自 然 言 語 処 理 85－6
（1 9 9 1. 9. 1 9）

並列効果の高い単一化解析法

Peter Neuhaus　　古瀬 蔵　　飯田 仁

ＡＴＲ自動翻訳電話研究所

効果的にプログラム可能な並列アーキテクチャが利用できるようになるにつれ、自然言語処理においても並列計算機が使われるようになってきた。しかし、超並列アルゴリズムなどで仮定される理論上のプロセッサの数と、実際に並列計算機で使用可能なプロセッサの数の違いの問題は議論されないことが多かった。本論文では、ＪＰＳＧのようなユニフィケーションに基づく文法を使って、ＣＹＫの並列化アルゴリズムを改良した効率的なパーサを提案し、与えられた並列計算機の並列度に関し、効果的に動作することを示す。

Unification-Based Parsing on Increasing Levels of Parallelism

P. Neuhaus　　O. Furuse　　H. Iida

ATR Interpreting Telephony Research Laboratories
Sampeidani Inuidani, Seika-cho, Soraku-gun, Kyoto 619-02, Japan

As effectively programmable parallel architectures become available their usage in natural language processing increases. But an often disregarded problem is the discrepancy between the number of processors required by so-called *massively-parallel* algorithms and the number of processors provided by the parallel machine actually at hand.

A parallel parsing algorithm on the basis of the well known CYK algorithm has been published. We present an efficient, further parallelized version for JPSG-like unification-based grammars and show the effectiveness of restricting parallelization with regard to the size of the parallel machine used.

1 Introduction

As effectively programmable parallel architectures become available their usage in natural language processing increases. Algorithms for parallel analysis, i.e. parsing and unification, have been published. For example, [Mat89] presents a parallel parser based on logic programming, [Lan90] shows the utilization of systolic computations for parallel parsing, [Fuj90] investigates parallel unification, to name only a few.

Some of the suggested algorithms are *massively* parallel (for instance [Lan90]). But an often disregarded problem is the discrepancy between the number of processors required by these algorithms and the number of processors provided by the parallel machine actually at hand. Usually this problem is handled by virtual processors and/or a scheduling scheme. However, this obviously introduces additional overhead, that in the worst case, negates the benefits of parallelization. So-called massively-parallel parsers are an interesting research subject, but most available parallel architectures will not allow for *massive* parallelism.

In this paper a parallel parsing algorithm for a JPSG-style unification-based grammar is presented. It is based on the CYK (Cocke-Younger-Kasami, for example [You67]) parsing[1] algorithm. The algorithm is parallelized on increasing levels in terms of the number of required processors. At the first level, as proposed by [Bar90], a linear number of processors (in the length of the input string) is used. We show that on further levels more and more parallelism can be achieved and that this is necessary for an efficient parser for natural language processing.

In section 3 the original CYK parsing algorithm and its first level parallelization are explained. After a brief discussion of possible parallelizations in section 4, section 5 presents our further parallelized versions that will fit *medium*-sized parallel machines. In section 6 implementational details are explained. Finally, section 7 gives a discussion of results that show how the choice of an appropriate parallelization scheme is crucial for efficient run-time results.

2 Unification-based Grammar

A unification-based grammar has two components: a (usually[2]) context-free grammar backbone and some feature structure formalism attached to it. For a basic introduction to unification-based grammars the reader is referred to [Shi86]. We use a grammar following the notion of JPSG as presented in [Gun87].

JPSG uses just one context free rule, *Mother* $\longrightarrow$ *Daughter Head*, to account for the structure of Japanese sentences. This assumption of binary structures in JPSG suggests the use of a CYK type of parser, because it requires the context-free grammar to be in Chomsky Normal Form[3], i.e. it must be binary (cf. section 3).

We do not restrict the grammar used to contain only one context free rule as was suggested in [Gun87]. Actually the grammar should try to restrict the context-free syntax as far as possible because checking context free rules is less expensive than doing unifications. [Tom91] reports that in certain parsers over 90% of the time is consumed by unification.

There are two main strategies combining a parser and a unification algorithm. One is to produce all parse trees of the input string first, and then to do all unifications in a second phase. The other way is to do the unification after each reduction step of the parser. Since many syntactically possible structures are semantically ill-formed, the latter strategy will rule out these structures at an early point in time. This kind of parser is called an *integrated parser* and will be our choice.

