INTRODUCTION

Many applications use parsers as a primary phase. The applications range from browsers to display web pages, search engines to find web pages, checkers in word processors, to compilers. For these parsers to work, the scanned tokens must be well formed and described by specific grammars. These grammars are always varying and may be non-deterministic. If the developer’s intention is to build a parser by hand, it would be a complex and costly task. Parser generators are used instead. Parsers generally fall into two types: bottom-up and top-down.

Existing parser generators such as YACC [1], Bison [2] and CUP [3] are types of LR parser generators but require the grammar to be LALR(1). They use a bottom-up approach and only handle deterministic grammars. To resolve non-determinism, GLR [4] is an extension to the LR parsers that launches a thread for each alternative. Each thread has its own stack and these stacks are combined into a graph structured stack (GSS); an example of GLR parser generators is Elkhound and an example of Elkhound-based C++ parser is Elsa [5]. The generated parser by Elkhound launches sub-parsers in pseudo-parallel to walk on all available paths. If one failed, it should die. Surviving parsers result in a parse forest. Elkhound proved to have an efficient parsing time.

Non-determinism means a decision that cannot be handled with finite lookahead available to the parser. Non-deterministic LR grammars can also be parsed with backtracking. [6, 7, 8, 9]. The backtracking approach suggests that when finding a non-deterministic rule, then a trial parsing is done to the first alternative. If the first alternative failed then backtrack to try another one. These steps are repeated until either all alternatives fail or any of them succeeds.

Regarding top-down parsers, which are used to parse type of grammars called LL, a set of parser generators are used such as Coco/R [10], ANTLR 3 [11], ANTLR 4 [12], and JavaCC [13]. ANTLR 3 can parse LL(*) by analyzing and building a DFA from the specified grammar statically. If it fails to build a DFA, it asks the user to specify the option {backtrack=true} explicitly to direct the generated parser to work with backtracking. ANTLR 4 generates ALL(*) parser that predicts which correct rule to expand. The prediction is done by building an augmented transition network (ATN) dynamically with the use of graph structured stack (GSS). The result of the prediction can be used to select the rule to expand. JavaCC doesn’t resolve the non-determinism problem but requires the programmer to specify the k lookahead explicitly. It doesn’t use backtracking but just look at the k symbol to select the path that can be visited. GLL[14] uses backtracking by building GSS nodes and attempting to parse non-deterministic alternatives in order.

Parsing expression grammars (PEGs) [15] resolves ambiguity by using a prioritized operator “/” to order the rules to be visited. PEGs also provides some predicates for false and successful parsing. Ford had
implemented parsers for PEGs like packrat and its generator Pappy [16, 17] that is written for Haskell. It memorizes all intermediate results to ensure linear-time performance. Rats! and Mouse [18, 19] are other implementations and are written for java. Mouse uses a very simple recursive descent parser with backtracking but doesn’t store intermediate results; this makes the parser inefficient.

This paper introduces a TGULL (Threaded General LL) parsing method. It handles all context free grammars. It is a recursive descent parser that allocates a thread for each non-deterministic alternative to parse it. Left recursion is converted to right recursion automatically by the generator. The TGULL parser is simple and can be built by hand, but for lengthy grammars it is preferred to use the generator.

2. RECURSIVE DESCENT PARSING

Formally, a context free grammar (CFG), G, consists of a set of N non-terminals, a set of T terminals, a set of symbols V ∈ (T U N), an element S ∈ N is denoted as start symbol and a set of grammar rules P in the form A → α where A ∈ N and α ∈ (T U N)*. The symbol ε denotes the empty string. A rule is in a form A → α₁ | ... | αₙ where αᵢ an alternative string of A.

A derivation step is the process of replacing a non-terminal symbol with one of the alternatives. The symbol ⇒ denotes the derivation symbol and the expansion γAβ ⇒ γAβ where γ, β ∈ (T U N)* and A → α is a grammar rule. A derivation sequence is the set of derivation steps that are used to derive a string. A string u can be derived from v by a sequence of derivation steps, v ⇒ β₁ ⇒ β₂ ⇒...⇒ βₙ ⇒ u. If n ≥ 0, the derivation can be written as v ⇒ u or v ⇒ u in case of n = 0. The symbol ⇒ denotes “doesn’t derive” symbol, which is the opposite symbol to ⇒. For example u ⇒ v means u can’t derive v directly or indirectly through a derivation sequence.

