ICOT Technical Report: TR-0757

TR-0757

Derivation of the Parallel Bottom-Up Parser from the sequential Parser

by
K. Takahashi (Mitsubishi)

April, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
(03)3456-3191 ~ 5
Telex ICOT J32964
Minato-ku Tokyo 108 Japan

Institute for New Generation Computer Technology

Derivation of the Parallel Bottom-Up Parser from the Sequential Parser

Kazuko TAKAHASHI
Central Research Laboratory, Mitsubishi Electric Corporation
8-1-1, Tsukaguchi-Honmachi, Amagasaki, 661, JAPAN
(TEL) +81-6-497-7141
takahashi@sys.crl.melco.co.jp

Abstract

This paper describes the derivation of a parallel program from a nondeterministic sequential program using a bottom-up parser as an example.

The derivation procedure consists of two stages: exploitation of AND-parallelism and exploitation of OR-parallelism. An interpreter of the sequential parser BUP [Matsumoto et al 83] is first transformed so that processes for the nodes in a parsing tree can run in parallel. Then, the resultant program is transformed so that a nondeterministic search of a parsing tree can be done in parallel. The former stage is performed by hand-simulation, and the latter is accomplished by the compiler of ANDOR-II, which is an AND/OR parallel logic programming language[Takeuchi et al. 89]. The finally derived program, written in KL1(Kernel Language of the FGCS Project), achieves an all solution search without side effects. It corresponds to an interpreter of PAX[Matsumoto et al 87], a revised parallel version of BUP. This correspondence shows that the derivation method proposed in this paper is effective in deriving efficient parallel programs.

1 Introduction

Due to the recent progress in parallel machines, the demand for software suitable for the new parallel architecture has also increased. As a result, vigorous research and development on parallelism are currently under way. Derivation of a parallel program from a sequential program is one of the important issues in this research area.

Generally speaking, it is hard to grasp the behavior of parallel programs, and when we write a parallel program, we have to pay attention to the control mechanism such as synchronization and communication. It is sometimes easier to parallelize an available sequential program than to create a totally new program. However, in this case, we often cannot fully extract the AND- and OR-parallelism inherent in the original program. AND-parallelism arises from the parallel execution of conjunctive goals, and OR-parallelism arises from nondeterminacy.

In the field of logic programming, early works of committed-choice languages such as Concurrent Prolog[Shapiro 83], PARLOG[Clark and Gregory 84] and GHC[Ueda 86] accomplish only AND-parallelism, not OR-parallelism which Prolog supports with a

backtracking mechanism, since alternatives are abandoned after commitment in these languages. If one wants to express a nondeterministic phenomenon in these languages, he/she has to explicitly specifies an all solution search. It not only puts a heavy burden on the programmer, but also makes the program hard to understand.

To solve these problems, AND/OR-parallel programming languages which exploit the advantages of both Prolog and committed-choice languages have been developed [Yang and Aiso 86] [Clark and Gregory 87] [Haridi et al. 88] [Naish 87] [Kliger et al. 88].

ANDOR-II[Takeuchi et al. 89] is one of these languages. ANDOR-II provides a declarative description for nondeterminacy by classifying predicates into two classes: AND-predicates and OR-predicates. An ANDOR-II program is compiled into a KL1(Kernel Language of the FGCS Project) [1] program using a *coloring* scheme so that both AND- and OR-parallelism can be achieved efficiently. OR-parallelism in ANDOR-II is accomplished by AND-parallelism in the object program, whereas intrinsic AND-parallelism among conjunctive goals is preserved.

In this paper, we apply ANDOR-II to derive a parallel parser from a nondeterministic sequential parser. The derivation procedure consists of two stages: exploitation AND-parallelism and exploitation of OR-parallelism. An interpreter of the sequential parser BUP[Matsumoto et al 83] is first transformed so that the processes for the nodes in a parsing tree can run in parallel. Then, the resultant program is transformed so that a nondeterministic search of a parsing tree can be done in parallel. The former stage is performed by hand-simulation, and the latter is performed by the ANDOR-II compiler. The finally derived program, written in KL1, achieves an all solution search without side effects. It corresponds to an interpreter of PAX[Matsumoto et al 87], a revised parallel version of BUP [2].

In a PAX system, grammer rules are translated into a KL1 program by the translator, and parsing is performed by direct execution of the translated program. On the other hand, in our approach, parsing is interpretively performed. Paring performed in these two methods show essentially the same behavior. This correspondence shows that the derivation method proposed in this paper is effective in deriving efficient parallel programs.

In Section 2, the bottom-up parsing system BUP is described and the intrinsic parallelism is analyzed. In Section 3 and section 4, exploitation of AND-parallelism and that of OR-parallelism are described, respectively. In Section 5, the correspondence between the derived program and the PAX interpreter is shown and some issues regarding the derivation of parallel program from sequential program are also discussed.

Throughout the paper, familiarity with parallel logic programming languages such as GHC, PARLOG and Concurrent Prolog is assumed.

2 Preliminaries

2.1 BUP System

In this paper, we will discuss the bottom-up parsing system called BUP (Bottom-Up parser embedded in Prolog)[Matsumoto et al 83]. It is a bottom-up left corner parser for the grammar rules written in Definite Clause Grammar(DCG).

[1] KL1[Chikayama et al. 88] is the language with some controls added to GHC.

