Parallel Execution of Tasks of Lexical Analyzer using OpenMP on Multi-core Machine

Rohitkumar Rudrappa Wagdarikar* and Rajeshwari Krishnahari Gundla**
*-**Computer Science and Engineering Dept, Walchand Institute of Technology,Solapur, Maharashtra,India
Email: rohit.wagdarikar@yahoo.com, gundlaradhika9@gmail.com

Abstract: Compiler is a system program that translates a source language into target language. The structure of a Compiler is composed of several phases. The first phase is lexical analysis or scanning. This is the only phase which interacts with original source code written by the programmer and perform the various tasks which includes, read the input characters and produce as output sequence of tokens that the parser uses for syntax analysis, and make entry to symbol table, Stripping out from the source program comments and white space in the form of blank, tab, new line characters. Another task is correlating error messages from the compiler with the source program and keeping the track of error by line number. In this paper we present concurrent execution of tasks of lexical analyzer. In a multi-core system, task parallelism is achieved when each core executes a different thread on the same or different data. The threads may execute the same or different code. In this paper we present an approach that lexical analyzer tasks are independently allocated to each core to perform parallelism which improves the performance of the lexical analyzer.

Keywords: Concurrent Lexical Analysis, Multi-core Machine, Lexeme-Handler, Whitespace-Handler, Comment-Handler, Newline-Handler, OpenMP.

Introduction

Compiler is a system program that translates a high level source language into low level target language. The structure of a Compiler is composed of several phases. These phases are divided into two parts: Front End and Back End. Front End contains Lexical analyzer, Syntax Analyzer, Semantic Analyzer and Intermediate Code Generator. Back End contains Code Optimizer and Code Generator. As shown in figure 1, First phase of the compiler is lexical analyzer which generates the token from source program. Second phase of compiler is syntax analyzer, Ref. [8] it analyze the code against the production rules specified by context free grammar to detect the error in the code. Third phase of compiler is semantic analyzer Ref. [8] which helps to interpret symbols, their type and their relation with each other. It also verifies the meaning of statement. Fourth phase of compiler is intermediate code generation which generates the intermediate code in the form of three address code, syntax tree and postfix notation. Fifth phase of compiler is code optimizer which performs the optimization on intermediate code to decrease space and time complexity. Last phase of compiler is code generation which generates the target code in the form of low level language (e.g. assembly language). The first phase is lexical analysis or scanning. This is the only phase which interacts with original source code written by the programmer and perform the various tasks which includes, read the input characters and produce as output sequence of tokens that the parser uses for syntax analysis, and make entry to symbol table, Stripping out from the source program comments and white space in the form of blank, tab, new line characters. Another task is correlating error messages from the compiler with the source program and keeping the track of error by line number Ref. [2].

It should be noted that our proposed architecture address the problem of sequential execution of different tasks of lexical analyzer, so proposed architecture concentrate on defining the architecture whose performs the parallel execution of different tasks of lexical analyzer. The reminder of this paper is outlined as follows. After discussion of existing system, we present the architecture of our parallel approach and parallel algorithm for the lexical analyzer. In section 3 we conclude with outlook of future work.

Literature Studies

As described by Amit Barve, Brijendra Kumar Joshi in Ref. [10], source code is divided into the total number of blocks by total number of CPU, then identify the pivot location. As describe the these authors there will be three different pivot locations, 1.Based on Construct 2. Based on White Space Characters and last one is 3.Based on lines. In the first one, they identify the start of iteration syntax like for loop, while loop etc. in second one it will identify the total number of white spaces from source program and make the blocks and give it to each CPU, and in third one it will identify the total number of lines from source program and make the blocks and assign it to each CPU.

As shown in figure 2 lexical analyzer takes the input source program into the input buffer and uses Lexeme Beginning Pointer (LBP) and Lexeme Forward Pointer (LFP) to find the lexeme. Initially LBP and LFP are pointed to start of the input buffer, only LFP moves forward and while moving forward it checks for pattern and it uses longest matching rule Ref. [2]. If the pattern matches with lexeme, it extracts the set of characters between LBP and LFP. Based on lexeme found, Lexical analyzer performs following tasks: 1.If extracted lexeme is a token then it generates <token id, value> and if token is identifier or constant then it makes entry to symbol table. 2. If extracted lexeme is comment it removes comment lines and checks for next lexeme. 3. If it finds whitespace then it removes it and checks for next lexeme. 4. If it finds newline, it keeps track of line number which is used by error handler.