The language L(G) is the set of all strings u, where u ∈ T, that can be derived from the start symbol, S. Parsing is the process of finding the derivation steps from S to u, S ⇒ u.

We define the first and following sets that are used in many parsing techniques as follows

First(A) = {t ∈ (T U ε) for some string α (A ⇒ tα or A ⇒ ε)}

First(αγβ) = First(αγ) if x ≠ ε
First(αγ) if x = ε

The follow set is defined as

Follow(A) = {t ∈ (T U ε) for some string β, α (S ⇒ βAα
S ⇒ ε or βAα
S ⇒ αA)}

A non-terminal A is said to be LL(1) if (I)A → a|β implies that First(α) ∩ First(β) = φ, and (II) if A ⇒ ε then First(A) ∩ Follow(A) = φ. A grammar is said to be LL(1) if ∀A ∈ N|A is LL(1).

A non-terminal A is said to be left-recursive if for some string τ a derivation sequence leads to A ⇒ Aτ.

A deterministic recursive descent parser (DRD) is built from a grammar as follows. For each non-terminal A with a rule A → α₁ | ... | αₙ, a parse function Pₐ is created. The parser is composed of a collection of parse functions. The Pₐ function is the first function to be called where the start symbol S is firstly expanded. The selection of which alternative αᵢ is executed is based on the current parsed input token t, that is t ∈ First(αᵢ). If t ∈ First(αᵢ) then the parse function returns a failure and reports an error. The input string is totally accepted upon reaching the end of input S token and at the same time no more rules to be expanded.

Two pitfalls cause the recursive descent (RD) to fail. The first occurs when the alternative can’t be decided from the current input token. The second occurs when there is a rule with left recursion that can be either explicit (A ⇒ Aγ), or implicit (A ⇒ Aγ). A hidden left recursion is a special type of the implicit (A ⇒ βAγ and β ⇒ ε). Left recursion causes the execution to go to infinite loop. The former obstacle can be handled by approaches such as left-factoring, investigating the k lookahead token LL(k) (see next paragraph) or backtracking. The later obstacle can be handled by converting left-recursion to right recursion.

Informally a grammar G is said to be LL(k), where k is a positive integer number k>0, if a set of k inputs must be checked before selecting which alternative to expand. If k is unlimited then the grammar is said to be LL(*) or non-deterministic grammar. [20] has a formal definition for LL(k). This paper intended to handle non-deterministic grammars by applying multi-threading technology.

3. HANDLING NON-DETERMINISM WITH THREADS

We are going to describe the idea by a simple grammar G₂₁.

S → A c a | B c b | d
A → a A | ε
B → a B | ε

The grammar G₂₁ is not LL (1) and is a non-deterministic grammar as it contains two alternatives for the production S with unlimited look ahead. With traditional recursive descent parsing, a parse function is created for each non-terminal rule. Three parse functions are created Pₛ, Pₐ, Pₐ. The main function is the entry point
where the parsing starts working. Two global variables are used, \( t \) and \( i \). Variable \( t \) refers to the current token whose location is marked by the variable \( i \). Two auxiliary methods are used, eat and error; the error is used to report a parse error and terminates the algorithm. The eat method advances the marker by 1 in case of a successful match, otherwise it reports an error.

Function main \{ \ i = 0; \ P_S \ ; \}
Function eat (symbol) \{ if(t = symbol) \ i = i + 1 ; \ t = tokens[i]; \} else error() \}

Function \( P_S \ ; \} \{ \ i \in \text{First}(Aca) \} \{ \ P_A \ ; \} \}
eat(a); else if \( t \in \text{First}(Bcb) \} \{ \ P_B \ ; \} \}
eat(c); eat(c); eat(b); else if \( t \in \text{Follow}(A) \} \{ \text{eat(d); else error()}} \}

Function \( P_A \ ; \} \{ \text{if} \ t \in \text{First}(aa) \} \{ \text{eat(a);P_A} \ ; \}
eat(a); else if \( t \in \text{Follow}(B) \} \{ \text{eat(d); else error()}} \}

Clearly if the input is “aaacb”, the parser would generate an error. The reason is that the first input symbol \( a \in \text{First}(Aca) \cap \text{First}(Bcb) \) and the flow of the execution would select the first alternative, which would report an error. To deal with such error we have created a sub-parser that is an instance of the whole parser but starts parsing from location \( j \) on non-terminal \( A \) where the non-determinism occurs. We used a parser configuration to record information about each parser. Each alternative is expanded and executed in a separate thread so that all threads are executing independently and in parallel. Some threads may fail, others may succeed or all fail. If all threads failed to parse the remaining parts then a general parse error is reported and if some succeeded the result obtained from the first succeeded thread is selected. One reason could cause the thread to fail is when consuming all its inputs without completing matching its rule.