[2] The system developed in a sequential language is particularly called SAX(Sequential Analyzer for syntaX and semantiCS). Basically SAX and PAX employ the same algorithm. We use the name "PAX" throughout the paper, since we consider the system on parallel environment.

```
% do :- goal(s,[the,man,walk],[]).

C1: goal((P,Q),S0,S2) :- goal(P,S0,S1),goal(Q,S1,S2).
C2: goal(C,S,S1) :- dict(F,S,S2), derive(F,S2,C,S1).

C3: derive(F,S,F,S).
C4: derive(F,S2,C,S1) :-
        rule1((Lemma <= (F,Rest))),
        goal(Rest,S2,S3), derive(Lemma,S3,C,S1).
C5: derive(F,S2,C,S1) :-
        rule2((Lemma <= F)), derive(Lemma,S2,C,S1).

C6: dict(F,[X|S],S) :- rule((F <= [X])).
C7: rule1((A <= (B,C)) :- rule((A <= (B,C)).
C8: rule2((A <= B)) :- rule((A <= B)), \+(B=(_,_)), \+(B=[_]).
```

Figure 1: BUP Interpreter

```
rule((s <= (np,vp))).          rule((det <= [the])).
rule((np <= (det,noun))).      rule((adj <= [beautiful])).
rule((noun <= (adj,noun))).    rule((noun <= [man])).
rule((noun <= (noun,relc))).   rule((noun <= [woman])).
rule((noun <= (that,s))).      rule((verb <= [walks])).
rule((vp <= (verb,np))).       rule((verb <= [loves])).
rule((vp <= verb)).
```

Figure 2: Rule Part

In a parsing tree, each node corresponds to a terminal or nonterminal symbol, and each leaf node corresponds to a terminal symbol. In BUP, starting from leaves, parsing proceeds in a bottom-up manner. When a subtree T is constructed, a larger subtree tries to grow by establishing itself at the root of T on its left corner.

The ordinal BUP system is a translator which translates the grammar rules to a bottom-up parser in which the grammer rules are embedded. Although it enables high performance, it is hard to understand and also difficult to modify the inference strategy. It is more suitable for our purpose to separate the inference part from the rule part and consider an interpreter. The inference part of an interpreter can easily been written in Prolog (Figure 1) [3]. An example of the rule part is shown in Figure 2. Although, BUP also employs top-down expectation for efficiency, we ignore it for simplicity and focus only on the essential part [4].

[3] This interpreter is based on the one provided in [Takeuchi and Furukawa 86].

[4] Top-down expectation can be treated in the same way.

2.2 Parallelism

Our purpose is to derive an interpreter for a parallel parsing system from this interpreter by extracting both AND- and OR-parallelism. In a parallel program, we regard each node in a parsing tree as a process. Each process has a communication channel with its adjacent process, and communicates through this channel by sending a message. As each node in a parsing tree corresponds to the goal *derive*, each process performs the following two types of jobs:

1. **rule expansion** Expand the grammar rule in which the symbol occupies the left corner of a parsing tree, and pass the information on the current subtree under construction to its adjacent process.

2. **subtree completion** Receiving information, try to complete the current subtree.

AND-parallelism arises from the parallel execution of the processes for the nodes in a parsing tree. In clause `C4`, the process which expands a rule for some node in a parsing tree calls the process for its adjacent node. These processes should not properly be called mutually but they should be created simultaneously. In order to accomplish this, we adopt a strategy that processes for leaf nodes in a parsing tree are initially created and allowed to communicate with each other via channels between adjacent processes.

OR-parallelism arises from the parallel search for alternative solutions. There are three possibilities for the head unification of a goal *derive* either with clauses `C3`, `C4` or `C5`. This search should be achieved without side effects such as Prolog's backtracking mechanism. In order to accomplish this, we adopt a strategy that multiple solutions due to the nondeterminism are merged in stream form and put into the channels.

3 Exploitation of AND-Parallelism

The first stage is exploitation of AND-parallelism.

In this stage, we generate new clauses by analyzing the program in Figure 1, with the aid of a well-known unfold/fold technique [Tamaki and Sato 84].

We expand a process network whose node corresponds to each terminal symbol, and the information on the current subtree under construction is flown as a message among these processes. Note that the generated program still provides a declarative description for nondeterminacy.

Expansion of a Network

First of all, we a-priori define two new predicates. The one is the predicate *parse* to expand a network at an initial state, and the other is the predicate *exp* to represent the jobs for each node in the network. The arguments of *exp* denote the identifier of the process, the input channel and the output channel. *Exp*1 and *exp*2 correspond to rule expansion and subtree completion, respectively.

```
C9:  parse([X|S],In,Out) :- exp(X,In,IO), parse(S,IO,Out).
C10: parse([],In,Out) :- Out=In.

C11: exp(X,In,Out) :- exp1(X,In,Out).
C12: exp(X,In,Out) :- exp2(X,In,Out).
```

Identification of a Unit Process

In Figure 1, a unit process corresponding to *exp* is unclear, since *derive* and *goal* are called mutually (clauses C2 and C4) and *goal* called from *derive* (C4) may distribute more than two *goal* processes (C1) which may call another *derive* (C2). Our purpose is to eliminate this mutual recursion, and to make each unit process perform a job only with its corresponding symbol.

Unfold clause C1 at its first goal, *goal*. This *goal* is called either from the first goal of clause C1 or from the second goal of clause C4. In both cases, if the first argument is in the form of (P,Q), then P is not in the form of $(P1,P2)$. Therefore, this goal cannot unify with clause C1. Thus, it is unfolded only by clause C2. As a result we get the clause C13.

```
C13: goal((P,Q),S0,S2) :- dict(F,S0,A),derive(F,A,P,S1),goal(Q,S1,S2).
```

In clause C4, the former part (*rule*1) performs rule expansion of a symbol, and the latter part(*goal* and *derive*) performs jobs of the adjacent symbol. These processes should be divided into two clauses. Thus, we introduce a new definition *cont*.

```
--- definition
C14: cont(Rest,Lemma,S2,C,S1):- goal(Rest,S2,S3), derive(Lemma,S3,C,S1).
```

Transformation process proceeds by using definition C14.

```
--- unfold C14 at its first goal
C15: cont((P,Q),Lemma,S2,C,S1) :-
        dict(F,S2,A), derive(F,A,P,B), goal(Q,B,S3),
        derive(Lemma,S3,C,S1).
C16: cont(Rest,Lemma,S2,C,S1) :-
        dict(F,S2,A), derive(F,A,Rest,B), derive(Lemma,B,C,S1).

--- fold body goals of C15 by C14
C17: cont((P,Q),Lemma,S2,C,S1) :-
        dict(F,S2,A), derive(F,A,P,B), cont(Q,Lemma,B,C,S1).

--- fold body goals of C4 by C14
C18: derive(F,S2,C,S1) :-
        rule1((Lemma <= (F,Rest))), cont(Rest,Lemma,S2,C,S1).
```

So far, we have obtained the set of clauses { C2,C3,C5,C6,C7,C8,C13,C16,C17,C18 } and the definition C14.

