Volume 6, Issue 6, June 2016 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

A Lexical Analyzer for Evaluation of Loops in Multi-Processors using Open MP

Shreya Tamrakar[*], **Ajay Kushwaha**
CSE, CSVTU, Chhattisgarh,
India

Abstract— The lexical analyzer is the first step of the compiler. Compiler having six steps with different task. All the steps of the compiler have done their work in a very proper and specific way. Here we are only focusing on the first step of the compiler that is Lexical Analyzer. To implement the lexical analyzer we are using lex tool to generate tokens, which is what lexical analyzer actually do. Lex is lexical analyzer generator is very effective and efficient tool to generate lexical analyzer. Here we are implementing the lexical analyzer with multi – core processor. One more thing is discussed here is parallel processing of the lexical analyzer with OpenMp concept. The OpenMp concept is to apply the parallel processing in lexical analyzer. The word OpenMp means "Open Multi Processing". The OpenMp concept is basically working on threads. Here we are using 2-3 threads for the experiment. For more improvement and more efficient result we can increase the number of threads or cores for the experiment.

Keywords— Lexical Analyzer, Multi – Core Machines,Parallel Processing,OpenMP concept,Lex and Yacc

I. INTRODUCTION

This document is a template. An electronic copy can be downloaded from the Journal website. For questions on paper guidelines, please contact the journal publications committee as indicated on the journal website. Information about final paper submission is available from the conference website.

The Lexical Analysis is the first step of the compilation process. The lexical analyzer has the task to create tokens for each word given in the source program. The lexical analyzer split all the sentences in the given program and grouped into tokens. Tokens are sequence of characters with collective parallel lexical analyzer has been implemented using the OpenMp connect. OpenMp means Open Multi Processing. Multi processing means multiple processes. The tokens created by the lexical analyzer are collectively called the lexemes. Here we are discussing about the parallel lexical analyzer. that may run or execute together or in parallel. So if we are talking about the parallel processing it can be implemented using threads. Threads are the very interesting part in parallel programming. But if we are doing it in parallel it may be easier and efficient if we are using the processor with multiple cores. So it was very interesting to implement the parallel lexical analyzer with OpenMp concept.

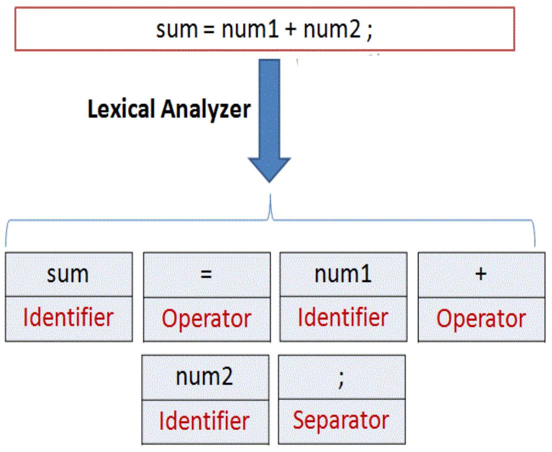

Fig-1: Working of Lexical Analyzer

1.1 Parallel Computer:

A parallel Computer is a multiple – processor computer system supporting parallel programming. Two important kind of parallel computers are multi – computers and centralizer multiprocessors. A multicomputer is a parallel computer constructed out of multiple computers and an interconnection network. The processors on different computers collaborate by passing messages to each other. A centralized multiprocessor also called a symmetrical multi-processor is more highly integrated system in which all CPUs share access to a single global memory. This shared memory supports communication and synchronization among processors.

1.2 Parallel Programming:

Parallel programming in a language that permit you to explicitly designate how, different portions of the computation may be executed concurrently by different processors.

1.3 Parallel programming using OpenMP Concept:

A set of major computer accessory dealer together describe the Open MP Application Program Interface (API). OpenMP gives a movable, extensible model for programers of shared memory parallel applications. This API supports C/C++ and FORTRAN on a wide variety of architectures.