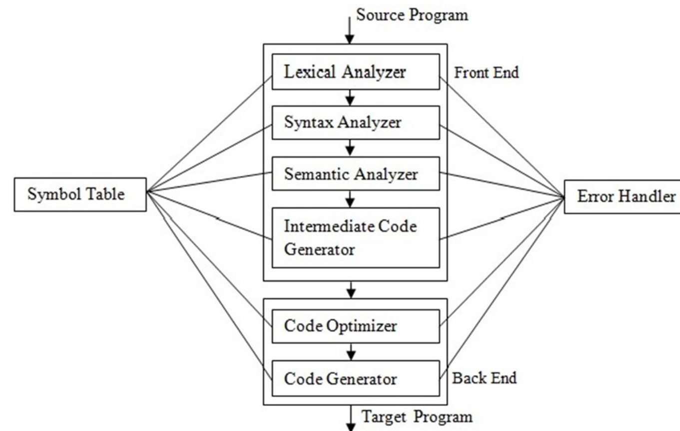


Figure1. Structure of Compiler

As described by Amit Barve, Brijendra Kumar Joshi in Ref. [9], multiple files can be taken as input and these files are placed in a single directory and each file is processing on different core on multi-core system to perform the tasks of lexical analyzer, by using this methodology they are performing parallelism in lexical analyzer.

In this paper we present concurrent execution of tasks of lexical analyzer by using OpenMP. OpenMP uses multi-core system. In a multi-core system Ref. [3], task parallelism is achieved when each core executes a different thread on the same or different data. The threads may execute the same or different code. OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications. OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. In OpenMP each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

Proposed System

As shown in figure 3 lexical analyzer takes the input source program into the input buffer and uses Lexeme Beginning Pointer (LBPtr) and Lexeme Forward Pointer (LFPtr) to find the lexeme from source program, which is assigned to core 1. White Space Pointer (WSPtr) is used to find white spaces from source program, which is assigned to core 2. Comments Beginning Pointer (CBPtr) and Comments Forward Pointer (CFPtr) are used to find comments from source program, which is assigned to core 3, and New Line Pointer (NLPtr) to keep track of line number, which is assigned to core 4. These pointers are independently working on its assigned core. The architecture diagram represents the tasks of cores as follows:

Core 1 – Lexeme-Handler – It finds the category of tokens. (Keyword, Identifier, Constant, Special Characters, Operators)

Core 2 – Whitespace-Handler – It finds whitespaces and removes it. Core 3 – Comment-Handler – It finds comments and removes comment lines. Core 4 – Newline-Handler – It keeps track of line number.

As shown in figure 4 lexeme, white space, comments and newline tasks are assigned to core 1, 2, 3 and core 4 respectively. Core 1-Whitespace-Handler uses WSPtr to find the white space. This pointer is pointed to start of the input buffer. WSPtr moves forward till it finds whitespace. Once it finds whitespace, it removes the white space from source program. This process is repeated till the end of the program. Core 2-Lexeme-Handler uses LBPtr and LFPtr. These pointers are pointed to start of the input buffer, only LFP moves forward and while moving forward it checks for pattern and it uses longest matching rule Ref. [2]. If the pattern matches with lexeme, it extracts the set of characters between LBPtr and LFPtr. If