We expect a unit process to perform its job only with the corresponding symbol and to leave the other job to the adjacent process. We create a unit process which satisfies this condition. The unit process *exp* has two kinds of jobs *exp*1 and *exp*2. We create *exp*1 and *exp*2 based on *derive* and *cont*, respectively.

Creation of *exp*1 from *derive*

First of all, we modify *derive* so that it sends the information as an output instead of calling *cont*. Secondly, we remove unnecessary arguments of *derive*. In Figure 1, the third argument of *derive* is used only for the terminal condition (the state of obtaining a complete subtree). This argument is unnecessary for *exp*1, since *exp*1 performs only rule expansion. The rest of the arguments correspond to identifier, input channel and output channel, in that order. Note that *derive* called from the body part (clause C5)

is a new process for an upper concept and it should be able to call both *exp*1 and *exp*2. Therefore, we replace it by *exp*.

As a result of this analysis, we obtain the following new definition for *exp*1.

```
C19: exp1(F,S2,S1) :-
        rule1((Lemma <= (F,Rest))), S1=(Rest,Lemma,S2).
C20: exp1(F,S2,S1) :-
        rule2((Lemma <= F)), exp(Lemma,S2,S1).
```

Creation of *exp*2 from *cont*

First of all, we move *dict* so that it is called when a network is expanded at an initial state. In BUP, *dict* is called only for a terminal symbol (i.e. a word in a given sentence) and not called for a nonterminal symbol (i.e. upper concept). Therefore, *dict* should be called when the proceses for terminal symbols are created.

Thus, the clause *parse* for expanding a network should be modified as follows.

```
C21: parse'([X|S],In,Out) :-
        dict(F,[X|S],S), exp(F,In,IO), parse'(S,IO,Out).
```

It exempts *cont* from calling *dict* from its body part. Therefore, *cont* is replaced by *cont'*.

```
C22: cont'(Rest,Lemma,[X|S],F,C,S1) :-
        derive(F,S,Rest,B), derive(Lemma,B,C,S1).
C23: cont'((P,Q),Lemma,[X|S],F,C,S1) :-
        derive(F,S,P,B), cont'(Q,Lemma,B,C,S1).
```

Secondly, we unfold each of these clauses at their first goal, *derive*. They are unfolded only by clause C3 which corresponds to a subtree completion.

```
C24: cont'(F,Lemma,[X|S],F,C,S1) :- derive(Lemma,S,C,S1).
C25: cont'((F,Q),Lemma,[X|S],F,C,S1) :- cont'(Q,Lemma,S,C,S1).
```

Thirdly, we modify *cont'* so that it sends the information as an output instead of calling *cont'*.

Next, we remove the unnecessary arguments of *cont'*. Several arguments become unnecessary because of unfolding. The fifth argument of *cont'* is originally used for the terminal condition, however, this process is already performed by unfolding. The variable X appearing in the third argument is no longer used. Therefore, we remove these arguments.

Finally, we rearrange the remaining arguments. The fourth argument, F, corresponds to an identifier of the process, the sixth argument, $S1$, corresponds to output channel, and the rests correspond to input arguments. *Derive* appearing in the body of C24 is replaced by *exp*, from the previous discussion.

At last, we obtain the following definition for *exp*2.

```
C26: exp2(F,(F,Lemma,S),S1) :- exp(Lemma,S,S1).
C27: exp2(F,((F,Q),Lemma,S),S1) :- S1=(Q,Lemma,S).
```

Addition of Special Processes

At the end of the first stage in the transformation procedure, the processes checking the termination are added. In C28 and C29, the terms 'begin' and 'end' denote the beginning and terminal positions in the sentence to be parsed.

```
% do :- parse([the,man,walks],begin,Out), fin(Out).

C29: fin(end).

C21: parse([X|S],In,Out) :- dict(F,[X|S],S), exp(F,In,IO),
        parse(S,IO,Out).
C10: parse([],In,Out) :- Out=In.

C11: exp(X,In,Out) :- exp1(X,In,Out).
C12: exp(X,In,Out) :- exp2(X,In,Out).

C19: exp1(X,In,Out) :- rule1((Lemma <= (X,Rest))),
        Out=(Rest,Lemma,In).
C20: exp1(X,In,Out) :- rule2((Lemma <= X)), exp(Lemma,In,Out).

C26: exp2(X,(X,Lemma,In),Out) :- exp(Lemma,In,Out).
C27: exp2(X,((X,Rest),Lemma,In),Out) :- Out=(Rest,Lemma,In).
C28: exp2(s,begin,Out) :- Out=end.
```

Figure 3: Transformed Program by Hand-Simulation

```
C28: exp2(s,begin,Out) :- Out=end.
C29: fin(end).
```

Figure 3 shows the finally obtained program.

The above transformation procedure indicates that the structure of the original program is so complicated. Transformation is not simple, and we often make large jumps to obtain a program suitable for parallel execution. However, sophisticated methods such as unfold/fold technique helps alleviate the difficulty.

4 Exploitation of OR-Parallelism

The second stage of the transformation procedure is exploitation of OR-parallelism. This stage is performed by the ANDOR-II compiler.