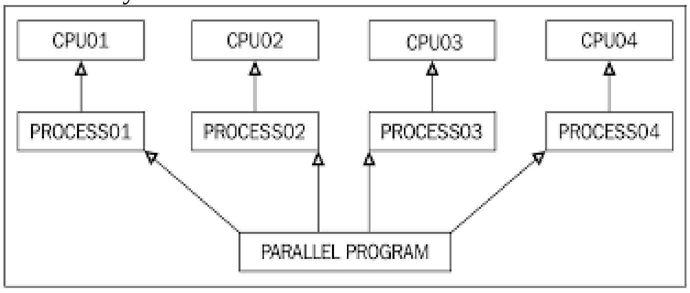

Fig-2: Parallel Programming Concept

II. LITERATURE REVIEW

In a compilation process where the parallel processing Very first time, Micknas and shell [14] identified the portion is inherent. They discussed about to break lexical analysis into scanning and screening. Their work proposed that the scanning of a text can be done by more than one task in parallel that is break the input into number of segments and pre-scanner per segment accumulate the initial and final states at each segment break. They have given only theoretical description. Deniele and Gregory [16] compiled the original flex kernel to run on each of the 8 SPEs(Synergetic Processing Elements) in cell processor. A cell processor has been developed by G. umarani. This is the first implementation of the parallel lexical analyzer. The concept is to breaking up the source program into fixed number of blocks and then doing lexical analysis in parallel.

In the past various strive have been done for parallel tokenization. The conventional method of constructing lexical analyzer is very tedious [13]. The first successful experiment to automatically generate lexical analyzer for sequential program was Lex[14]. Lex (Lexical Analyzer Generator) and YACC (Yet another Compiler Compiler) are the tool which helps to create a compiler. Anyone can create their own compiler by using Lex and YACC tools. The Lex will read your patterns and generate code for lexical analyzer or scanner [11]. In the erection of lexical analyzer it is necessary to identify some part of the code or statement of existing language that can be transformed into parallel blocks [7]. This is urgent to create parallel constructs of the processes. These blocks have been taken as input and generate lexemes.

Very first time, Micknas and shell [14] identified the portion in a compilation process where the parallel processing is inherent. They proposed to break lexical analysis into scanning and screening. Their work proposed that the scanning of a text can be done by more than one task in parallel that is break the input into number of segments and pre-scanner per segment accumulate the initial and final states at each segment break. They have given only theoretical description. Deniele and Gregory [16] compiled the original flex kernel to run on each of the 8 SPEs(Synergetic Processing Elements) in cell processor. G Urmani [17] developed a parallel lexical analyzer for the cell processors. The concept is based on breaking up the source program into fixed number of blocks and doing lexical analysis in parallel.

For a program to work it is vital to recognize some part of code or statement of existing language that can be converted into parallel blocks. Bob Beak and Dave Ollen [18] developed a model for parallel programming. This model expands the Unix model and uses language extensions. It requires runtime library support from operating system too.

Linux kernel 2.6 and after succor binding of any process to any processor through setaffinity() function. It is also feasible to load a program from perpetual storage and bind it to a processor using taskset command. These two characteristics can be used to schedule any program / process to any of the available processors.

Their study was only for the loops, and their concept is to reduce the process scheduling time of the loops. With the advent of Parallel Lexical Analyzer it is possible to reduce the scanning time and the tokenizer generate tokens faster. They will implement their methodology and apply it. As the result comes an optimized lexical analyzer is there and the process scheduling time reduced as the number of CPU increases. They are using Round Robin Algorithm for their experimental use. In their study for the ease they have taken for loop only.

III. METHODOLOGY

The method that we are using to implement the parallel lexical analyzer is the Lex tool with OpenMp concept. The OpenMp concept is used to implement the parallel programming concept. Here we are trying to implement the parallel lexical analyzer and focuses on the loops and execution of the loop with this concept in different processors. Here is the flow chart of the process of the execution of the whole program. All these steps are having their own task. The step are explained as follows:

Step – 1: The first step is to compile the str_find.cpp file. The str_find.cpp file is to find the loops and its processing time for the given input to the str_find.cpp.

Step-2: The second step is to apply the threads to parallel computing.

Step-3: Now give the input to the str_find.cpp so that the program may find the loops nad their execution time for the input file.
Step-4: Now run the lex file to generate the lex.yy.c for tokenization.
Step-5: This step is to give the input file to tokenize. After this process we have to find the speed up of the loop execution in different processors. The formula for speedup is :

$$Speed\ up = \frac{\text{Sequential execution time}}{Paralle\ execution\ time}$$

The speed up should be calculated for both the PCs so that we can compare the results for them.