A parser configuration can be defined as a 4-tuple \( \text{conf}(i,A_p,s,r) \) where \( i \) is the start position to parse from, using Rule \( A \) over alternative \( a, s \) is the parser state that can take values (init, start and finish) and \( r \) is the result obtained from the parser. We define a function create which is used to create a parser configuration from the above given parameter. The set \( \mathcal{L} \) is used to hold all newly created parsers configurations. When reaching a point of non-determinism, configurations are created and added to the set \( \mathcal{L} \). We assume the function add \( (C_i,\mathcal{L}) \) is used to add a parser configuration to the set \( \mathcal{L} \). If the result of \( \text{sub}_p \) succeeds, the main parser may continue from location \( j+1 \). We define three additional utility functions start \( (\mathcal{L}), \text{wait(\mathcal{L}) and success(\mathcal{L))} \). The first causes children parsers with predefined configuration to start on separate independent working threads. The second causes the current parent to wait for its children to change their states to finish. The third retrieves the configuration of the first successful sub-parser.

Function main \{ \ i = 0; \ P_S \ ; \}
Function eat (symbol) \{ if(t = symbol) \ i = i + 1 ; \ t = tokens[i]; \} else error() \}

Function \( P_S \ ; \} \{ \ i \in \text{First}(Aca) \cap \text{First}(Bcb) \} \{ \ P_A \ ; \}
Add \( (C_1,i,P_S;’init’,R_1,\mathcal{L}) \)
Add \( (C_2,i,P_S;’init’,R_2,\mathcal{L}) \)
start(\mathcal{L})
wait(\mathcal{L})
\( i \in j+1; \) return \}
error() \}

Function \( P_A \ ; \} \{ \text{if} \ t \in \text{First}(aa) \} \{ \text{eat(a);P_A} \ ; \}
Function \( P_B \ ; \} \{ \text{if} \ t \in \text{Follow}(B) \} \{ \text{eat(d); else error()}} \}

Function \( P_\text{S}_1 \) \{ \ P_A ; \}
eat(c); eat(a); \}
Function \( P_\text{S}_2 \) \{ \ P_B ; \}
eat(c); eat(b); \}
Function \( P_\text{P} \) \{ \text{if} \ t \in \text{First}(aa) \} \{ \text{eat(a);P_A} ; \}
eat(a); \}
Function \( P_\text{P} \) \{ \text{if} \ t \in \text{Follow}(B) \} \{ \text{eat(d); else error()}} \}

The above algorithm that uses threading technology to resolve non-determinism, preserves the same nature of RD. Using threading to parse non-deterministic grammar may cause explosive number of threads to be created. This may cause the execution time to be degraded to the lowest level of performance.

To compute the total number of active threads, and for simplicity of computations we may consider the case in which rules are directly non-deterministic, in other words rules of many alternatives that have the common prefix and has recursion. If a rule \( A \rightarrow t A \beta_1 | ... | t A \beta_n | \epsilon \) where \( t \in \mathcal{T} \), and an input string that has prefix of \( t \) symbols of length \( l \), this would generate a tree like Fig 1:

![Figure 1: Tree generated from parsing the rule](image)
Each node in the tree corresponds to a thread and each thread can generate children threads. A parent thread must wait until all its children finish parsing. The more the number of input tokens, the more the tree depth. Each level 1 in the tree contains $n^1$ nodes where 1 is the level number. Summing up all nodes in all levels results in the total number of threads $N = \sum_{i=0}^{k} n^i = (n^{i+1} - 1)/(n - 1)$ equals to the number of active nodes (leaf nodes) $n^1$ in addition to the number of inactive (internal nodes) threads $(n^i - 1)/(n - 1)$. The number of threads grows exponentially in relation to the number of input tokens.

