International Research Journal of Advanced Engineering and Science
ISSN (Online): 2455-9024

A Parallel Architecture for Parsing Proposition Logic

Weihan Huang
Master of Computer Science Department, State University of New York, at Buffalo, U.S.A.
Master of Physics Department, National Hsing Hua University, Taiwan
Email address: weihanh@yahoo.com.tw

Abstract— *In this paper I present a parallel architecture for parsing proposition logic sentences. Firstly, I will give an introduction to proposition logic. And then I will give some example sentences of proposition logic, the context free grammar of proposition logic, and the parse trees of example sentences. Next, I design a single thread algorithm specialized for parsing proposition logic sentences. What follows is an introduction to do parallel processing in natural language programming, in which the syntax, semantics and pragmatics of parallel programming are shown. Then I will present a parallel architecture specialized for parsing proposition logic sentences programmed in natural language programming. Lastly, I compare the differences in time complexities among the general parser, CYK algorithm, the specialized single thread parser, and the specialized parser in parallel architecture.*

Keywords— *Parse, parallel processing, proposition logic.*

I. Introduction to Proposition Logic[1]

A proposition is a symbol that takes value of true or false. For example, let proposition P represents "John is here", it can only take one value, either P is true, i.e. John is here; or P is false, i.e. John is not here. Similarly, we can let proposition Q represents "Mary is here". These two representations are just the abstract conceptualization.

Next let's introduce the proof system. It is mainly by Modus Ponens Rule: if we know A, and we know A->B, then we can infer that B. Then we can start to prove in an axiomatic system. Assume we have axioms
(1) P or Q
(2) not Q
Please note that (P or Q) is equivalent to (not Q -> P). So we can have
(3) not Q -> P
By (2) and (3) and Modus Ponens Rule, we can deduce that
(4) P

Hence we have P, not Q. i.e. We have "John is here" and "Mary is not here". So just like algebra, we firstly transform natural language sentences to propositions containing only one segment of letter, then we do "reasoning" to get the results P and not Q.

This is a brief introduction to proposition logic. Next, I will give some example sentences of proposition logic, the context free grammar of proposition logic, and the parse trees of example sentences.

II. Context Free Grammar of Proposition Logic Sentences

Firstly, I give some example sentences of proposition logic in the following.

P
not P
not (P)
not not P
P or Q
(P or Q)
not (P or Q)
P or not Q
not P or Q
not P or not Q
P and (Q or R)
P and not (Q or R)
(P or Q) and (R or S)
not (P or Q) and (R or S)
(P or not Q) and not (not R or S)
((P imply Q) and R) or S
P and (Q and (R or S))
not (P or Q) and not (R or S)
not ((not P and Q) imply not R) or not S

These are all the example sentences of proposition logic. Next, I will give the context free grammar[2] of proposition logic.

sentence := (sentence)
sentence := not sentence
sentence := sentence binaryOperator sentence
sentence := variable

binaryOperator := and
binaryOperator := or
binaryOperator := imply

variable := P
variable := Q
variable := R
variable := S

So by inspecting on the context free grammar, we can see that there are 3 main structures: parenthesis, not, and binary operator. A specialized single thread parser will be

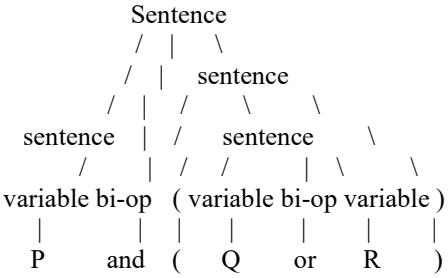
Weihan Huang, "A Parallel Architecture for Parsing Proposition Logic," *International Research Journal of Advanced Engineering and Science*, Volume 11, Issue 2, pp. 1-5, 2026.

International Research Journal of Advanced Engineering and Science
ISSN (Online): 2455-9024

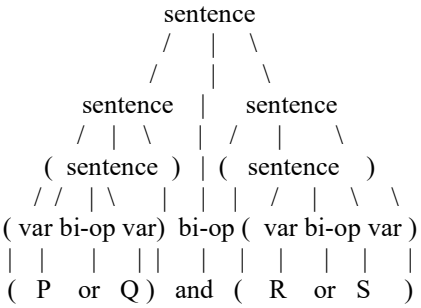
illustrated in the next section which takes advantage of these 3 main structures.

Next, I give 2 examples of the parse tress[3] of proposition logic sentences.