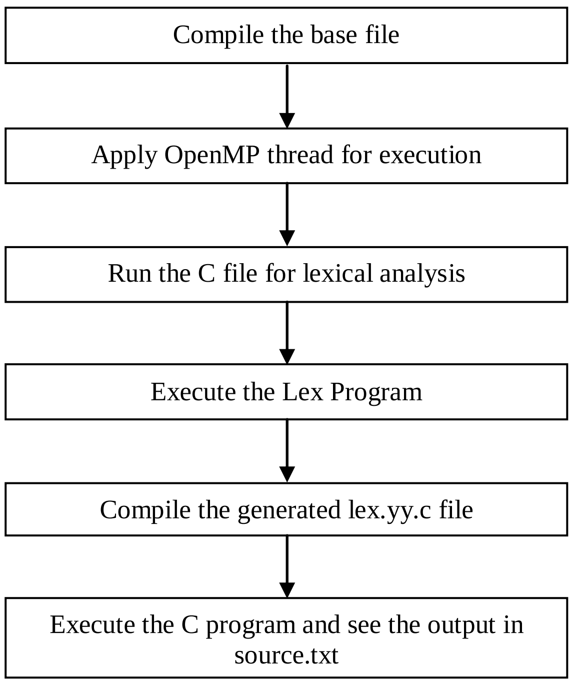
Compile the base file
Apply OpenMP thread for execution
Run the C file for lexical analysis
Execute the Lex Program
Compile the generated lex.yy.c file
Execute the C program and see the output in source.txt

Fig.2: Process and execution

The above figure and the description show the process of the overall system. The parallel lexical analyser generator is used in this process for generating the tokens. And this process will also calculate the execution time for the loops present in the input program. This is the actual parallel programming system.

IV. RESULT

As the methodology shows the process of execution, we are giving different inputs and groped them into number of loops. Here we are using two different systems that are:

1. Processor : Intel® Pentium® CPU B950 @ 2.10 GHz 2.10 GHz
   RAM : 2.00 GB
   System type: 32 bit Operating system
2. Processor : Intel® Core™ I3 CPU M370 @ 2.40GHZ 2.39 GHz
   RAM : 2.00 GB
   System Type: 32 bit Operating System

We are performing all the above shown process on both the systems in sequential as well as in parallel and compare the result.

For the Pentium Processor:

For Pentium processor the table 1 shows the result for sequential processing of the system. the table 2 is for parallel processing. Both tables are calculating time for the different inputs.

Table 1: Time taken in sequential lexical analysis of C programs

No. of loops	Average no. of statements (including state -ments in for loop)	Time taken for (b) (in milli seconds)	Average time taken outside for loop (in milli seconds)	Avg no. of statem -ents in for loop
(a)	(b)	(c)	(d)	(e)
1	12.6	0.0035641	0.00141438	4.5
2	17.2	0.00367276	0.0024165796	10.2
3	21	0.0144287	0.00104387	8.3

4	29.3	0.006714636	0.00491445	9.2
5	36.5	0.00650871	0.0010342	10
8	48.5	0.00812822	0.00125069	15
10	68.9	0.00907454	0.00101322	27.8
14	71.2	0.00905972	0.00104387	31.7

Table 2: Time taken in parallel Lexical analysis of C programs with Pentium processor

Number of for loops	Time taken in for loop (in milli seconds)	Maximum time Max(table1(d),(b)) (in milli seconds)	Speed up (table1.c)/(c))
(a)	(b)	(c)	(d)
1	0.00154523	0.00154523	2.31
2	0.003169088	0.003169088	1.16
3	0.00475037	0.00475037	3.04
4	0.006689015	0.006689015	1.01
5	0.0057079	0.0057079	1.14
8	0.009460535	0.009460535	0.85
10	0.00858322	0.00858322	1.05
14	0.0087315	0.0087315	1.03

Now all the above process will be performed in i3 processor also. We are doing this for comparing the result of both the processors.

For I3 Processor:

For I3 processor the table 3 shows sequential execution for the same input given in Pentium processor and table 4 shows the result for parallel processing. For both the processors we are calculating speedup with the help of given formula that is the speedup is the ration of sequential and parallel processing time.

