Parallel Syntax Analysis on Multi-Core Machines

Amit Barve
Asst. Professor, CSE
VIIT, Pune (India)
barve.amit@gmail.com

Brijendra Kumar Joshi
Professor
MCTE Mhow (India)
brijendrajoshi@yahoo.com

ABSTRACT -***A multi-core machine has more than one execution unit per CPU on single motherboard. With the advent of multi-core machines parallelization has become an essential part in recent compiler research. Parallel parsing is one of the areas that still needs significant work to utilize the inherent power of multi-core architecture. This paper presents an algorithm that performs parallel syntax analysis of C programs on multi-core architecture. Reasonable speed-up up to 6 was achieved on syntax analysis of C files of GCC 4.8.3.***

Keywords: Parallel Syntax Analysis, Flex, Bison, Processor Affinity, Multi-Core Architecture.

I. INTRODUCTION

A compiler is a program that reads a source program in one language and translates it into an equivalent program in another language. The process of compilation is divided into various phases. The very first phase is known as Lexical Analysis or scanning and the program which performs this task is called lexical analyzer or scanner or laxer. The lexical analyzer takes stream of characters as input and groups them into meaningful sequences called lexemes. For each lexeme, lexical analyzer generates a token which is consumed by subsequent phase i.e. syntax analysis. Syntax analysis phase also known as parsing takes as input a stream of tokens to create a tree-like intermediate representation also known as Syntax Tree. Syntax trees depict the grammatical structure of the token stream. For example, if an expression is written as

a= b+c *20 (1)

then the lexical analyzer output for the given expression would be

$id_1 = id_2 + id_3 * const$ (2)

Here "a" is the lexeme that is mapped to a token id_1 where ***id*** is a symbol used to represent an identifier and ***1*** is an index into the symbol table entry for ***a***. Similarly ***b*** and ***c*** are mapped to tokens id_2, id_3 respectively. The **=, +, *** lexemes are mapped to tokens **=, +, *** respectively, since they are abstract symbols for assignment, addition and multiplication operators in that order. 20 is lexeme that is mapped to the token ***const***. The syntax analyzer takes these tokens as input and produces a parse tree structure shown in Figure 1.

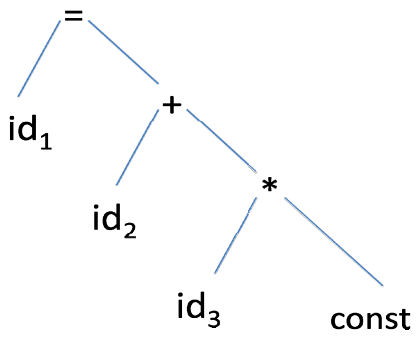

Figure 1: Parse tree for expression a= b+c *20

The detailed description of all the phases of a compiler can be found in popular texts [1][2][3][4].

II. CLASSICAL WAY OF PARSING

The main aim of the syntax analysis phase is to take the tokens produced by the lexical analyzer and use some parsing algorithm to verify that the stream of tokens represents a legal string in the language. The parsing algorithms are mainly classified into two categories, top-down parsing and bottom-up parsing. These refer to the order in which nodes in the parse tree are constructed. In top-down approach the construction of the tree starts from root and proceeds towards the leaves while in bottom up approach parse tree begins with leaves and proceeds towards the root. Some popular top-down parsing algorithms are recursive decent parsing (also called predictive parsing) and non-recursive decent parsing. Bottom-up parsing includes some algorithms like Simple LR parser (SLR), Canonical LR Parser (CLR), and Look Ahead LR (LALR) parsing.

In LR parsing parser reads input from left to right and produces a right most derivation in reverse. The term LR(k) parser is also used, where k refers to the number of unconsumed look ahead input symbols that are used in making parsing decisions. Depending on how the parsing table is generated, an LR parser can be called SLR, LALR,

or CLR Parser. LALR parsers have more language recognition power than SLR parsers. Canonical LR parsers have more recognition power than LALR parsers.

Some researchers further explored these parsing algorithms. Cohen and Roth [5] described an approach for determining the time taken to parse sentences accepted by a deterministic parser. They considered recursive decent parser and a bottom up parser (SLR parser) for the analysis. Gerardy [6] describes the experimental comparison of parsing methods. He explored recursive decent parsing, SLR and operator precedence parsing methods. T. Anderson et al proposed an Efficient LR(1) Parser [7].