3 Parallel Parsing

Before introducing the first level of parallelization we will explain the original sequential CYK algorithm as presented in [You67].

[1]A remark concerning the term "parser": we will use it both in the sense of parsing a context free grammar and in the sense of analyzing natural language which itself consists of parsing and unification. Its use should be clear from the context.

[2][Mat89] covers non-context-free grammars, too.

[3]Actually it is easily possible to allow unary rules which may sometimes be convenient.

3.1 The CYK Parser for Context Free Languages

If a grammar contains only productions of the form $A \rightarrow B\ C$ or $A \rightarrow t$ for some *non-terminals A, B and C* and some *terminal t*, it is said to be in *Chomsky normal form.* The standard CYK algorithm determines for all sub-strings of the input string their possible (sub-)tree structures. This is done by building a ***table***, here M. The first row contains the pre-terminals (the A of "$A \rightarrow t$") as ***table entries*** and is indexed $M_{i,i}$, where $i = 0, \ldots, n-1$ and n is the length of the input string.

The next row's entries – indexed as $M_{i,i+1}$, for $i = 0, \ldots, n-2$ – are computed by ***combining*** two adjacent entries of the previous line, i.e. $M_{i,i}$ and $M_{i+1,i+1}$. Combining two entries means, that the non-terminals in the two sets are *joined* to pairs and for each pair the grammar is checked (by a table lookup) if it can be reduced by a production. If this is the case this production's left side is added to the set $M_{i,i+1}$. Thus this new entry describes two tokens of the input string. For example, in figure 1 the verb "ita" (past tense of "to be") and the noun "toki" ("time") form a noun phrase.

For the next rows, each entry describes substrings of the input string. For the computation of entry $M_{i,j}$ (describing all possible structures of the substring from tokens i to j of the input string), for all $k = i \ldots (j-1)$, entries $M_{i,k}$ and $M_{k+1,j}$ are combined. For example, figure 1 shows the computation of $M_{2,4}$ at the third row. Entries $M_{2,2}$ and $M_{3,4}$ are combined by joining **V** and **PP**, for which no grammar rule exists. Entries $M_{2,3}$ and $M_{4,4}$ are combined by joining **NP** and **P**, yielding **PP**. For entries in the next row three combinations will take place, and so on.

Eventually table entry $M_{0,n-1}$ will contain the first non-terminals of all possible parses. If it contains the root symbol, obviously the string has been accepted. Actually at this point no tree is directly available. How to get the parse tree (in our implementation) will be described in section 6.

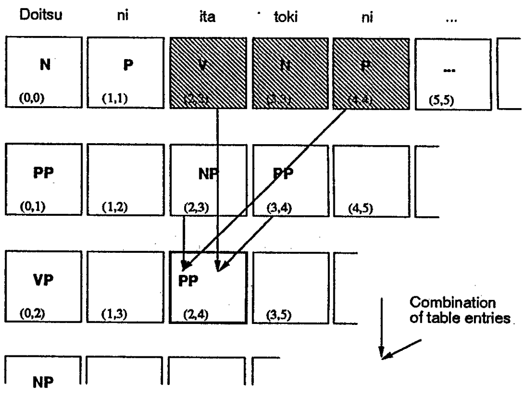

Figure 1: The combination scheme for entries in table M, here entry $M_{2,4}$

Since each table entry can contain at most all non-terminals of the grammar the combination of two table entries is of constant complexity in the length of the input string. That, the overall complexity is $O(n^3)$.

3.2 A Parallel CYK Parser

As shown by [Bar90] the above described algorithm's time complexity can be decreased to $O(n^2)$ by utilization of a number of processes linear in the number of input tokens, the time-processor product remaining the same.

As can be seen from the description of the sequential CYK algorithm the computation of a row entry depends only on previous rows. For example in figure 1, while $M_{2,4}$ is computed by one process, other processes can compute entries $M_{0,2}$, $M_{1,3}$, $M_{3,5}$, and so on. That means that all entries of a row can be computed in parallel. We need only ensure that the computation does not start before the required entries of previous rows are computed.