4.1 Language ANDOR-II

ANDOR-II is a parallel logic programming language which exploits AND- and OR-parallelism. In ANDOR-II, predicates are classified into two classes : AND-predicates and OR-predicates. An AND-predicate is defined by a set of guarded clauses. An OR-predicate is defined by a set of non-guarded clauses. A clause of either type can contain both AND-predicates and OR-predicates in its body part. Every predicate in ANDOR-II must be associated with a mode declaration which specifies each argument as either input(denoted by '+') or output('−'). Similar to flat languages, a goal in a guard part is restricted to a test predicate.

An ANDOR-II program for the program in Figure 3 can be generated with a slight modification (Figure 4). Nondeterministic predicates *exp* and *exp*1 are regarded as OR-predicates, and the rests are AND-predicates. As for an unary predicate *rule*, the

argument has to be divided, since the mode of each argument of each predicate is either input or output in ANDOR-II. Given in the form of $rule((A <= (B, C))$, the variable B is used for an input, while A and C are used for outputs.

The execution of ANDOR-II exploits both AND- and OR-parallelism as follows. Body goals in each clause are executed in parallel. If an OR-predicate is invoked, possible worlds are created. Each world is painted by a distinct color and AND-parallel computation proceeds on each world (Figure 5).

The solution obtained on a world is painted with the color of the world and all the solutions associated with their colors are packed in stream form, which we call *a colored vector*. Some goals have an extra cover called *shell* to handle these types of data. If a goal with a shell receives such a colored vector, it creates the worlds each of which is painted with the color of each element. Computation for each data proceeds in each colored world and the result is painted with the color of the world. They are recomposed into a vetor form (Figure 6). Basic computations such as unification and arithmetic operations are applied only to a tuple of data sharing the same color.

A color may be refined as the computation proceeds. Therefore, color is considered as a history of the clause selections during the computation.

4.2 Compilation Based on Coloring Scheme

An ANDOR-II program is compiled into a KL1 program using a coloring scheme. OR-parallelism in ANDOR-II is realized by AND-parallelism in the object program, whereas intrinsic AND-parallelism among conjunctive goals is preserved. The fundamental idea is described in [Takeuchi et al. 89][Takahashi et al. 90].

The main job of the compiler is the transformation of OR-predicates and the creation of appropriate shells for the goals.

An OR-predicate is transformed into a deterministic predicate in KL1. Clauses defining an OR-predicates are realized by AND-parallel execution of conjunctive goals corresponding to their computations. And their solutions are merged to recompose a colored vector. For instance, *exp* is transformed as follows, where *exp_Core_1* and *exp_Core_2* invoke *exp1* and *exp2*, respectively. $C1, C2$ denote new colors created by adding a new color element to the corrent color C.

```
exp1_Core(X,In,Out,w(C)) :- true |
    add_Color_element(C,C1),
    add_Color_element(C,C2),
    exp1_Core_1(X,In,Out1,w(C1)),
    exp1_Core_2(X,In,Out2,w(C2)),
    merge({Out1,Out2},Out).
```

The cover of *shell* is put on goals that might receive a colored vector. There are different types of shells depending on the data type. The compiler analyzes the goal which needs a shell and the appropriate shell type [5]. Each goal is transformed into one covered with a proper shell.

For example, take the first clause of *parse* in Figure 4. *Parse* receives the output of *exp* via the channel *IO*, where *exp* is an OR-predicate. Therefore, the goal *parse* is covered with a shell, and this clause is transformed as follows.

[5] ANDOR-II system has another compilation method which is described in [Takeuchi and Takahashi 91]. In this method, data analysis is not done during compilation but done at the execution.

```
% do :- true | parse([the,man,walks],begin,Out), fin(Out).

:- mode fin(+).
 fin(end) :- true | true.

:- mode parse(+,+,-).
 parse([X|S],In,Out) :- true |
        dict(F,[X|S],S), exp(F,In,IO), parse(S,IO,Out).
 parse([],In,Out) :- true | Out=In.

:- mode exp(+,+,-).
:- or_relation exp/3.
 exp(X,In,Out) :- exp1(X,In,Out).
 exp(X,In,Out) :- exp2(X,In,Out).

:- mode exp1(+,+,-).
:- or_relation exp1/3.
 exp1(X,In,Out) :- rule1(Lemma,X,Rest), Out=(Rest,Lemma,In).
 exp1(X,In,Out) :- rule2(Lemma,X), exp(Lemma,In,Out).

:- mode exp2(+,+,-).
 exp2(X,(X,Lemma,In),Out) :- true | exp(Lemma,In,Out).
 exp2(X,((X,Rest),Lemma,In),Out) :- true | Out=(Rest,Lemma,In).
 exp2(s,begin,Out) :- true | Out=end.

:- mode dict(-,+,-).
 dict(F,[X|S],S) :- true | rule(F,[X]).

:- mode rule1(-,+,-).
 rule1(A,B,C) :- true | rule(A,B,C).
:- mode rule2(-,+).
 rule2(A,B) :- true | rule(A,B), B\=[_].

:- mode rule(-,+,-).
rule(Lemma,np,Rest) :- true | Lemma=s, Rest=vp.
rule(Lemma,det,Rest) :- true | Lemma=np, Rest=noun.
rule(Lemma,adj,Rest) :- true | Lemma=noun, Rest=noun.
rule(Lemma,noun,Rest) :- true | Lemma=noun, Rest=relc.
rule(Lemma,verb,Rest) :- true | Lemma=vp, Rest=np.

:- mode rule(-,+).
rule(Lemma,verb) :- true | Lemma=vp.
rule(Lemma,[the]) :- true | Lemma=det.
rule(Lemma,[beautiful]) :- true | Lemma=adj.
rule(Lemma,[man]) :- true | Lemma=noun.
rule(Lemma,[woman]) :- true | Lemma=noun.
rule(Lemma,[walks]) :- true | Lemma=verb.
rule(Lemma,[loves]) :- true | Lemma=verb.
```