III. SYNTAX ANALYZER GENERATOR

Syntax Analyzer generator also known as parser generator takes a grammar specified by the programmer and generates a syntax analyzer that recognizes valid "sentences" in that grammar. Johnson [8] is considered to be the first one to take up the challenge of generating syntax analyzers from specifications in the form of a grammar and the tool developed is by far the most important in compiler development. Now a days Bison [9] is used as parser generator. This tool is open source and is freely available under Linux and its variants. It generates a parser that recognizes LALR(1) grammar. It generates a parser that recognizes LALR(1) grammar.

IV. PARALLEL PARSING

Parallel parsing has been attempted by many in the past. Lincoln [10] first proposed the concept of parallel object code for FORTRAN and COBOL job cards in an environment that consisted of IBM 704 uniprocessors and CDC 6500 of ILLIAC IV. The parallel processing was achieved by assigning completely different user jobs to different processors. Zosel[11] focused on recognizing FORTRAN DO-loops that can be collapsed into vector instructions for CDC 7600 machines. For the first time, Mickunas and Shell[12] recognized the areas in a compilation process where the parallel processing is inherent. They proposed to split lexical analysis into scanning and screening. They also developed a parallel parsing method based on LR parsing. Hickey and Katcoff[13] have analyzed parsing algorithms for upper bound on speedup whereas Cohen and Kolodner[14] have estimated speedup in parallel parsing.

Object Oriented parsing was proposed by Yonezmva and Oshava[15]. Khanna et al[16] proposed the partitioning of grammar to make it suitable for parallel compilation. Chandwani et al[17] developed a parallel algorithm for CKY-parsing for context free grammars. The effort cited in reference [12]-[17] to develop parallel parsing algorithms are of theoretical importance only. Their practical implementations have not been seen so far in real programming languages for multi-cores machines. Barve and Joshi[18][19][20] developed some algorithms for doing parallel lexical analysis on multi-core machines. Their approach is to divide the source code into number of blocks and perform lexical analysis on individual blocks. Their approach was good for parallel lexical analysis. In this paper the concept is extended to parallel syntax analysis.

V. PRACTICAL PARALLEL SYNTAX ANALYSIS ALGORITHM

For doing syntax analysis in parallel for a large software of multiple files first we need to select the folder which has all required files. Since our aim is to do only syntax analysis, individual files can be analyzed syntactically independent of one other. After selection of folder we need to do syntax analysis of files present in the folder in parallel. Parallel syntax analysis can be done by selecting the file and scheduling it to a specific processor for syntax analysis. The steps for parallel syntax analysis are given in figure 2.

Algorithm : Parallel Syntax Analysis
Input: C File, Processor Number
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 individual files (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 the file from the folder. (b) Perform syntax analysis on selected file by assigning processor affinity.

Figure 2: Parallel Syntax Analysis

The above algorithm of fig. 2 was implemented in C for parallel syntax analysis of GCC 4.8.3. The C code is given in fig.3(a), fig.3(b) for lack of space only main functions of the code are given. Calculation of time taken in parsing of individual files is done separately.

```
1.  int main(int argc, char *argv[])
2.  {
3.  FILE *fp;
4.  fp = fopen("file.txt", "r");
5.  assert(fp);
6.  starterline = malloc(10000);
7.  while (fscanf(fp, "%s", starterline)
    != EOF)
8.  {
9.  Extract(starterline, cpuno);
10. if(cpuno==cpumax)
11. {
12. cpuno=2;
13. cpuno--;
14. }
15. cpuno++;
16. }
17. fclose(fp);
18. return 0;
19. }
```

Figure 3(a): C implementation of Parallel Syntax Analysis

```
1.  int Extract(char  *StartLineNo,int
    cpuno)
2.  {
3.  char *run="taskset -c";
4.  char *cpuno1;
5.  sprintf(text1, "%d",cpuno);
6.  cpuno1=text1;
7.  char *scan="./scan";
8.  char *p, text[10000];
9.  char *s=" ";
10. sprintf(text,"%s",starterline);
11. p = text;
12. char *c = malloc(strlen(run) +
    strlen(p) +  strlen(s)+  strlen(s)+
    strlen(s)+  strlen(cpuno1)+
    strlen(scan)+ 1);
13. strcpy(c, run);
14. strcat(c, s);
15. strcat(c,cpuno1);
16. strcat(c, s);
17. strcat(c,scan);
18. strcat(c,s);
19. strcat(c, p);
20. system(c);
21. return 0;
22. }
```

Figure 3(b): C implementation of Parallel Syntax Analysis

The Fig.3 (a) shows the body of main function of the program in which line numbers 3-5 show opening a file named as file.txt which contains the information of all C file present in GCC 4.8.3. Line numbers 7-16 show a while loop in which Extract function is called which takes the line number and CPU number as arguments. The line number is taken from file.txt.