The threshold is used to limit the number of created threads. Two of the threads are rejected and there is a procedure for each non-terminal. The backtracking in TGLL is similar to TDPL [20]. We created a procedure for each non-terminal $A$. With $A \rightarrow \alpha_1 | ... | \alpha_n$ rule, the procedure takes as input the position of the token where to start parsing and returns either success or fail. If the procedure succeeded to match the alternative $\alpha_i$ with inputs $\alpha_i, \alpha_{i+1} ... \alpha_k$, then it returns success to the caller, which in turn continues parsing from the token position $k+1$. On the contrary, if the procedure failed to parse the alternative $\alpha_i$, then it tries $\alpha_{j+1}$ until one succeeds or all fail. If all failed, the procedure returns fail to the caller which may try one of its other alternatives.

Augmenting threading with backtracking is not complex. A variable threshold is used to specify the maximum number of threads to be created. From the comparison of the threshold value with the size of the set $\mathcal{L}$ we can detect either to create new thread or to allow the current thread to work with backtrack. The program for grammar $G_1$ becomes:

Function main () { i = 0; if(! $P_S$ ()) error();}

Eat (symbol){if(t = symbol) { i = i + 1 ; t = tokens[i]; return true;} else return false;}

Function $P_S$ () { if( $t \in \text{First}(\text{Aca}) \cap \text{First}(\text{Bcb})$ ) { if( size($\mathcal{L}$) less than threshold) {
  Add (C1) = conf (i, $P_S$; 'init', R1), $\mathcal{L}$
  Add (C2) = conf (i, $P_S$; 'init', R2), $\mathcal{L}$
  start($\mathcal{L}$)
  wait($\mathcal{L}$)
  $C_1 = \text{success($\mathcal{L}$)}$
  if($C_1 > \text{null}$) { i = j+1; return }
  error();}
else{
  Boolean result = $P_S$();
  if(result == false) result = $P_S$();
  return result;}} if ( $t \in \text{First}(\text{Aca}) - \text{First}(\text{Bcb})$ ) {if($P_S$()) return false;else if($t \in \text{First}(\text{Bcb}) - \text{First}(\text{Aca})$) {if($P_S$()) return false;else if($t \in \text{First}(\text{d})$) {
    if(!eat(d)) return false;
  } else return false;}
else return true;}

Function $P_S$1(){
if(!$P_A$()) return false;
if(! eat(c)) return false;
if(! eat(a)) return false;
return true; }

Function $P_S$2(){
if(!$P_B$()) return false;
if(! eat(c)) return false;
if(! eat(b)) return false;
return true;}

Function $P_A$() {
if($t \in \text{First}(\text{aA})$) {if(!eat(a)) return false; if(!$P_A$()) return false;}
else if($t \in \text{Follow}(\text{A})$) return true; else return false; return true;}

Function $P_B$() {
if($t \in \text{First}(\text{aB})$) {
5. FORMAL DEFINITION OF TGLL ALGORITHM

Let the rule of the non-terminal A defined as \( A \rightarrow a_1 \ldots a_n \). The sets \( A_1, A_2, \ldots, A_m \) contain some alternatives such that if \( a_i \in A_j \) for some \( 1 \leq i \leq n \) and some \( 1 \leq j \leq m \) then \( \bigcap_{x_i \in A_j} \text{First}(a_i) \neq \emptyset \). We say that the non-terminal A is deterministic or \( A \in LL(1) \) if all \( |A_j| = 1 \) and \( A \in LL(1) \) if there exists some \( A_j \) such that \( |A_j| > 1 \). The set \( A_j \) is null able if there exists \( a_i \in A_j \) such that \( a_i = \epsilon \). The non-terminal A is null able if there is at least one \( A_j \) null able.

We define \( \text{First}_a(A_j) \) set to be all the terminals that appear as the first symbol to all alternatives that belong to \( A_j \) or formally \( \text{First}_a(A_j) = \bigcap_{x_i \in A_j} \text{First}(a_i) \). It is clear that \( \bigcap_{x_i \in A_j} \text{First}(a_i) = \emptyset \). An alternative \( a_i \) may appear in more than one set.

The sets \( A_1, A_2, \ldots, A_m \) are required to be sorted in topological order, that is \( A_1 \) precedes \( A_2 \) and can be written as \( A_1 < A_2 \) if \( \text{First}_a(A_1) \cap \text{First}_a(A_2) = \emptyset \) and \( |\text{First}_a(A_1)| < |\text{First}_a(A_2)| \).