Let P_i for $i = 0 \ldots n-1$ be the processes then P_i computes entries $M_{i,i} \ldots M_{i,n-1}$. *Synchronize*(P_i, P_{i+1}) makes sure that process P_i does not proceed to compute entry $M_{i,j}$ until process P_{i+1} finished its computation of entry $M_{i+1,j}$. We get the following algorithm:

```
begin P{i}
  for j=i to (n-1)
  do
    synchronize(P{i},P{i+1})
    compute-entry(M{i,j})
end P
```

This algorithm has been proposed in the field of programming language parsing. In [Bar90] it is argued that – with respect to existing parallel architectures – a parallel parsing algorithm that's processor usage is of linear complexity (in the length of the input string) is much more realistic than an algorithm with quadratic complexity is. Though input strings in natural language analysis are much shorter, a similar argument applies to natural language parsing as well.

3.3 Theoretical Complexity vs. Actual Usage of Processors

Because of the much shorter strings in natural language analysis the complexity in terms of the O-calculus is not so important. The O-calculus is only valid for asymtotic considerations. For us the ratio of the number of processes to the number of (hardware) processors is much more important. If it becomes extremely big or small then the parsing is far from being efficient.

The parallel CYK algorithm presented above starts out with n processes doing one combination of entries. With every new row one process terminates but each remaining process has to combine one more pair of entries, thus the quadratic time complexity. What should concern us is that once there are less processes than actual processors we waste computational power. The effect of this waste can be seen in the results presented below.

4 Possible Parallelization of Unification-based Parsers

The basic analysis of the parallelization of unification-based parsers (cf. [Kat90]) shows that there are three sources of parallelism exploitable by a parsing algorithm:

1. context-free grammar
   (a) independent sub-trees
   (b) structural ambiguities
2. disjunctions of feature structures
3. recursive unification of complex feature structures

A parse tree usually contains ***non-overlapping subtrees*** that can be computed in parallel. Moreover, if there is a ***structural ambiguity*** in a sentence, two or more parse trees will be created. The related unifications can be done simultaneously.

Disjunction of feature structures means that one terminal or non-terminal has more than one feature structure. For example, a verb may have two different feature structures, for example because of a transitive and an intransitive meaning. Thus there are two possibilities to form a verb-phrase by adding an inflection. The corresponding unifications can be done in parallel.

Two *complex feature structures* are said to unify if the values (possibly again complex structures) of corresponding attributes unify. Complex feature structures therefor imply recursion. Its parallelization would lead to massive parallelism[4]. We did not exploit this source of parallelism (item 3 above) because it counteracts achieving a parsing algorithm efficient for *medium*-sized parallel architectures. For a discussion of this kind of massive parallelization and its results the reader is referred to [Fuj90].

5 Our Further Parallelized Parsing Algorithms

We are presenting one parsing algorithm on several levels of parallelization. Thus the following subsections show algorithmic realizations of the discussion above.

5.1 Combining Table Entries is Independent

Let us examine the example of computing a row entry from above. To compute entry $M_{2,4}$ it is necessary to combine entries $M_{2,2}$ with $M_{3,4}$ and $M_{2,3}$ with $M_{4,4}$. These two combinations can be done in parallel, let us say by a *worker*-process.

This will lead to the usage of $O(n^2)$ processors[5]. The synchronization overhead increases because now all *worker*-processes have to be synchronized because an entry is not entirely computed until all related "workers" are finished. To prevent them from overwriting each other's results, a *lock* must be used also (*with-lock* acquires the lock, executes the body and releases the lock). Thus we get the following algorithm:

```
begin P{i,k}  \\ worker for k-th combination
  for j=i to (n-1)
  do
     for l=i to (j-1)
     do
        synchronize(P{i,k},P{i+1,l})
     combine(M{i,k},M{(k+1),j})
     with-lock
       write(M{i,j})
end P
```

This approach is somewhat naive because in an actual parse there are often empty entries. A *worker* combining empty entries just consumes time for synchronization. After looking at another source of parallelism in the next section we will consider further the combination of table entries.

5.2 Combining Non-Terminals is Independent

For the combination of two entries each non-terminal in the first entry is combined with each non-terminal in the second entry. While the check of the grammar table is very fast and can be done sequentially the unification of two items is slow. So the unifications should be done in parallel after the sequential syntax check has sorted out all impossible combinations with respect to the context free grammar. The synchronization mechanism used here is that of a *fork*, i.e. the entry-combination routine *spawns* unification processes and waits for them to be terminated.