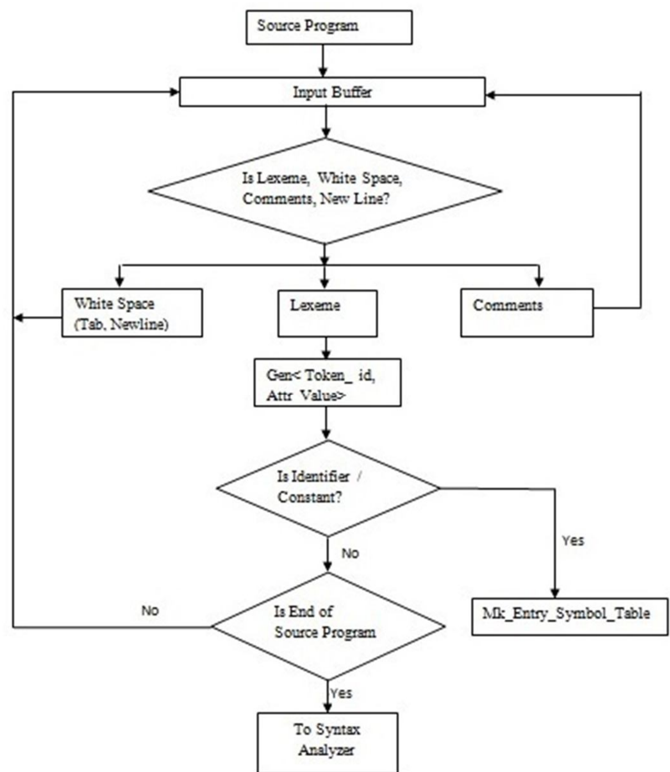
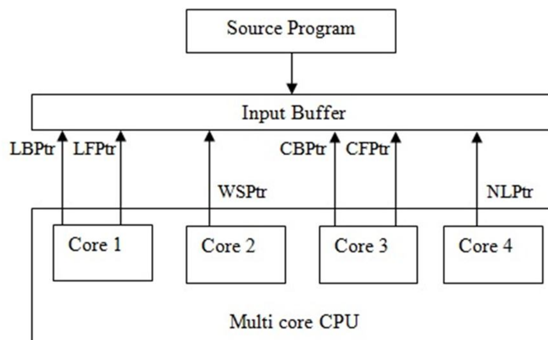


Figure2. Existing system of Lexical Analyzer



LBPtr → Lexeme Beginning Pointer

LFPtr → Lexeme Forward Pointer

WSPtr → White Space Pointer

CBPtr → Comment Beginning Pointer

CFPtr → Comment Forward Pointer

NLPtr → New Line Pointer

Figure3. Architecture of Concurrent execution of tasks of Lexical Analyzer

extracted lexeme is a token, it generates <token id, value> and if token is identifier or constant then it makes entry to symbol table. This process is repeated till the end of the program. Core 3 – Comment-Handler uses CBPtr and CFPtr to find the comments from the source program. These pointers are pointed to start of the input buffer. CBPtr and CFPtr moves forward till they find '/' character. Once it finds '/' character, CFPtr pointer moves forward to find '/' or '*'. If it finds '/' next to '/' which is initiated by CBPtr, then it removes the whole line from the source program or if it finds the '*' next to '/' initiated by CBPtr, then CFPtr keeps moving forward till it finds '*' followed by '/'. Once it finds '*' followed by '/' then statements between CBPtr to CFPtr are removed from the source program. This process is repeated till the end of the program. Core 4 – Newline-Handler uses NLPtr to keep the track of new line. NLPtr counts the '\n' sentinels from source program, this count is used by error handler to correlate the error message with line number.

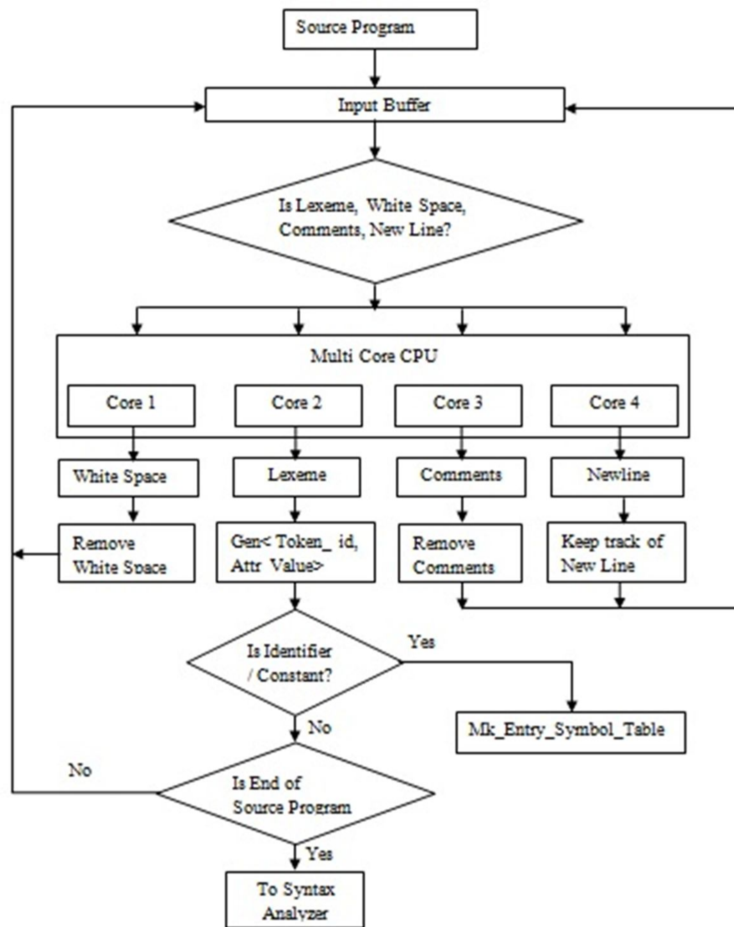


Figure4. Proposed System of Lexical Analyzer

Lexeme-Handler
It finds the category of tokens (Keyword, Identifier, Constant, Special Characters, Operators). Below pseudo code is used to find an identifier.