The first example is P and (Q or R)



The other example of parse tree is shown below for sentence (P or Q) and (R or S).



III. Single Therad Version Of Parsing Proposition Logic

To compare my parallel architecture with sequential algorithm, here I will design a single thread version of parsing proposition logic sentences. This algorithm is specialized for proposition logic sentences only.

This algorithm will parse 3 times by linear scanning the whole sentence which is tokenized ahead. The first time of parse will parse the parenthesis structure of the input sentence tokens. The second time of parse will parse the not structure of the last result sentence tokens. And the third time of parse will parse the binary operator (and, or, imply) structure of the last result sentence tokens. Because these 3 parse algorithms are pretty similar, I only show the first parse of the parenthesis structure in the following.

I will use natural language programming[4] to show the main algorithm of the parse of parenthesis structure. To make the code clean, I insert "(" at the beginning of the token list, and insert the ")" in the end of the token list, and remove them after the parse.

```
[codebook]
Single thread parser;

[global variable]
input sentence : string;
current token list : token node list : {};
result list : list : {};

[function]
parse parenthesis structure {
 let $$$current token list$$$ be (tokenizes $$$input
     sentence $$$);
 add (new token node "(") to $$$current token list$$$ at
     position 1;
  add (new token node ")") to $$$current token list$$$;
  let $$$result list$$$ be (get list from 1);
}

get list from $number$ : return $list$ {
  let $list$ be {};
  add ($$$current token list$$$ $number$) to $list$;
  $number$ increases by 1;
  while($number$<=(length of $$$ current token list $$$)),
        do{
    new token node $token node$ is ($$$current token list
                $$$ $number$);
    if((string of $token node$)="("), then {
      new list $parenthesis$ is (get list from $number$);
      add $parenthesis$ to $list$;
     let $number$ be ((position of (<<token node>>
              ($parenthesis$ (length of $parenthesis$))))+1);
     $number$ increases by 1;
    } else if((string of $token node$)=")"), then {
       add $token node$ to $list$;
       return $list$;
    } else {
       add $token node$ to $list$;
       $number$ increases by 1;
    }
  }
}
```

Here I have used one advanced programming techniques "global variable" in natural language programming. A global variable is a variable declared in a codebook that can be evaluated and assigned in any other places of the codes. A global variable is syntactically enclosed by 3 $ such as $$$variable$$$. For more details, please see reference [5] for global variables. This algorithm calls recursively to itself when it encounters the sign "(", which is left parenthesis :

```
get list from $number$;
```

Therefore, it creates new parenthesis inside the list:

```
add $new list$ to $list$;
```

And it ends when meeting the sign")" which is right parenthesis.

Weihan Huang, "A Parallel Architecture for Parsing Proposition Logic," *International Research Journal of Advanced Engineering and Science*, Volume 11, Issue 2, pp. 1-5, 2026.

For other kinds of tokens, it simply adds into the list of the parenthesis.

```
add (current token) to $list$;
```

Since it is similar to parse the not-structure and the binary-operator structure. Here I just show the parenthesis parsing above. Next, I will introduce the parallel processing in natural language programming.

IV. Intorduction To Parallel Processing In Natural Language Programming

The syntax for the parallel programming is simple:

```
Parallel run { expression1;
               expression2;
               expression3;
               …… }
```

The expressions will parallel run with the main algorithm. And it will throw a variable #task id list# containing the task ids of each expression. We can also use (task id) function to know the current thread's task id.
The following is an example of parallel programming :

```
***** parallel run.enProgramming ****

parallel run {
  function one (function three(function three 21));
  function one (function three 1);
    function one (function three 2);
}

write a row ("task id list is "+#task id list#);

new number $count$ is 1;
infinite loop {
  $count$ increases by 1;
  If($count$=5), then { end program; }
  write a row "main program;
  stop for 0.2 seconds;
}

[function]
function one $number$ {
  infinite loop {
    write a row ("task"+(task id));
      stop for 0.1 seconds;
  }
}

function two $number$ {
  infinite loop {
    write a row ("task"+(task id));
      stop for 0.5 seconds;
  }
}

function three $number1$ : return
                $number2$ {
  return ($number1$+1);
}

%%%%%%%%  screen output %%%%%%

task3
task4
task2
task id list is {2,3,4}
main program
task3
task2
task3
task2
main program
task3
task2
task3
task2
task4
task3
task2
task3
task2

%%%%%%%%%%%%%%%%%%%%%%%%
```

This is a brief introduction to parallel programming in natural language programming. To study it more detailed, please see reference[6].