This further parallelization covers item 2 of section 4. If a non-terminal has disjunctive feature structures there are multiple copies of it in the table entry, each with a different feature structure. Parallelizing the combination of all non-terminals of two table entries obviously leads to a parallel unification of the disjunction of feature structures. The resulting algorithm is shown in the next section after one more level of parallelization has been added.

[4]To avoid massive parallelism here a *worker/agenda* mechanism could be helpful. Though the BEHOLDER package will supply such a mechanism it has not been implementation yet.

[5]More exactly the number of processors will be $\lfloor n/2 \rfloor * \lceil n/2 \rceil$

5.3 Reconsider Combination of Entries

The fork mechanism described in section 5.2 can be expanded to the entire computation of an entry. That is not only the joining of non-terminals but also the combination of table entries is executed in the *fork*. Thus it covers not only all unifications of one combination of a pair of entries but all necessary combinations of previous entries. That yields the following algorithm:

```
begin P{i}
  for j=i to (n-1)
  do
     synchronize(P{i},P{i+1})
     compute-entry(M{i,j})
end P

begin compute-entry(M{i,j})
  for k=i to (j-1)
  fork
       combine(M{i,k},M{(k+1),j})
end compute-entry
```

```
begin combine(M{i,k},M{l,j})
  for all pairs in M{i,k} X M{l,j}
  do
     if reducible
     then mark pair
  for all marked pairs
  fork
       unify
end combine
```

6 Implementation

We implemented several parallel versions of a parser along the above described levels of parallelization on a Sequent Symmetry S81 with 12 tightly-coupled processors. The coding was done in CLiP (cf. [CLiP]), a parallel LISP based on Allegro CL. To deal with specific problems of CLiP the BEHOLDER package proved to be very useful.

Under CLiP only 11 processors are available and another processor has been spent for monitoring, such that we could test the different parsers with 1 to 10 processors. On these, so called, *Light Weight Processes* were scheduled. A sequential reference version also has been implemented.

In our algorithm on a tightly-coupled architecture the parsing table M is held in common memory. In contrast to the original algorithm presented in section 3.1, we store pointers to the constituents of each non-terminal. Thus, an entry can contain multiple copies of one non-terminal – each with different constituents[6]

The stated synchronization condition is implemented by blocking and unblocking processes. Before a process computes an entry it reads a channel (here a mail-box) from its right neighbour. If it is empty (indicating that required data is still being computed by its right neighbour) the process blocks until a message arrives in its mail-box (thereby unblocking the process). If the mail-box is not empty all required data is available and the process does not have to block. Each process tells its left neighbour through a sync mail-box that it has completed the computation of the previous entry.

We are using the quasi-destructive graph unification algorithm as presented in [Tom91]. It solves the efficiency problems in case of unification failures[7] most elegantly, while it is still easy to be altered to run concurrently.

7 Evaluation

In the following the implementations of the presented algorithms will be referred to as:

[6]This has a heavy impact on the time complexity of the first level parallelization. Since now an entry can contain more non-terminals than there are in the grammar the combination of two entries is not of linear complexity anymore. Theoretically the complexity became even exponential with respect to ambiguities in the grammar and the input string.

[7]If a unification algorithm creates copies of feature structures but eventually the unification fails, the copying is a waste of time and memory resources.

- `:sequ` – the sequential CYK algorithm (section 3.1)
- `:lin` – the first level parallelization presented in [Bar90] (section 3.2)
- `:sqr` – the naive parallelization using $O(n^2)$ processors (section 5.1)
- `:fork` – the further parallelized parsing algorithm (sections 5.2 and 5.3)