Pseudocode1

```
Start
Input Buffer buff [ ]
LBPtr=0, LFPtr=0;
Defining Grammar
U → [ \_ ]
L → [a-zA-z]
D → [0-9]
WS → [ ]
ID → [U|L][L|D]+
While( !End of Input String )
{
        If (LBPtr==0 && LFPtr==0)
        {
                While (buff [LFPtr] != WS)
                        LFPtr++;
                Lexeme = Extract string between LBPtr to LFPtr;
                Apply pattern ID to Lexeme
```

```
                If (ID)
                Make entry to symbol table
        }
}
End
```

Whitespace-Handler

It finds whitespaces and removes it.
Below pseudo code is used to find white space.

Pseudocode2

```
Start
Input Buffer buff [ ]
WSPtr=0;
Defining Grammar
 WS → [ ]
While( !End of Input String )
{
    If (buff[WSPtr]==WS)
     {
       Delete White Space
      WSPtr++;
      }
  }
End
```

Comment-Handler

It finds comments and removes comment lines.
Below pseudo code is used to find comments.

Pseudocode3

```
Start
Input Buffer buff [ ]
CBPtr=0, CFPtr=0;
While( !End of Input String )
{
    If (buff [CBPtr] ==’/’)
     {
      CFPtr++;
      If (buff [CFPtr] ==’/’)
       Delete entire line;
     Else if (buff [CFPtr] ==’*’)
      {
        While(1)
        {
            CFPtr ++;
            if (buff [CFPtr]==’*’)
            {
              CFPtr++;
              If (buff [CFPtr]==’/’)
             {
                Delete string from buff[CBPtr] to buff[CFPtr];
                Break;
             }
            }
```

```
            }
        }
    }
         While (buff [LFPtr] != WS)
         LFPtr++;

         Lexeme = Extract string between LBPtr to LFPtr;
        Apply pattern ID to Lexeme
        If (ID)
       Make entry to symbol table
}
End
```

Newline-Handler

It keeps track of line number.
Below pseudo code is used to find newline.

Pseudocode4

```
Start
Input Buffer buff [ ]
NLPtr=0, Count=0;
While( !End of file )
{
    If (buff[NLPtr]=='\')
      {
           NLPtr++;
           If(buff[NLPtr]=='n')
          {
              Count++;
             If (IsError)
              Return line number = count to Error Handler;
          }
       }
}
End
```

Parallelism can be achieved in OpenMP by using #pragma omp parallel Ref. [5]. With this code, system can create multiple threads. By default system can create 4 threads; user can create more than 4 threads with the help of #pragma omp parallel num_threads(n) method, where n is an integer. User can get the thread number by omp_get_thread_num() method. #pragma omp sections directive is the way to distribute different tasks to different threads. Syntax of section directive is as follows:

```
#pragma omp parallel
        {
                #pragma omp sections
                {
                        #pragma omp section
                        {  // code for Task 1 }
                        #pragma omp section
                        {  // code for Task 2 }
                        #pragma omp section
                        {  // code for Task 3 }
                }
        }
```

The above mentioned pseudo codes of tasks of lexical analyzer - Pseudo code 1, 2, 3, 4 can be written in OpenMP Sections directive to perform concurrent execution of tasks of lexical analyzer, which is given below:

Concurrent execution of tasks of Lexical Analyzer

Pseudo code for Concurrent execution of tasks of Lexical Analyzer

```
Start
Input Buffer buff [ ]
 #pragma omp parallel shared (buff)
  {
      #pragma omp sections
       {
            #pragma omp section
            {
                // Pseudo code for Identifier
                     tid = omp_get_thread_num();
                     printf(" Thread %d starts..\n", tid); //Thread 1 Starts
                     goto pseudocode1
            }
            #pragma omp section
            {
               // Pseudo code for White Space
                   tid = omp_get_thread_num();
                   printf(" Thread %d starts..\n", tid); //Thread 2 Starts
                   goto pseudocode2
             }
             #pragma omp section
            {
                 // Pseudo code for Comments
                    tid = omp_get_thread_num();
                    printf(" Thread %d starts..\n",tid); //Thread 3 Starts
                   goto pseudocode3
             }
            #pragma omp section
            {
                 //Pseudo code for Newline
                     tid = omp_get_thread_num();
                     printf(" Thread %d starts..\n",tid); //Thread 4 Starts
                      goto pseudocode4
             }
      }
}
End
```

Speedup

According to Amdahl's law Ref. [1], execution time is improved if independent code/ tasks are divided into multiple processors and executed parallel / concurrently. In this paper we are dividing tasks of Lexical Analyzer into multiple cores to execute concurrently in order to improve execution time.
As mentioned in Ref. [7] Speed up is performance metric which is given as:

$$T_{parallel} = \sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p)$$

Where n is size of problem
p is number of processors
σ (n) is the program's serial part execution time,
ϕ(n) is the program's parallel part execution time, and
κ(n, p) is the communication time.

Efficiency is the ratio of speed up obtained to the number of processors used Ref. [7]. It measures processors utilization. Parallel system efficiency of solving an n-size problem on P processors is given by

$$0 \le \varepsilon(n, p) \le \frac{\psi(n, p)}{p} \le 1$$

Our algorithm is designed in such a way that operations are performed in parallel/concurrent manner and there is no communication required among cores and no sequential execution.

In this paper, sequential operations refer to sequential lexical analysis of all tasks on single processor whereas operations in parallel/concurrent refer to sequential lexical analysis of all tasks independently distributed among available cores.

Based on Ref. [4] Ref. [6] and Ref. [7] Parallel/concurrent execution is more efficient than sequential execution. In this paper, tasks of lexical analyzer are distributed independently among different cores where core 1 performs the task of Lexeme-Handler, core 2 performs the task of Whitespace-Handler, core 3 performs the task of Comment-Handler, and core 4 performs the task of Newline-Handler. These tasks are executed concurrently so it improves the efficiency of lexical analyzer phase of compiler.

Conclusion and Future Work

An algorithm for performing parallel/concurrent execution of tasks of lexical analysis which can use multi-core machine is presented. It is clear that substantial amount of time can be saved in lexical analysis phase by distributing tasks of lexical analyzer across number of cores. So in this paper tasks of lexical analyzer such as token generation, removing white spaces and keeping track of line for error handler are done concurrently by sections directive from OpenMP which improves the speed of lexical analyzer.

This speedup is expected to increase further if source code (which is input to lexical analysis phase) is divided on some criteria and processed parallel. We are currently working on further phases of compiler to improve their efficiency in order to improve the overall efficiency of compiler.

References

[1] Amit Barve, Brijendra Kumar Joshi. “Improved Parallel Lexical Analysis Using OpenMP on Multi-core Machines” ELSEVIER-Volume 49, 2015, Pages 211-219

[2] Alfred Aho and Jeffrey Ullman, “Principles of Compiler Design” Second Edition.

[3] P. Gepner, M.F. Kowalik “Multi-Core Processors: New Way to Achieve High System Performance” IEEE- ISBN: 0-7695-2554-7- 16 October 2006 Multi core

[4] Michael J. Quinn “Parallel Programming in C with MPI and Open MP” Published by McGraw-Hill Education (2003) ISBN 10: 0070582017 ISBN 13: 9780070582019

[5] Paul Edmon “Introduction to OpenMP” HARVARD Faculty of Arts and Sciences, ITC Research computing Associate. Section

[6] Mark D. Hill , Michael R. Marty ”Amdahl's Law in the Multicore Era” Published by the IEEE Computer Society July 2008

[7] Alaa Ismail El-Nashar “TO PARALLELIZE OR NOT TO PARALLELIZE, SPEED UP ISSUE” Published by IJDPS Vol.2, No.2, March 2011

[8] Syntax analyzer, “https://www.tutorialspoint.com.”

[9] Amit Barve, Brijendra Kumar Joshi, “Parallel Lexical Analysis of Multiple Files on Multi-Core Machines” Published by IJCA Volume 96– No.16, June 2014

[10] Amit Barve, Brijendra Kumar Joshi,, “Parallel Lexical Analysis on Multi-Core Machines using Divide and Conquer” Published by IEEE, 04 April 2013