V. A Parallel Architecture For Parsing Proposition Logic Sentences

For each variable place P in the sentence, it is associated with an object of class "logic element". In short, we will use "P's logic element" to represent the object. The class of "logic element" is defined in the following:

```
[class]
logic element;

[data]
operator : string : null;
left element : logic element : null;
right element : logic element : null;

begin position : number : null;
end position : number : null;

variable : string : null;
variable position : number null;
```

```
right parenthesis reached : boolean;
left parenthesis reached : boolean;

locked : boolean : false;
active : boolean : true;
touched history : number list list : {};
thread number : number : null;
```

Each logic element will spread to their neighbors in the order firstly right neighbor token node in the sentence will be merged, and then left neighbor token node in the sentence will be merged. The spreading of logic element runs parallelly for each variable place P and its associated logic element. During the spreading, two logic elements could meet and then be merged into one logic element. And finally, only one logic element will exist, and it is the final answer we want. So, this is the parallel architecture of parsing proposition logic sentences.
The code of this parallel algorithm is shown below.

```
[function]
run $logic element$ {
    if(($logic element$ is active) and ($logic element$ is not
            locked)), then {
        lock $logic element$;
        spread $logic element$;
    }
}

parse $string$ : return $logic element$ {
  initialize sentence and logic element list;
  while(true), do {
    for each $logic element$ in
          $$$logic element list$$$, do {
            if((begin position of $logic element$=1) and
              ((end position of $logic element$)=(length of
                  $$$sentence$$$))), then {
              return $logic element$;
        } else {
              parallel run {
                  run $logic element$;
              }
      }
  }
}
```

There are 2 places need to be taken care of. They are the program structures of “parallel run” and “lock”.

```
Firstly, for “parallel run”, it is
  parallel run {
    run $logic element$;
  }
```

Because it is inside the loop of for each logic element in logic element list, so for each element it will be run parallelly with the main program and the run logic element threads. This is the essence of parallel architecture.

The second one to address is “lock”. In the beginning, every logic element is locked, so if the main thread iteration tries to run the logic element again, it will not work because the logic element is locked. The locks will be unlocked after the logic element is merged.
This is the end of the section of parallel architecture of parsing proposition logic sentences. Next and lastly, I will compare the time complexities of different parsing algorithms.

VI. Comparisons of Time Complxities of Different Parsing Algorithms

There are 3 algorithms I will compare the time complexities. The first one is the general grammar parser CYK algorithm [7]. The second one is the specialized single thread algorithm mentioned in section III. And the third one is the specialized parallel algorithm mentioned in section V.

The word “general” for CYK algorithm instead of “specialized” is in that the CYK algorithm works in all kinds of input with arbitrary context free grammar. So, it is a general algorithm. And the specialized algorithms in section III and section V can only work for the given context free grammar in section II of the proposition logic sentences.

Let N be the length of the input sentence, i.e. the number of tokens of the input sentence.

The CYK algorithm[7] uses dynamic programming to parse the input sentence. The time complexity of CYK algorithm is O(N^3 * |G|) where N is the length of input sentence and |G| is the number of grammar rules.

The time complexity of the specialized single thread parser is O(3N), because it will parse the sentence 3 times: the first time for (), the second time for the not, and the third time for the binary operator. And each time it takes O(N) time complexity.

Lastly, the time complexity of the specialized parallel algorithm is (N*M/(number of variable places)), where M is the length of the data attributes of a logic element. For each token in the sentence, it will update the data attributes in a logic element once, and there are N tokens in the sentence, so there are totally N*M updates for all data attributes. Because the merges run in parallel by number of threads of each variable place, so the final complexity is O(N*M/(number of variable places)).

References

[1] Proposition logic
https://en.wikipedia.org/wiki/Propositional_calculus
[2] Context Free Grammar
https://en.wikipedia.org/wiki/Context-free_grammar
[3] Parse Tree
https://en.wikipedia.org/wiki/Parse_tree
[4] Natural Language Programming

Weihan Huang, “Natural Language Programming in English”, LAP LAMBERT Academic Publishing Book, 2022-08-30
[5] Global Variable
[4]pp142
[6] Parallel Programming
[4]pp157
[7] CYK algorithm
https://en.wikipedia.org/wiki/CYK_algorithm