Figure 4: ANDOR-II Program

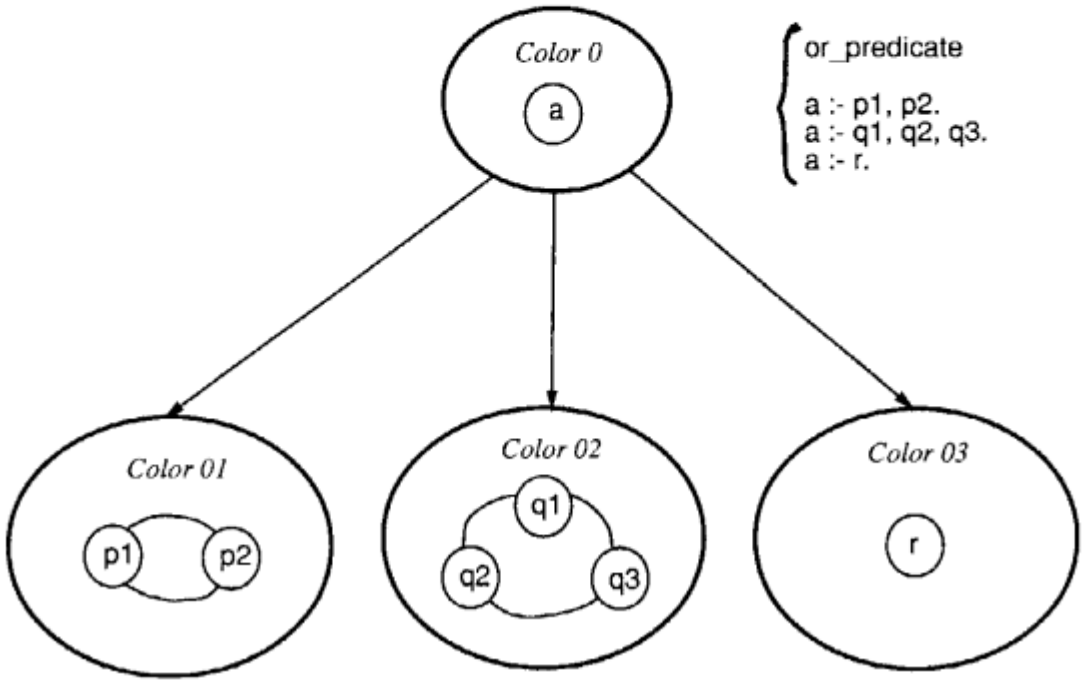

Figure 5: Computation Model of ANDOR-II

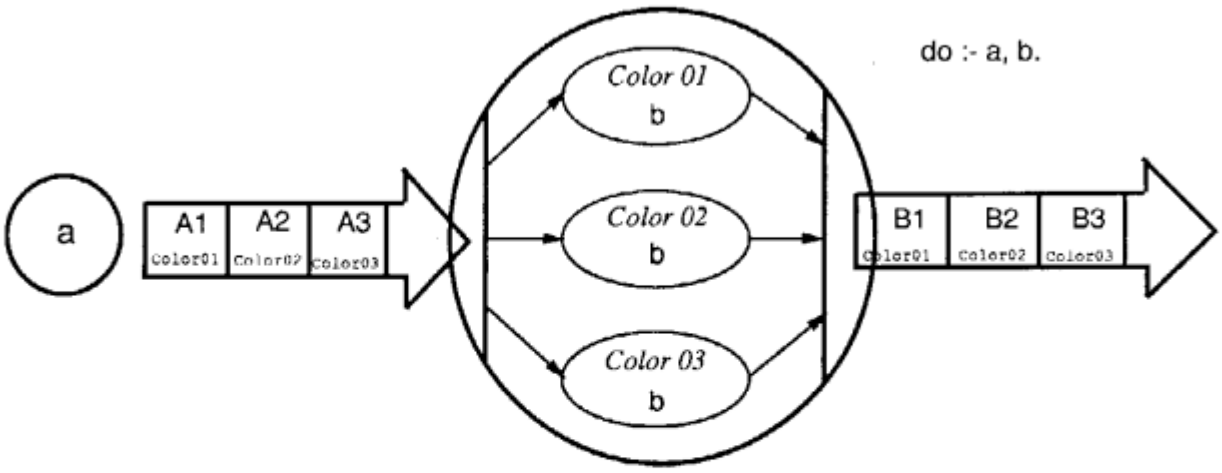

Figure 6: Shell and Colored Vector

```
parse_Core([X|S],In,Out,w(C)) :- true |
        dict_Core(F,[X|S],S,w(C)), exp_Core(F,In,IO,w(C)),
        parse_Shell(S,IO,Out).
```

Shell is a structure which handles a colored vector. It decomposes a set of input colored vectors into a tuple of values, passes them to the corresponding core processes, and collect the output values together to make a set of vectors. Each core process corresponds to computation with one color. Core processes are executed in parallel and the solutions are put into output channels as soon as they are generated by fair merge operators.

For example, the shell for the predicate *parse* is defined as follows.

```
parse_Shell(X,[v(Y,Cy)|Ys],Z) :- true |
        parse_Core(X,Y,Z1,w(Cy)),
        parse_Shell(X,Ys,Z2),
        merge({Z1,Z2},Z).
parse_Shell(X,[],Z) :- true | Z=[].
```

5 Discussion

5.1 Comparison with PAX

The PAX system is a translator from grammer rules written in DCG into a parallel program such as KL1 in which the rules are embedded. The generated program is a parser for the given grammer rules. It performs parsing so that AND/OR-parallelism is fully exploited. If an element of a parsing tree is derived, then it needs no retrial, and parsing without side effects is performed.

The generated parser for the rules in Figure 2 is shown in Figure 7[Matsumoto et al 87], and the result of parsing a sentence $[the, man, walks]$ is shown in Figure 8.