In the Fig.3 (b) the Extract function prepares a string which consists of processor affinity command (line number 3); CPU number (line number 4-6), file name and scan (line number 7), where Scan is the ANSI C syntax analyzer generated using Bison. Final string is then executed by using system call (line number 12-20).

VI. BINDING PROCESSES ON MULTI-CORE MACHINES

The binding of any process to any processor can be done in Linux through setaffinity*()* function[21][22]. *taskset* command can be used to load a program from permanent storage and bind it to a specific processor. These two features can be used to schedule any program/process to any of the available processors. Line number 3 in Extraction function of figure 3 shows the processor affinity using taskset command.

VII. SPEED-UP

The main goal of parallel methodologies in programming is that the parallel programs execute faster as compare to sequential ones. The ratio of sequential and parallel execution time can be represented as Speedup which can be expressed as:

$$Speedup = \frac{Sequential\ execution\ time}{Parallel\ execution\ time} \qquad (3)$$

The operations performed by a parallel algorithm can be put into three categories [23]:

(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 syntax analysis of all files on single processor whereas operations in parallel refer to sequential syntax analysis of files distributed among available processors. Since syntax 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 syntax analyses.

VII. EXPERIMENTAL RESULTS

The experiment based on the above algorithm was carried out on Ubuntu 12.04 LTS on Wipro Netpower server with Intel Xeon E5606 base duel CPU Quad Core machine with 8MB on chip cache and 24 GB RAM and processor speed 2.13 GHz having 8 cores in total. For testing and appreciable results a huge software is required therefore we explored GCC 4.8.3 software package. We have considered only C files. The total of 20,642 C files are present in GCC 4.8.3. Minimum file size is 0 byte (umips-lwp-1.c) and maximum is 6.4 MB (bid_binarydecimal.c). To get accurate

results *init* process whose pid is 1, was bound to CPU 0 using *setaffinity()* and remaining CPUs were exclusively used for parallel syntax analysis.

For the experiment we have considered *ANSI C* grammar specifications to generate both lexical and syntax analyzers. The lexical analyzer is generated using *flex* [24] and syntax analyzer was generated using *bison*. Tables 1-3 show average time taken in syntax analysis of all C files of GCC. In these tables, time taken by 1 CPU is nothing but sequential analysis. The speedup is calculated by using equation (3). For example, in Table 1, for 2 CPUs, the speedup is as given below:

$$Speedup = \frac{\begin{matrix}Time\ taken\ in\\ squential\ syntax\ anaysis\end{matrix}}{\begin{matrix}Time\ taken\ in\ parallel\\ syntax\ analysis\\ by\ 2\ CPUs\ in\ parallel\end{matrix}}$$

$$= \frac{\begin{matrix}Time\ taken\ in\ squential\\ syntax\ anaysis\ by\ 1\ CPU\end{matrix}}{\begin{matrix}Time\ taken\ in\ parallel\\ syntax\ analysis\\ by\ 2\ CPUs\ in\ parallel\end{matrix}}$$

$$= \frac{31.03\ \text{Sec}}{16.28\ \text{Sec}} = 1.90 \qquad (4)$$

Similarly speedup was computed for more number of processor.

The distribution of C files of GCC 4.8.3 is done in ascending order, descending order and random order based on file sizes. Fig. 4 shows the comparison in the speedup of all three file distribution techniques. It is clear from observations that significant amount of time can be saved by the use of this approach for a large software package.

TABLE 1. SYNTAX ANALYSIS OF GCC WITH DISTRIBUTION OF FILES IN ASCENDING ORDER OF SIZE.

No. Of CPUs	Time taken in (Seconds)	Speedup
1	31.03	1
2	16.28	1.90
3	10.9	2.84
4	8.57	3.62
5	6.76	4.59
6	5.66	5.48
7	4.98	6.23

TABLE 2. SYNTAX ANALYSIS OF GCC WITH DISTRIBUTION OF FILES IN DESCENDING ORDER OF SIZE.

