 msec per sentence.

Short Length Sentences			
Number of Processors	Avg. of Parsing Time(msec)		
	PSTFS	LiLFeS	ALE
1	625	125	1590
10	160		
20	156		
30	127		
40	205		
50	142		
60	170		

Long Length Sentences			
Number of Processors	Avg. of Parsing Time(msec)		
	PSTFS	LiLFeS	ALE
1	19307	30867	389370
10	3208		
20	2139		
30	1776		
40	1841		
50	1902		
60	2052		

Table 3: Average parsing time per sentence

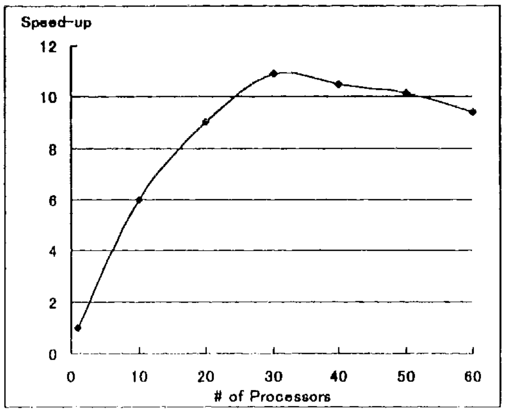


Figure 7: Speed-up of parsing time on chart-based parallel HPSG parser

4.3 Discussion

In both parsers, parsing time reaches a 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

[7] This sample grammar is converted to LiLFeS style half automatically.

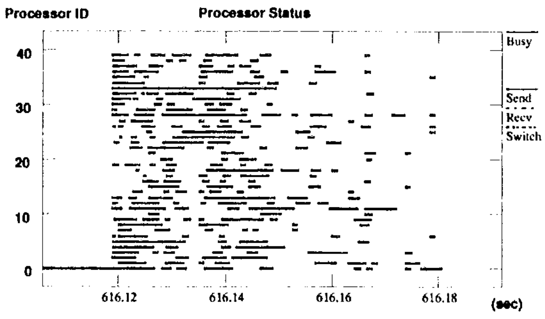


Figure 8: Processors status

framework for the goal to provide a parallel programming environment for real-time NLP. In addition, our parallel HPSG parsers are considerably more efficient than other sequential HPSG parsers.

However, the speed-up is not proportional to the number of processors. We think that this is because the parallelism extracted in our parsing algorithm is not enough. Figure 8 shows the log of parsing Japanese sentences by the CKY-style parser. The black lines indicate when a processor is busy. One can see that many processors are frequently idle.

We think that this idle time does not suggest that parallel NLP systems are useless. On the contrary, this suggest that parallel NLP systems have many possibilities. If we introduce semantic processing for instance, overall processing time may not change because the idle time is used for semantic processing. Another possibility is the use of parallel NLP systems as a server. Even if we feed several sentences at a time, throughput will not change, because the idle time is used for parsing different sentences.

5 Conclusion and Future Work

We described PSTFS, a substrate for parallel processing of typed feature structures. PSTFS serves as an efficient programming environment for implementing parallel NLP systems. We have shown the feasibility and flexibility of our PSTFS through the implementation of two HPSG parsers.

For the future, we are considering the use of our HPSG parser on PSTFS for a speech recognition system, a Natural Language Interface or Speech Machine Translation applications.

References

Adriaens and Hahn, editors. 1994. *Parallel Natural Language Processing.* Ablex Publishing Corporation, New Jersey.

Bob Carpenter and Gerald Penn. 1994. ALE 2.0 user's guide. Technical report, Carnegie Mellon University Laboratory for Computational Linguistics, Pittsburgh, PA.

Bob Carpenter. 1992. *The Logic of Typed Feature Structures.* Cambridge University Press, Cambridge, England.

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

Ralph Grishman and Mehesh Chitrao. 1988. Evaluation of a parallel chart parser. In *Proceedings of the second Conference on Applied Natural Language Processing*, pages 71–76. Association for Computational Linguistics.

T. Kasami. 1965. An efficient recognition and syntax algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambrige Research Lab., Bedford, Mass.

Takaki Makino, Minoru Yoshida, Kentaro Torisawa, and Jun'ichi Tsujii. 1998. LiLFeS — towards a practical HPSG parser. In *COLING-ACL'98 Proceedings*, August.

Yuji Matsumoto. 1987. A parallel parsing system for natural language analysis. In *Proceedings of 3rd International Conference on Logic Programming*, pages 396–409.

Yutaka Mitsuishi, Kentaro Torisawa, and Jun'ichi Tsujii. 1998. HPSG-style underspecified Japanese grammar with wide coverage. In *COLING-ACL'98 Proceedings*, August.

Anton Nijholt, 1994. *Parallel Natural Language Processing*, chapter Parallel Approaches to Context-Free Language Parsing, pages 135–167. Ablex Publishing Corporation.

Takashi Ninomiya, Kentaro Torisawa, Kenjiro Taura, and Jun'ichi Tsujii. 1997. A parallel cky parsing algorithm on large-scale distributed-memory parallel machines. In *PACLING '97*, pages 223–231, September.

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

Henry S. Thompson, 1994. *Parallel Natural Language Processing*, chapter Parallel Parsers for Context-Free Grammars–Two Actual Implementations Comparesd, pages 168–187. Ablex Publishing Corporation.

Kazunori Ueda. 1985. Guarded horn clauses. Technical Report TR-103, ICOT.