To illustrate the requirement that the sets must be in topological order we use the following grammar \( G_3 \):

\[
S \rightarrow ABc \mid abc \mid abe \mid bb \\
A \rightarrow a[a|b]|e \\
B \rightarrow b|e
\]

The symbol a appears as first symbol in alternatives \( A_1 = \{ ABc, abc, abe \} \), b appears as first symbol in alternatives \( A_2 = \{ ABc, bb \} \) and finally c appears as first symbol in alternative \( A_3 = \{ ABc \} \). Since \( \text{First}_a(A_1) \cap \text{First}_a(A_2) = \{ a \} \) \( \cap \{ b \} = \emptyset \), then they can appear in any order; since \( \text{First}_a(A_1) \cap \text{First}_a(A_3) = \{ a \} \cap \{ a, b, c \} = \emptyset \) and \( |\text{First}_a(A_1)| < |\text{First}_a(A_3)| \) then \( A_1 < A_2 < A_3 \).

The correct topological order of the sets \( A_1, A_2, \text{and } A_3 \) are \( A_1 \ll A_2 \ll A_3 \) or \( A_2 \ll A_1 \ll A_3 \). If the topological order was \( A_1 \ll A_2 \ll A_3 \) on input string \( bb \), then the parser algorithm would first check if \( b \in \text{First}_a(A_1) \) which fails; the check moves to \( \text{First}_a(A_2) \); this time the check succeeds. Inside \( A_2 \) we can find the alternative \( \{ ABc \} \) which fails to parse the input \( bb \), the algorithm then returns failure. If the order was \( A_1 \ll A_3 \ll A_2 \), the check would use \( A_2 \) alternatives to be parsed either in parallel or using backtracking. At this time, the parsing would succeed on the alternative \( bb \) and the algorithm returns true.

5.1 Defining Utility Methods

The algorithm uses a utility method called \( \text{eat()} \) which attempts to consume an input symbol and it returns true if it succeeds otherwise it returns false.

\[
\text{eat}(x,j) \{ \text{if}(x = t) \{ j = j + 1 ; t = \text{tokens}[j]; \text{return true;} \} \text{else return false;} \}
\]

5.2 Parsing an Alternative

An item may be either terminal or a non-terminal. We define the method \( \text{Parse}_i(x) \) that is used to parse an item. For a terminal \( x \),

\[
\text{Parse}_i(x,j) = \{ \text{if}(\text{eat}(x,j)) \{ \text{return true} \} \text{else return false} \}
\]

For a non-terminal \( X \),

\[
\text{Parse}_i(X,j) = \{ \text{if}(\text{Parse}_i(X_j)) \{ \text{return true} \} \text{else return false} \}
\]

(The Parse is a general method that calls the procedure associated for the non-terminal)

An alternative \( a_k \) for a rule \( A \rightarrow a_1 \ldots a_n \) is composed of a sequence of terminals and non-terminals \( x_1, x_2, \ldots, x_n \) where \( x_i, 1 \leq i \leq n \) may be a terminal or non-terminal. We define for each \( a_k \) a method \( \text{Parse}_a(A \rightarrow a_k, j) \) as follows:

\[
\text{Parse}_a(A \rightarrow a_k, j) = \{ \text{if}(\text{Parse}_i(x_j)) \{ \text{return false} \} \text{else if}(\text{Parse}_i(x_m)) \{ \text{return false} \} \text{return true} \}
\]

5.3 Parsing a Group of Alternatives

A group \( A_k \) may contain more than one alternative \( \beta_1, \ldots, \beta_f \) provided that the intersection of their first symbols is not empty. The group \( A_k \) has only one element if \( f = 1 \). A group of alternatives can be parsed either parallel or with backtracking. The selection is dependent upon a threshold variable. The value of the threshold can be set by the programmer or set to the number of processors to allow maximum parallelism.

5.3.1 Parsing Parallel Alternatives (Group)

We define \( \text{Parse}_p(A_k, j) \) for each \( A_k \) to parse the group of alternatives in parallel as follows:

\[
\text{Parse}_p(A_k, j) = \text{Add}(\text{conf}(j, \beta_1, \text{‘init’, R}), L) \\
\ldots \\
\text{Add}(\text{conf}(j, \beta_f, \text{‘init’, R}), L)
\]

\[
\text{start}(L) \\
\text{wait}(L) \\
C_1^j = \text{success}(L) \\
\text{if}(C_1^j = \text{null}) \{ j = j + 1 ; \text{return true;} \} \text{else return false;}
\]

The start() and wait() are utility method which can be implemented using any programming language library that support thread programming. The success
method can be easily written as it iterates over all sub-parsers and retrieve their results the first one with result ‘success’ is retrieved otherwise null is returned.