In this parsing, the processes for the words in the sentence to be parsed are generated at an initial state. *Begin* is the seed of the data flown among processes. A process for each symbol receives the information from its left-adjacent process, performs rule expansion and subtree completion in parallel, merges these results into stream form, and sends it to the right-adjacent process. For example, the process for the symbol *np*1 generates $noun(X)$ for an input X, and the process for the symbol *np* generates the merged stream of the results of *np*1 and *np*2. Therefore, if X is in stream form, the output of *np* is a stream whose element contains a stream.

In the translated program, the processes to compose/decompose a stream are explicitly represented, and the treatment of a failure case is also explicitly represented. It is troublesome to write these processes directly. The ANDOR-II compiler also generates a code with such treatments from a declarative description.

The main job of the ANDOR-II compiler is the transformation of OR-predicates and the creation of appropriate shells for the goals. All alternatives for an OR-predicate are executed in AND-parallel and the solutions are collected in stream form. This stream is decomposed and processed for each element, and the result is recomposed into stream form by the special structure *shell*. Those are the same jobs performed by the translated program of the PAX translator.

It is easy to observe that the interpreter derived from the ANDOR-II compiler shows fundamentally the same behavior as the parser generated by the PAX translator. The difference is that in the former, solution obtained in a world is associated with its own

```
% do :- the([begin],D2),man(D2,D3),walks(D3,D4),fin(D4).

fin([end]) :- true.
otherwise.
fin([_|T]) :- fin(T).

np(X,Y) :- Y=[np(X)|T], np2(X,T).
np(X,Y) :- np1(X,Y1), np2(X,Y2), merge({Y1,Y2},Y).

np1(X,Y) :- Y=[np(X)].
np2([],Y) :- Y=[].
np2([verb(X)|T],Y) :- vp(X,Y1), np2(T,Y2), merge({Y1,Y2},Y).
otherwise.
np2([_|T],Y) :- np2(T,Y).

vp([],Y) :- Y=[].
vp([np(X)|T],Y) :- sentence(X,Y1), vp(T,Y2), merge({Y1,Y2},Y).
otherwise.
vp([_|T],Y) :- vp(T,Y).

det(X,Y) :- Y=[det(X)].
adj(X,Y) :- Y=[adj(X)].

noun(X,Y) :- Y=[noun(X)|T], noun2(X,T).
noun(X,Y) :- noun1(X,Y1), noun2(X,Y2), merge({Y1,Y2},Y).

noun1(X,Y) :- Y=[noun(X)].
noun2([],Y) :- Y=[].
noun2([det(X)|T],Y) :- np(X,Y1), noun2(T,Y2), merge({Y1,Y2},Y).
noun2([adj(X)|T],Y) :- noun(X,Y1), noun2(T,Y2), merge({Y1,Y2},Y).
otherwise.
noun2([_|T],Y) :- noun2(T,Y).

verb(X,Y) :- Y=[verb(X)|T], vp(X,T).
verb(X,Y) :- verb1(X,Y1), verb2(X,Y2), merge({Y1,Y2},Y).

verb1(X,Y) :- Y=[verb(X)].
verb2(X,Y) :- vp(X,Y).

sentence([],Y) :- Y=[].
sentence([begin],Y) :- Y=[end].
otherwise.
sentence([_|T],Y) :- sentence(T,Y).

the(X,Y) :- det(X,Y).              woman(X,Y) :- noun(X,Y).
beautiful(X,Y) :- adj(X,Y).        walks(X,Y) :- verb(X,Y).
man(X,Y) :- noun(X,Y).             loves(X,Y) :- verb(X,Y).
```

Figure 7: Parser Generated by PAX Translator

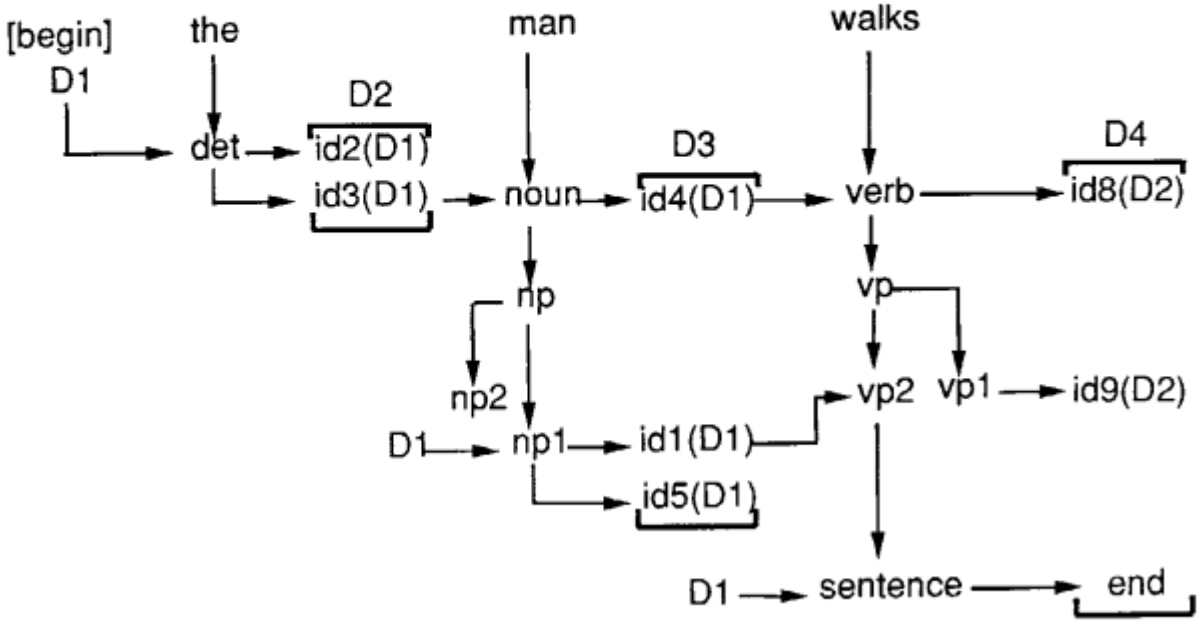

Figure 8: Example of Parsing by PAX

color and multiple solutions are handled in the form of a colored vector, while in the latter, each element of the stream is simple data without a color. The reason for coloring is that when a process has more than one input channel, we have to check whether a set of data comes from the same world, since only the computation for the set of data sharing the same color can be permitted. Generally speaking, these possible worlds may proliferate as computation proceeds and its number is unknown. Moreover, if you treat the cyclic structure such as a feedback system, this mechanism becomes more complicated. This causes the essential difficulty in handling AND/OR-parallelism, and at the same time, it is the main reason for the overhead.