No. Of CPUs	Time taken in (Seconds)	Speedup
1	32.5	1
2	16.72	1.94
3	10.91	2.97
4	8.36	3.88
5	7.28	4.46
6	5.86	5.54
7	5.15	6.31

TABLE 3. SYNTAX ANALYSIS OF GCC WITH DISTRIBUTION OF FILES IN RANDOM ORDER OF SIZE

No. Of CPUs	Time taken in (Seconds)	Speedup
1	32.14	1
2	16.91	1.90
3	11.51	2.79
4	8.79	3.65
5	7.21	4.45
6	6.09	5.27
7	5.44	5.90

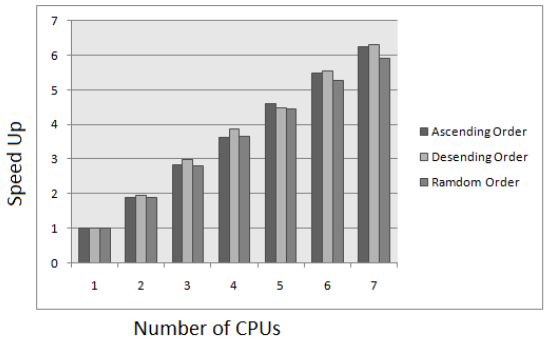

Fig. 4. Speed Up in Syntax Analysis of GCC 4.8.3

IX. CONCLUSION

Parallel syntax analysis of multiple C source files was presented in this paper. It was assumed that lexical analysis of individual files that were scheduled on a processor for syntax analysis was done on the same processor. The speed up obtained for 7 CPUs was 6.31 which is quite reasonable. The speedup would further increase with more number of processors. Though this increment in speedup would be decreasing as the time devoted to distribute files would increase.

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]. J. Cohen, M.S. Roth;"*Analysis of Deterministic Parsing Algorithms*"; Communication of ACM Vol. 21, No. 6, pp.448-458; June 1978.

[6]. R. Gerardy; "*Experimental Comparison of Some Parsing Methods*"; ACM SIGPLAN Notices Vol. 22 Issue 8, pp. 79 – 88; August 1, 1987.

[7]. T. Anderson, J. Eve, J.J. Horning; "*Efficient LR(1) Parsers*"; Acta Informatica Vol. 2, Issue 1 , pp 12-39 1973.

[8]. S. C. Johnson; "*YACC: Yet Another Compiler Compiler*"; Computing Science Technical Report no 32, Bell Laboratories, Murray Hills, New Jersey, 1975.

[9]. www.gnu.org/s/bison**.**

[10]. N. Lincoln; "*Parallel Compiling Techniques for Compilers*"; ACM Sigplan Notices, 10(1970), pp. 18-31, 1970.

[11]. M. Zosel; "*A Parallel Approach to Compilation*"; Conf. REc. ACM Sysposium on Principles of Programming Languages, Boston, MA, pp. 59-70, October 1973.

[12]. 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.

[13]. Timothy Hickey, Joel Katcoff; "*Upper Bounds for Speedup in Parallel Parsing*"; Journal of the ACM (JACM), Vol. 29, No. 2, pp. 408 – 428, 1982.

[14]. J. Cohen, Stuart Kolodner; "*Estimating the Speed up in Parallel Parsing*"; IEEE Transactions on Software Engineering, January 1985.

[15]. Akinori Yonezmva, Ichiro Ohsawa; "*Object-Oriented Parallel Parsing for Context-Free Grammars";* Proceedings of the 12th conference on Computational linguistics – Vol. 2, Budapest, Hungry, pp. 773–778, 1988.

[16]. Sanjay Khanna, ArifGhafoor, AmritGoel; "*A Parallel Compilation Technique Based on Grammar Partitioning*"; Proceedings of ACM annual conference on Cooperation, Washington, D.C., USA, pp. 385 – 391, 1990.

[17]. M. Chandwani, M. Puranik , N.S. Chaudhari, "*On CKY-Parsing of Context Free Grammars in Parallel*"; Proceedings of the IEEE Region 10 Conference, Tencon 92, Melbourne Australia, pp. 141-145, 1992.

[18]. 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.

[19]. Amit Barve and Brijendrakumar 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.

[20]. Amit Barve and Brijendra kumar Joshi; "*Parallel lexical analysis of multiple files on multi-core machines*"; International Journal of Computer Applications; Vol. 96, No.8, June 2014.

[21]. http://www.linuxjournal.com/article/6799?page=0,1.

[22]. http://www.cyberciti.biz/tips/setting-processor-affinity-certain-task-or-process.html (Last accessed on 05-Aug-2014)

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

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