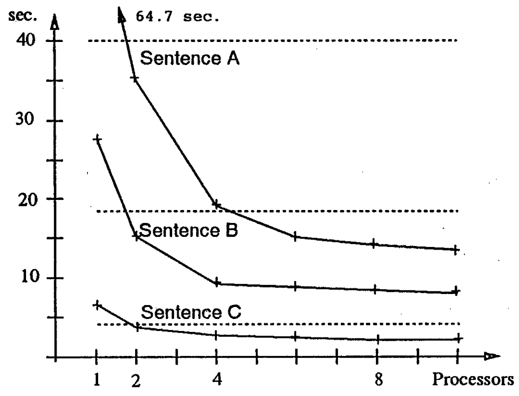

Figure 2: Parsing times for three sentences by *:fork*

Figure 2 shows the run-time of parsing three sentences by *:fork* compared to the sequential version of the algorithm (the dotted lines). The sentences are, A: ***onamae to gozyuusyo wo onegai shi masu*** ("Would you please give me your name and address?" 819 top-level unifications), B: ***soredewa kochira kara sochira ni tourokuyoushi wo ookuri itashi masu*** ("Then, I'll send you a registration form." 321 top-level unifications) and C: ***wakari mashita*** ("I see." 34 top-level unifications).

The synchronization overhead is compensated for when at least two or three processors are used. The speed up was 1:3 for the 10-processor run vs. the 1-processor run in all tests. It was at least 1:2 for the 10-processor run vs. the sequential reference version.

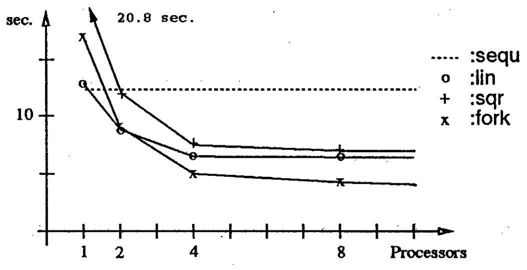

Figure 3: Comparison of all four versions

Figure 3 compares all versions of the presented algorithm for sentence D: *wakara nai ten ga gozaimashi tara watakushidomo ni itsu demo okiki kudasai.* ("Please feel free to ask if there's anything you don't understand." 141 top-level unifications).

The *:lin* version suffers from its not using all processors. This waste shows especially at its bad scaling: The run-time for more than four processors was more or less constant in all tests. The reason is, that *:lin* actually does not use most of the processors much. Still, this version is easy to implement and maybe interesting for very small parallel machines.

Finally, the *:sqr* version clearly is not usable at all. The reason is the large amount of sychronization necessary with this algorithm. Even, its implementation is more complicated than that of the other versions.

The run-time of *:fork* shows, that this is a better way to do things. Though it required more synchronization than :lin, it was faster with three or more processors. Also, it showed a better scaling.

8 Conclusion

We introduced the CYK parsing algorithm and its parallelization following [Bar90]. On this basis an efficient parallel unification-based parser has been presented. It used further parallelism for better utilization of the parallel machine and incorporated a unification algorithm. The strength of the presented parallel parsing algorithm lies in the combination of a fast unification algorithm with a parser that is parallelized with respect to the underlying architecture. The reduced parallelism, in contrast to massively parallel algorithms, resulted in good run-times on a medium-sized parallel machine. Even with as few as four processors, good speed-ups could be achieved.

References

[Bar90] D. T. Barnard, D. B. Skillicorn: *Parallel Parsing: A Status Report (chp. 4)*, Queen's University, 1990

[CLiP] Franz Inc.: *Allegro CLiP Manual*, 1989

[Fuj90] T. Fujioka, H. Tomabechi, O. Furuse, H. Iida: *Parallelization Technique for Quasi-Destructive Graph Unification Algorithm*, reprint of WGNL 80-7, IPSJ, 1990

[Gun87] T. Gunji: *Japanese Phrase Structure Grammar*, D. Reidel Publishing, 1987

[Kat90] S. Kato: *Performance Evaluation of Parallel Algorithms for Unification-Based Parsers*, reprint of WGNL 77-2, IPSJ, 1990

[Lan90] L. Langlois: *Parallel Parsing via the backward trace of systolic computations*, Proceedings of the Workshop of Parallel Computation, Kingston, 1990

[Mat89] Y. Matsumoto: *Natural Language Parsing Systems based on Logic Programming*, Thesis at Kyoto University, 1989

[Shi86] S. M. Shieber: *An Introduction to Unification-based Approaches to Grammar*, CSLI Lecture Notes No. 4, 1986

[Tom91] H. Tomabechi: *Quasi-Destructive Graph Unification*, Proceedings of ACL91

[You67] D. H. Younger: *Recognition and Parsing of Context-Free Languages in Time n^3*, Information and Control Vol. 10, p. 189–208, 1967