Parallel Lexical Analysis of Multiple Files on Multi-Core Machines

Amit Barve
Assistant Professor
CSE- Deptt. VIIT Pune, India

Brijendra Kumar Joshi
Professor
MCTE Mhow, India

ABSTRACT
The multi-core machines open new doors to achieve parallelism in single machine. This new architecture has influenced every field of computing. Parallel Compilation is one of the areas that still needs serious research work to fully exploit the inherent power of the architecture. In this paper a parallel lexical analysis algorithm is presented that is capable of doing lexical analysis of more than one source file simultaneously thereby improving performance.

Keywords
Lexical Analysis, Parallel Lexical Analysis, Flex, Processor Affinity.

1. INTRODUCTION
Compiler is a complex program that translates source program written in one language into an equivalent program in target language. This translation process is often carried out through number of phases. Lexical analysis, also known as scanning, is the first and very important phase. The lexical analyzer reads stream of characters and produces as output a number of *tokens* which are consumed by the next phase of the compilation process i.e. syntax analysis. In addition to tokenization the lexical analyzer also recognizes instances of tokens referred to as lexemes. A *lexeme* is used to uniquely identify the tokens. For example, suppose two variables of type *int* are declared in C language as

int number, position;

then token generated for *number* and *position* would be same. If **ID** is the token used for variable names then on reading *number* and *position*, the lexical analyzer would generate token **ID**. The lexeme corresponds to these two tokens i.e. **ID** (for *number*) and **ID** (for *position*) would be "number" and "position" respectively. Detailed explanation of all the phases of a compiler are dealt with in excellent texts [1][2][3][4].

Compiler and its phases are mostly written in sequential manner and for single processor systems. With the advent of multi-core machines, it is possible that complier and its phases be parallelized. In this paper we present an approach to parallelize lexical analysis phase for huge number of files.

2. LEXICAL ANALYSIS
Lexical analysis is the only phase of a compiler which interacts with the original source code. It reads the sequence of characters and produces stream of tokens and makes their entry in a data structure called symbol table. The tokens and entries in symbol table are further used by the following phase i.e. syntax analyzer.

The program which performs lexical analysis is called a lexical analyzer, lexer or scanner**.** Figure 1. Shows the overall process of communication between lexical analyzer and syntax analyzer. Lesk and Schmidt [5] developed a tool for generating lexical analyzer from specifications in the form of regular expressions. Now a days Flex [6] is used for generating lexical analyzer

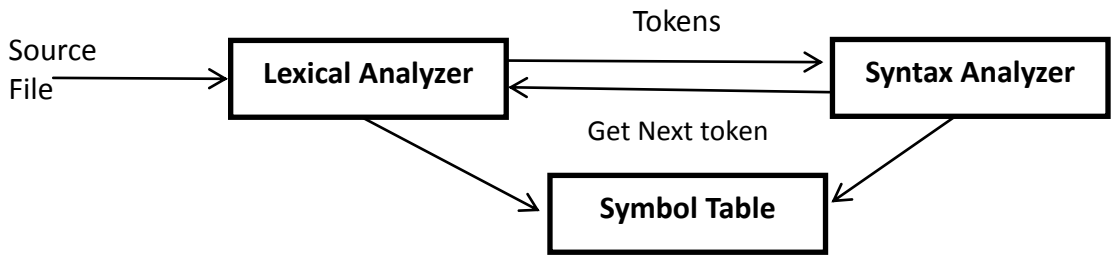

Figure 1. Interaction of Lexical Analyzer with Parser

3. PARALLEL LEXICAL ANALYSIS
In the past, various attempts have been made to perform parallel lexical analysis. Umarani developed a parallel lexical analyzer for cell processor architecture [7]. In his approach the source code was split into fixed sized blocks using dynamic block splitting algorithm. Daniele and Gregory [8] compiled the original flex kernel and ran it on each SPEs (Synergetic Processing Elements) on IBM cell processor. Mickunas and Shell [9] split lexical analysis process into scanning and screening. The proposed work suggested that the scanning of a text can be done by more than one task in parallel. They have provided only theoretical explanation of the proposed method.