However, restricted to this example of a parser, coloring is unnecessary, since the number of possibilities can be determined by a statical analysis, all predicates have only single input arguments, and there is no cyclic structure. An optimized program without a color is exactly a PAX interpreter.

In the appendix, we show an optimized program of an object program generated by the ANDOR-II compiler.

5.2 Issues on Program Transformation

To derive a parallel program from a sequential program, the approach explained here consists of two stages: exploitation of AND-parallelism and exploitation of OR-parallelism. The former stage can be considered as a process of changing the strategy for extracting high parallelism. Another approach is a direct compilation of a BUP interpreter by the ANDOR-II compiler, after slight syntactical modification. In this case, a nondeterministic process of *derive* is regarded as an OR-predicate and the other predicates as

AND-predicates. However, the derived program does not extract parallelism to the full extent, since processes for adjacent symbols are not created simultaneously, but rather sequentially.

From the viewpoint of program transformation, it is important to justify the transformation method. Some method such as the unfold/fold method preserves the minimal Herbrand model during the transformation and guarantees termination. However, in the case of transformation from a sequential program to a parallel program, it is generally hard to justify the transformation in this sense, since the structure of the program is so complicated. In this case, it is reasonable and enough to take such a transformation into account that can derive the program which preserves the I/O relation of the original program.

Accomplishing nondeterminacy is also an important issue. If one writes a declarative program, it tends to have more nondeterminacy. Several methods have been reported regarding the derivation of a parallel program from a nondeterministic sequential program [Ueda 86b] [Ueda 87] [Tamaki 86] [Takeuchi et al. 89]. A layered-stream method proposed in [Okumura and Matsumoto 83] is a kind of programming paradigm in which recursively defined data structure called layered stream is used. All the values in the same layer gives alternative solutions to the same subproblem. Although a layered-stream achieves high parallelism, it is burdensome for a novice user to write a program using this paradigm. For a program such as the parser discussed here, for which coloring is unnecessary, color can be removed, and the optimized program is equivalent to one which is written using a layered-stream method. Therefore, ANDOR-II which provides a declarative description is advantageous for the same class of problems.

6 Concluding Remarks

In this paper, we described the derivation of a parallel program from a nondeterministic sequential program using a bottom-up parser as an example. An interpreter of the sequential parser BUP is first transformed so that processes for the nodes in a parsing tree can run in parallel. Then, the resultant program is transformed so that a nondeterministic search of a parsing tree can be done in parallel. The former stage is performed by hand-simulation and the latter stage is performed by the ANDOR-II compiler. The finally derived program in KL1 achieves an all solution search without side effects, which corresponds to an interpreter of PAX.

In a PAX system, grammer rules are translated into a Prolog program by the translator, and parsing is performed by direct execution of the translated program. On the other hand, in our approach, parsing is interpretively performed. Paring performed in these two methods show essentially the same behavior. This correspondence shows that the derivation method proposed in this paper is effective in deriving efficient parallel programs.

For future works, we need more applications to evaluate this transformation method.

Acknowledgements

The author would like to thank Dr. A. Takeuchi of Sony CSL for helpful comments. This research was done as a subproject of the Fifth Generation Computer Systems (FGCS) project. She would like to thank Dr. K. Fuchi, Director of ICOT, for the opportunity of doing this research and Dr. K. Furukawa, Vice Director of ICOT, and Dr. R. Hasegawa, the Chief of Fifth Laboratory, for their advice and encouragement.

References

[Chikayama et al. 88] Chikayama,T., H.Sato and T.Miyazaki, "Overview of the Parallel Inference Machine Operating System(PIMOS)," Proc. of Int. Conf. on Fifth Generation Computing Systems, pp.745-754, 1988.

[Clark and Gregory 84] Clark,K.L. and S.Gregory, "PARLOG: Parallel Programming in Logic," Research Report DOC 81/16, Imperial College of Science and Technology,1984.

[Clark and Gregory 87] Clark,K.L. and S.Gregory, "PARLOG and Prolog United," Proc. of 4th Int. Conf. on Logic Programming, pp.927-961,1987.

[Haridi et al. 88] Haridi,S. and P.Brand, "ANDORRA Prolog - An Integration of Prolog and Committed Choice Languages," Proc. of Int. Conf. on Fifth Generation Computing Systems, pp.745-754, 1988.

[Kliger et al. 88] Kliger,S., E.Yardeni, K.Kahn and E.Shapiro, "The Language FCP(;,?)," Proc. of Int. Conf. on Fifth Generation Computing Systems, pp.763-773, 1988.

[Matsumoto et al 83] Matsumoto,Y., H.Tanaka, H.Hirakawa H.Miyoshi and H.Yasukawa, "BUP: A Bottom-Up Parser Embedded in Prolog," New Generation Computing, pp.145-158, Vol.1, No.2, 1983.

[Matsumoto et al 87] Matsumoto,Y, "A Parallel Parsing System for Natural Language Analysis," New Generation Computing, pp.63-78, Vol.5, No.1, 1987.

[Naish 87] Naish,L., "Parallelizing NU-Prolog," Logic Programming, Proc. of 5th Int. Conf. and Symposium, pp.1546-1564, 1988.