5.3.2 Parsing Backtrack Alternatives (Group)

We define Parse\(_g\)(\(A_k, j\)) for each \(A_k\) to parse the group of alternatives with backtracking as follows:

\[
\text{Parse}_g(A_k, j) = \\
\text{if}(\text{Parse}_g(A_β, j))\{\text{return true}\} \\
\text{else if}(\text{Parse}_g(A_β, j))\{\text{return true}\} \\
\text{return false}
\]

We define Parse\(_g\)(\(A_k, j\)) for each \(A_k\) to parse a group of alternatives as follows:

\[
\text{Parse}_g(A_k, j) = \\
\left\{ \\
\begin{array}{l}
\text{if}(|A_k| = 1) \\
\text{if}(|A_k| > 1 \text{ and } \text{threads} < \text{threshold}) \\
\text{if}(|A_k| > 1 \text{ and } \text{threads} \geq \text{threshold}) \\
\end{array}
\right.
\]

5.4 Parsing a Rule

Assume a grammar rule is \(A \rightarrow a_1 \ldots a_m\). The alternatives that has intersections for their first set symbols are grouped into groups \(A_1, A_2, \ldots, A_m\). We define Parse\(_N\)(\(A, j\)) as follows:

\[
\text{Parse}_N(A, j) = \\
\text{if}(t \in A_1)\{\text{return Parse}_N(A_1, j)\} \\
\text{else if}(t \in A_m)\{\text{return Parse}_N(A_m, j)\} \\
\text{if}(A \text{ is nullable and } t \in \text{Follow}(A))\{\text{return true}\} \\
\text{return false}
\]

6. MEMORIZING INTERMEDIATE RESULTS

By storing intermediate results, the parsing time with backtracking can be reduced from exponential time to polynomial time for general parsing and in case of PEG it will be reduced to linear time [16, 17]. Clearly storing information avoids re-parsing the same part more than once. The same concept can be used with the multi-thread model. Storing intermediate results can reduce the parsing time by numbering rules and storing parse results in a table. The table has two entries. The first entry is the parsed token number and the second entry is the rule number. The table uses hashing to map entries which takes time efficiency \(O(1)\) for the first entry and also for the second entry. Each node stored in the table stores a number to indicate the end token position of parsing.

For example fig 2 shows an example of a table data structure constructed by parsing the input string “\(x=1*2*3;\)” using grammar \(G_4\).

\[
\begin{align*}
1) S & \rightarrow X \\
2) X & \rightarrow Q ; X \\
3) X & \rightarrow \epsilon \\
4) Q & \rightarrow id = E \\
5) E & \rightarrow T + E \\
6) E & \rightarrow T - E \\
7) E & \rightarrow T \\
8) T & \rightarrow F * T \\
9) T & \rightarrow F / T \\
10) T & \rightarrow F \\
11) F & \rightarrow id \\
12) F & \rightarrow num \\
13) F & \rightarrow (E)
\end{align*}
\]

The first entry 2 means that the thread started parsing a token in position number 2. From token position 2 to token position 2, it can be reduced by rule 12 or rule 10. From token position 2 to token position 6 can be reduced by rule 8.

![Figure 2: Construction of shared storage](image)
thread to write at a time, but it will increase the execution time.

7.2 Parallel Parsing with Fork/Join Model

Another way to apply multiprocessing efficiently is by using the Fork/Join framework. Java 7 has recently added a Fork/Join framework in its library [22]. The Fork/Join is a multi-threaded programming style that works with divide-and-conquer approach. It allows the problem to be divided into smaller sub-problems; each sub-problem can be solved by the same or different way from the main problem. The process of division continues until reaching the atom. The atom is the smallest problem that cannot be divided and must be solved directly.

The benefits of using the java Fork/Join framework is that it can manage tasks in the same way the operating system manages threads. The difference is that it manages tasks in a light weight manner but the operating system manages threads in a heavy weight manner. Threads are created only once and saved in a thread-pool area, thus avoiding thread allocation and re-allocation. Each main task and its sub-tasks that are held in a queue are scheduled to a thread. Threads can be created equal to the number of cores (by using java function Runtime.getRuntime().Available Processors()) or as the programmer specifies. Another benefit is that if a main task and all its sub-tasks are idle (e.g. waiting for some event to occur), the system can steal tasks from other threads. Since the cost of constructing a new thread is greater than the parsing time, the Fork/Join model has greatly enhanced the parsing time.