Barve and Joshi[10] developed a parallel lexical analyzer for multi-core machines. Their work is based on detection of parallel constructs in C programs and performing lexical analysis of each construct on different CPUs, using *flex*. They considered *for, while, do..while* loops, *switch..case, if..else* statements as the potentially parallel constructs. They extended their work of parallel lexical analysis of programs by identifying the blocks that can be processed in parallel. The source code was split into number of blocks by

identifying pivot locations. Pivot elements considered were new line character, white space character and various constructs as mentioned above. The detailed comparison of all three algorithms can be found in [11].

The past work is limited up to a single file. For large software like GCC [12], Linux kernel [13], in which thousands of C files are present these approaches will not be suitable. In this paper the work is extended so that large numbers of files can be analyzed for lexical errors.

4. PROCESSOR AFFINITY

The binding of any process to any processor can done in Linux through setaffinity*()* function[14][15]. *taskset* command can be used to load a program from permanent storage and bind it to a processor. These two features can be used to schedule any program/process to any of the available processors. For example, to bind a process whose pid is 222. Following command can be used

taskset – p 1 222

Where p stands for process, 1 represents the CPU number and 222 is the process id of the process. This sets the affinity of process 222 to CPU 1.

5. SPEEDUP

The parallel methodologies in programming designed to aim that parallel programs execute faster as compare to sequential. Speed up is the ration between sequential and parallel execution time, it can be represented as

$$Speedup = \frac{Sequential\ exectution\ time}{Parallel\ exection\ time}$$

The operations performed by a parallel algorithm can be put into three categories [16]:

a. Operations that must be performed sequentially.
b. Operations that can be performed in parallel.
c. Operations requiring communication among processors.

In this paper, sequential operations refer to sequential lexical analysis of all files on single processor whereas operations in parallel refer to sequential lexical analysis of files distributed among available processors. Since lexical analyses of files are independent from one another the communication overhead is almost negligible. Though communication overhead is present between master processor distributing tasks and processors executing these tasks, it is present only when tasks are distributed and when they finish. It is assumed that the communication overhead is zero as compared to actual lexical analyses.

6. PARALLEL LEXICAL ANALYSIS OF MULTIPLE FILES

For doing parallel lexical analysis of multiple files first we need to select the folder which has all files of the project. After selection we need to do lexical analysis of files present in the folder in parallel. Parallel lexical analysis can be done by selecting the file and scheduling it to a specific processor. The algorithm is as follows:

1. Select the source folder.
2. Scan the source folder. While scanning, write the following information in a file say file.txt
   a. Path of files in the package.
   b. Size of the files.
3. Open the file.txt in read only mode.
4. For each line written in file.txt do the following in parallel.
   a. Select a file from the folder.
   b. Perform lexical analysis on selected file by assigning processor affinity.

7. EXPERIMENTAL RESULTS

The Experiment based on above algorithm was carried out on Ubuntu 10.04 LTS on Sony Vaio Core i7 Laptop with 4GB RAM and Processor Speed 1.73GHz having 8 cores in total. For testing and appreciable results a huge software is required therefore GCC 4.0.0 software package was explored and used. Only C files were considered for the purpose. In total 6318 C files are present in the package. Minimum file size is 10 bytes (trivial.c) and maximum is 686.8 KB (parse.c). To get accurate results and time *init* process whose pid is 1 was bound to CPU 0 using *setaffinity()* and remaining were exclusively used for parallel lexical analysis. Table 1 shows time taken in lexical analysis of all C files of GCC. It is clear that significant amount of time can be saved by the use of this approach for a large software package. Figure 2 demonstrates the result pictorially.

Table 2 and Figure 3 show the average speedup achieved by using the algorithm on the GCC mentioned above. The speedup is averaged across 10 runs of the algorithm.