[Okumura and Matsumoto 83] Okumura,A. and Y.Matsumoto, "Parallel Programming by Layered-Stream Methodology," pp.224-231 Proc.of Symposium on Logic Programming, 1987.

[Shapiro 83] Shapiro,E., "A Subset of Concurrent Prolog and Its Interpreter," ICOT TR-003, 1983.

[Takahashi et al. 87] Takahashi,K. and A.Takeuchi, "Generation of Parallel Parser by ANDOR-II: An Inference System for Concurrent Systems," Proc. of 4th National Conference of Japan Socicty for Software Science and Technology, pp.423-426, 1987(In Japanese).

[Takahashi et al. 90] Takahashi,K., A.Takeuchi and T.Yasui, "A Parallel Problem Solving Language ANDOR-II and Its Parallel Implementation," ICOT TR-558, 1990.

[Takeuchi and Furukawa 86] Takeuchi,A.and K.Furukawa, "Partial Evaluation of Prolog Programs and Its Application to Meta Programming," Proc. of IFIP Congress 86, H.-J. Kugler(ed), North-Holland, pp.415-420, 1986.

[Takeuchi et al. 89] Takeuchi,A., K.Takahashi and H.Shimizu, "A Parallel Problem Solving Language for Concurrent Systems," Concepts and Characteristics of Knowledge-Based Systems, M.Tokoro,Y.Anzai and A.Yonezawa(eds.),North-Holland,pp.267-296,1989.

[Takeuchi and Takahashi 91] Takeuchi,A. and K.Takahashi, "An Operational Semantics of AND- and OR- Parallel Logic Programming Language, ANDOR-II," LNCS-491, Concurrency: Theory, Language, and Architecture, pp.173-209,Springer-Verlag, 1989.

[Tamaki 86] Tamaki,H., "Stream-based Compilation of Ground I/O Prolog into Committed-choice Languages," Proc.of 4th Int. Conf. of Logic Programming pp.376-393,1987.

[Tamaki and Sato 84] Tamaki,H. and T.Sato, "Unfold/fold Transformation of Logic Programs," Proc. of 2nd Int. Conf. on Logic Programming, pp.376-393,1987.

[Ueda 86] Ueda,K., "Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard," ICOT TR-208, 1986.

[Ueda 86b] Ueda,K., "Making Exhaustive Search Programs Deterministic," Proc. of 3rd Int. Conf. on Logic Programming, LNCS 225, Springer, pp.270-282, 1986.

[Ueda 87] Ueda,K., "Making Exhaustive Search Programs Deterministic, Part II," Proc. of 4th Int. Conf. of Logic Programming, pp.356-375, 1987.

[Yang and Aiso 86] Yang,R. and H.Aiso, "P-Prolog: A Parallel Logic Language Based on Exclusive Relation," Proc. of 3rd Int. Conf. of Logic Programming, pp.255-269, 1986.

APPENDIX

```
fin_Shell([],Out) :- true | Out=[].
fin_Shell([A|B],Out) :- true |
    fin_Core(A,R), fin_Shell(B,Rs), merge({R,Rs},Out).

fin_Core(end,Result) :- true | Result=success.
otherwise.
fin_Core(_,Result) :- true | Result=[].

parse_Shell(A,[B|C],D) :- true |
    parse_Core(A,B,E), parse_Shell(A,C,F), merge({E,F},D).
parse_Shell(A,[],E) :- true | E=[].

parse_Core([X|S],In,Out) :- true |
    dict_Core(F,[X|S],S), exp_Core(F,In,IO), parse_Shell(S,IO,Out).
parse_Core([],In,Out) :- true | Out=In.
otherwise.
parse_Core(_,_,C) :- true | C=[].

exp_Core(A,B,C) :- true |
    exp_Core_1(A,B,F), exp_Core_2(A,B,G),merge({F,G},D).

exp_Core_1(A,B,C) :- true | exp1_Core(A,B,C).
exp_Core_2(A,B,C) :- true | exp2_Core(A,B,C).

exp1_Core(A,B,C) :- true |
    exp1_Core_1(A,B,D), exp1_Core_2(A,B,E), merge({D,E},C).

exp1_Core_1(X,In,Out) :- true |
        rule1_Core(Lemma,X,Rest), Out=[(Rest,Lemma,In)].
exp1_Core_2(X,In,Out) :- true |
        rule2_Core(Lemma,X), exp_Core(Lemma,In,Out).

exp2_Core(X,(X,Lemma,In),Out) :- true | exp_Core(Lemma,In,Out).
exp2_Core(X,((X,Rest),Lemma,In),Out) :- true | Out=[(Rest,Lemma,In)].
exp2_Core(s,begin,Out) :- true | Out=[end].
otherwise.
exp2_Core(_,_,Out) :- true | Out=[].

dict_Core(F,[X|S],S) :- true | rule_Core(F,[X]).

rule1_Core(A,B,C) :- true | rule_Core(A,B,C).
rule2_Core(A,B) :- true | rule_Core(A,B), B\=[_].

rule_Core(Lemma,np,Rest) :- true | Lemma=s, Rest=vp.
rule_Core(Lemma,det,Rest) :- true | Lemma=np, Rest=noun.
rule_Core(Lemma,adj,Rest) :- true | Lemma=noun, Rest=noun.
rule_Core(Lemma,noun,Rest) :- true | Lemma=noun, Rest=relc.
rule_Core(Lemma,verb,Rest) :- true | Lemma=vp, Rest=np.
```

```
rule_Core(Lemma,verb) :- true | Lemma=vp.
rule_Core(Lemma,[the]) :- true | Lemma=det.
rule_Core(Lemma,[beautiful]) :- true | Lemma=adj.
rule_Core(Lemma,[man]) :- true | Lemma=noun.
rule_Core(Lemma,[woman]) :- true | Lemma=noun.
rule_Core(Lemma,[walks]) :- true | Lemma=verb.
rule_Core(Lemma,[loves]) :- true | Lemma=verb.
```