8. EXPERIMENTAL EVALUATION

In this section we show the time and memory measurements for our experiments. We show the impact of different implementations of the parser, the effect of applying multithreading and the multi-threaded Fork/Join model. The time is compared with ANTLR V4 and JavaCC [13]. We used syntactic look ahead with JavaCC to allow the resolution of non-determinism. The time measurements do not include semantic action execution, only the parse time is measured. Time is measured after the two executions so as to make sure that the Java Virtual Machine has its stable state. Moreover, the average of three consecutive measurements is recorded.

All the experiments are done on a machine with processor model of Intel® Core™ i5-2450M CPU @ 2.5GHz 2.5 GHz. Memory is 4 GB. The operating system used 64-bit Windows 7. Java Development Kit version jdk1.8.0_05 is used as a compilation environment. Memory is measured according to the equation:

\[
\text{Runtime runtime} = \text{Runtime.getRuntime()}; \\
\text{long memory} = \text{runtime.totalMemory()} - \text{runtime.freeMemory()};
\]

The garbage collector is called to make sure that all leak memories are freed for accurate memory measurements.

8.1 Experiments with Planned Test

Working on the grammar listed in figure 4 and varying the input size, we notice that only two rules have non-determinism, namely:

\[
\text{expr:Term\+Expr\Term\-Expr\Term; Term:Factor\ast\Term\//\Term\Factor;}
\]

8.1.1 Planned Test 1

Two types of input strings are planned to be tested to show the power of working with threading. The first type examines the depth in the first rule. For example \(x=1*2*3*....\cdots n;\) is a set of multiplication operators, which is the worst case analysis for parsers that work with backtracking as the first and the second alternatives of the first rule would fail and the third alternative would succeed. By testing this type of input, three threads are created, one for each alternative of the first rule, and all of the three alternatives remain alive until reaching the token by which a thread can either succeed or fail. For the second rule, the decision would be quick by eating a factor and scanning the next input token ‘+*’; the first thread keeps alive and the remaining alternatives die.

Fig 3 presents a graph of this planned test. The graph illustrates that working with raw threads (4 threads allocated as the test runs on 4 core processors) has the worst time analysis as it grows tremendously with small increase in the file size. The graph compares also the run time for multi-thread execution without storing intermediate results, which performs better than the case when storing intermediate results. Working with only one thread and working with the Fork/Join model is more efficient than ANTLR. JavaCC has the shortest time from all tested models. The tests were made on input file sizes up to 7000 bytes, as ANTLR V4 gave an exception message for the higher sizes even if after increasing the stack size with the option –Xss4m. JavaCC, Fork/Join and single-thread models seem to be coinciding due to their small time measurements.

\[\text{Exception in thread "main" java.lang.StackOverflowError at org.antlr.v4.runtime.atn.ATNState.getNumberOfTransitions(ATNState.java:178)\}]

Another zoomed version of the graph without the multi-threaded model is shown in Fig 4. The figure shows that working with only one thread and working with Fork/Join model is more efficient than working with ANTLR V4. JavaCC and one-threaded are very close, but JavaCC is more efficient.
Figure 3: Time Measurements comparison between Antlr v4, one-thread, multi-thread (4 threads), Fork/Join, JavaCC and multi-thread without storage (plan 1).

Figure 4: Time Measurements comparison between Antlr v4, one-thread, Fork/Join model and JavaCC (plan 1).

8.1.2 Memory Measurement of Planned Test 1

The memory comparison is shown in fig 5. The comparison shows that one-thread, Fork/Join and Javacc have the least memory consumption. Antler V4 requires extra storage to store the NFA of the ATN simulator. The 4-thread model with shared intermediate storage consumes high storage as each thread accesses the intermediate storage to store its information. The multi-thread model without sharing intermediate storage also consumes high memory as each thread constructed must have its own context that consumes an amount of memory.
Figure 5: Memory measurement comparison between Antlr v4, one-threaded, multi-threaded (4 threads), Fork/Join and JavaCC modes (plan 1).

8.1.3 Analysis of Planned Test 1
In our opinion, there are two reasons behind the inefficient performance of the multi-thread model. The first reason is the frequent allocation and de-allocation of threads which can be solved by using a thread pool. The thread pool allows the creation of threads and allocation of resources only once and re-using threads many times as needed. The thread allocation time can be more costly than parsing the part allocated to that thread. The second reason is in the shared data structure in case of shared memory for storing intermediate results. Storing intermediate results is supposed to reduce the parsing time. Since the shared data structure allows many threads to access it at the same time, so synchronization must be done to prevent concurrent write problems. Synchronizing the shared data structure causes one thread to acquire the lock while other threads to be blocked waiting for the lock.