Table 3: Time taken in sequential lexical analysis of C programs

No. of loops	Average no. of statements	Time taken for(b)	Average time taken outside for	Average no. of statements in for loop
(a)	(b)	(c)	(d)	(e)
1	12.6	0.000931114	0.0026081	4.5
2	17.2	0.004665748	0.00553118	10.2
3	21	0.0052519	0.00153026	8.3
4	29.3	0.007111246	0.003252308	9.2
5	36.5	0.00288026	0.00348406	10
8	48.5	0.00280242	0.010956625	15
10	68.9	0.00348422	0.00448213	27.8
14	71.2	0.00394008	0.000651111	31.7

Now according to the speed up in the loop processing we are plotting a graph so that we can see the difference in the performance of both the PCs.:

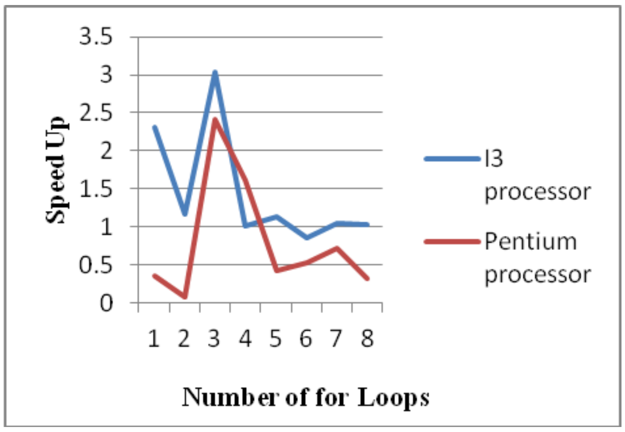

Fig.: performance of both the processors.

The above figure shows the comparison between both the processors. The result shows that the I3 processor gives better performance than Pentium processor.

V. CONCLUSION

The objective of my project is to study and implementation of parallel lexical analyzer and parallel lexical analyzer is tested in multiple core processors. The advantages of using multi core processor systems over tradition single processor systems, and finally compare the computation speed in different cores with similar input.

We performed all the operations with different source programs. The result for the research is discussed previously. The tables shown in above are describing about the speedup of the process for different loops individually as well as jointly. The graph shows the difference between the sequential and multi processing environment. The speed up of the program is also depends upon the number of lines and number of nested loops in the program.

The further work on this research is that we can implement this work for multiple files at a time. Like we can say a parallel lexical analyzer deals with multiple files at a time and tokenize them

REFERENCE

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

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

[3] David Gries; "Compiler Construction for digital Computers"; John

[4] Wiley & Sons Inc. USA, 1971.

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

[6] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[7] Haili Luo,Research of applying regular grammar to making model for lexical analyzer, 6th International Conference on Information Management, Innovation Management and Industrial Engineering,2013

[8] Amit Barve , dr. Brijendra Joshi, Aparallel Lexical analyzer for multi-core machines on CONSEG,2012

[9] Halli Luo, Researchof applying Finite Automaton in the modelling of Lexical analyzer, IEEE International Conference on Information Management, Innovation Management and Industrial Engineering, 2012

[10] Xiaohong Xiao , The design and implementation of C- like Language Interpreter on International Symposium on Intelligence Information Processing and Trusted Computing, 2011

[11] Kai Zhang, Junchang Wang, Bei Hua, Xinan Tang, Kai Zhang, Junchang Wang, Bei Hua, Xinan Tang on IEEE 17th International Conference on paralle*l and Distributed System,2011*

[12] Mohit Upadhyaya , Simple Calculator Compiler Using Lex and YACC on Electronics Computer Technology(ICECT), 3rd international conference on 2011 (Volume-6)

[13] Biswajit Bhowmik , A New Approach of Compiler Design in Context of Lexical Analyzer and Parser Generation for NextGen Language on International Journal of Computer Application (0975-8887) Volume6-No. 11,September 2010

[14] Aastha Singh, Sonam Sinha , Archana Priyadarshi , Compiler Construction on International Journal of Scientific and Research Publications, Volume 3, Issue 4, April 2013

[15] 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.

[16] B. Beck and D. Olien;"A Parallel Programming Process Model";IEEE Software,pp 63-72,may 1989.

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

[18] Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on 01/2010; DOI: 10.1109/SSIRIC.2010