Table 1. Time taken in Parallel Lexical Analysis of GCC 4.0.0 package

No of CPUs	Time (in Minutes)
1	1.50
2	1.41
3	1.37
4	1.35
5	1.31
6	1.28
7	1.25

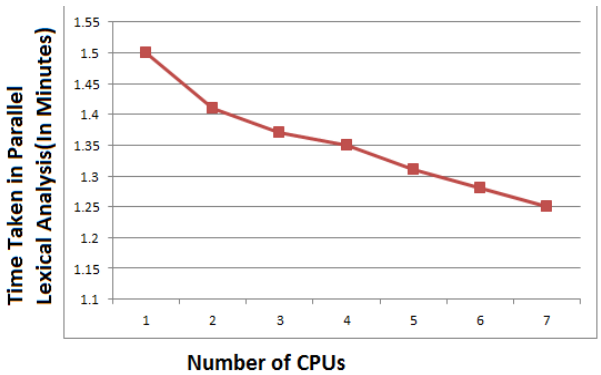

Figure 2. Time taken in Parallel Lexical Analysis of GCC 4.0.0 Package

Table 2. Speed Up in Lexical Analysis process

Number of CPUs	Speed Up
1	1
2	1.06
3	1.09
4	1.11
5	1.14
6	1.17
7	1.25

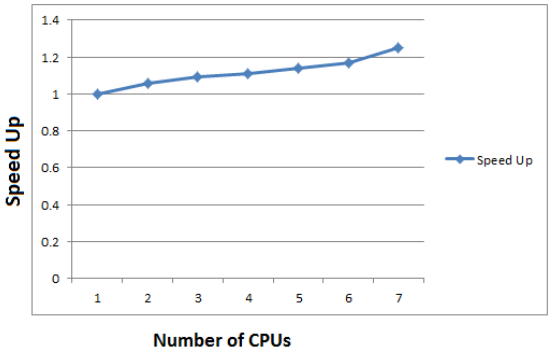

Figure 3: Speed Up

8. CONCLUSION

An algorithm for parallel lexical analysis of multiple files which can use multi-core machines is presented. It is clear from experiments that substantial amount of time can be saved in lexical analysis phase by distributing files across number of CPUs. With the increase in number of processors the overall time in compilation would definitely be far too less as compared to all-serial approach. Speedup can be further investigated if individual files can also be scheduled for lexical analysis on multiple processors using one of the approaches explored earlier [10][11].

9. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D.Ullman; "*Principles of Compiler Design*"; Addison Wesley Publication Company, USA, 1985.

[2] Alfred V. Aho, Ravi Sethi, Jeffrey D.Ullman; "*Compilers: Principles, Techniques and Tools"*;Addison Wesley Publication Company, USA, 1986.

[3] Jean Paul Tremblay,Paul G. Sorenson;"The Theory and Practice of Compiler Writing";McGraw-Hill Book Company USA 1985

[4] David Gries; "*Compiler Construction for digital Computers*"; John Wiley & Sons Inc. USA, 1971.

[5] M. E. Lesk, E. Schmidt; "*Lex- A Lexical Analyzer Generator*"; Computing Science Technical Report No. 39, Bell Laboratories, Murray Hills, New Jersey, 1975.

[6] http://flex.sourceforge.net/

[7] G. Umarani Shrikant ;"Parallel Lexical Analyzer on the Cell Processor"; IEEE SSIRI-C 2010; pp. 28-29;2010.

[8] Daniele Paolo Scarpazza, Gregory F. Russell;" High Performance regular expression scanning on Cell /B.E. Processor; ICS 2009; pp. 14-25, 2009.

[9] M. D. Mickunas, R. M. Schell; "*Parallel Compilation in a Multiprocessor Environment*"; Proceedings of the annual conference of the ACM, Washington, D.C., USA, pp. 241–246, 1978.

[10] Amit Barve and Dr. Brijendra Kumar Joshi;"A Parallel Lexical Analyzer for Multi-core Machine"; Proceeding of CONSEG-2012,CSI 6th International confernece on software engineering;pp 319-323;5-7 September 2012 Indore,India.

[11] Amit Barve and Brijendra kumar Joshi, "Parallel lexical analysis on multi-core machines using divide and conquer," *NUiCONE- 2012 Nirma University International Conference on Engineering* , pp.1,5, 6-8 Dec. 2012. Ahmedabad, India.

[12] gcc.gnu.org/

[13] https://www.kernel.org/

[14] http://www.linuxjournal.com/article/6799.

[15] http://www.cyberciti.biz/tips/setting-processor-affinity-certain-task-or-process.html

[16] Michael J. Quinn;"*Paralle Programming in C with MPI and OpenMP* "; pp.159-160. Tata McGraw-Hill Publication, New Delhi 2003.