Both the multi-threaded Fork/Join model and working with only single thread seem to have linear graphs with very small slope. The multi-threaded Fork/Join model costs some extra time to allocate threads which is a constant time. The Fork/Join model graph is less than ANTLR V4 as it allocates threads only once and also avoids the problem of locking threads on the shared data structure. We did not use intermediate storage to store intermediate results obtained from each thread in the Fork/Join model in order to avoid falling in the concurrency problem and also to reduce the memory usage. The worker stealer feature implemented in the Fork/Join model allows the idle thread to steal some tasks from other busy threads. This feature maximized the CPU utilization and the task throughput.

8.1.4 Planned Test 2
The other type of input string that is tested is the variation of depth in both rules, like “x=1*2*-----*n / 1*2*-----*n + 1*2*-----*n / 1*2*-----*n - 1*2*-----*n / 1*2*-----*n””. Fig 6 shows the comparison between ANTLR V4, one-thread model, multi-threaded model, multi-threaded Fork/Join and JavaCC. At first, it seems that using multi-threads is faster than ANTLR V4. But for the same reasons stated in the analysis of plan 1, the graph tends to grow very fast by small increase in the input file size. Multi-threaded Fork/Join model is more efficient than ANTLR V4. A more detailed graph without multi-threaded model is shown in fig 7. The Fork/Join model and the one-threaded model are more efficient than ANTLR for the same reasons stated in the analysis of plan 1. JavaCC is the most efficient one.
Figure 6: Time Measurements comparison with multi-threaded (plan 2)

Figure 7: Time Measurements comparison without multi-threaded (plan 2)

Memory measurement of plan 2 is shown in Fig 8. JavaCC, Fork/Join and one-threaded models seem to be coinciding due to small values. The one-threaded model has greater memory as it needs to store intermediate values but the Fork/Join model has less memory as it doesn’t need to store intermediate values.
8.2 Experiments with java Grammars

In this section, we test the parser generated from the Fork/Join model with threshold 4 on java source files. 15387 source files with size 141 MB from the JDK 1.7 are used in the experiment. The grammar was taken from Antler V4 site (https://github.com/antlr/grammars-v4). Antlr provides two versions of java grammar and we had selected the fastest one to test with it. For fair comparison as JLex is a slow lexer tool we, replaced the JLex with Antler lexer in this experiment. The experiments show that the parser with fork/join has the fastest execution time, and the smallest memory.

Fig 9 shows that the Fork/Join without storage has time less than JavaCC and Antler V4 by about 16.64% and 48.25% in sequence. The Fork/Join without storage has less memory consumption than JavaCC and Antler V4 by 15.21% and 87.89% in sequence. The reason for time reduction is that Antler V4 predicts which rule to visit next by creating an adaptive network and this consumes time as tokens have to be visited twice once for the prediction and another time for the real visit of the rule. In JavaCC also tokens are visited twice as the parser has to look at the k token which can decide the correct rule to execute.

Our parser visits the tokens only one time as when ambiguity occurs a separate task on a thread is allocated for each path and tasks are working in parallel. Parsers that work with backtrack fall into the problem of executing correct actions, they always execute incorrect actions or duplicate action execution. Our parser delays the action execution until all rules are completely checked. The space requirements for our parser requires O(n) in the worst case; if each token is associated with an action it will be stored in a queue to be executed at later time.
9. CONCLUSION AND FUTURE WORK

The paper proposes a recognizer for context free grammars that resolves non-determinism by launching multiple threads to parse the different alternative rules. The threads are augmented with backtracking to avoid explosive thread creation. A practical parser generator is done for it. The generator can generate a recursive descent Fork/Join model. The analysis shows that working with the multi-threaded Fork/Join model can be faster than famous existing parser generators such as JavaCC and Antler and even with smaller memory. Creating a parser that executes with a single thread and backtracking has approximately the same parsing time as the Fork/Join model but due to the thread allocation constant time, the Fork/Join model takes a small extra constant time than the one-threaded model. However, the Fork/Join model consumes less memory than the single-threaded model. We are seeking for more parsing improvements that can benefit from the parallelism and multi-core technologies which became available to all users.

REFERENCES


