Design Engineering

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

Hybrid Model for Concurrent Execution of Lexical Analyzer on Multi-Core Systems using Dynamic Task Allocation Algorithm and Auto Keyword Detection Method

Vaikunta Pai T.[1], P. S. Aithal[2],

[1]Associate Professor, College of Computer Science & Information Science,
Srinivas University, Mangalore-575001, India.
ORCID: 0000-0001-6100-9023; Email: vaikunthpai@gmail.com

[2] Professor, College of Computer Science & Information Science,
Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0002-4691-8736; Email: psaithal@gmail.com

Abstract:

The processing power of machines will continue to accelerate massively. Modern eras of computing are driven by elevated parallel processing by the revolution of multi-core processors. This continuing trend toward parallel architectural paradigms facilitates parallel processing on a single machine and necessitates parallel programming in order to utilize the machine's enormous processing power. As a consequence, scanner generator applications will eventually need to be parallelized in order to fully leverage the throughput benefits of multi-core processors.

This article discusses the way of processing the tasks in parallel during the scanning stage of lexical analysis. This is done by recognizing tokens in different lines of the source program in parallel along with auto detection of keyword in a character stream. Tasks are allocated line-by-line to the multiple instance of the lexical analyzer program. Then, each of the instances is run in parallel to detect tokens on different cores that are not yet engaged. Theoretical and practical results indicate that the suggested methodology outperforms the sequential strategy in terms of tokenization consistently. It significantly decreases the amount of time spent on lexical analysis during the compilation process.

Keywords: Multi-core architecture, Lexical analyzer, keyword detection

I. INTRODUCTION

Detection of the programming language's tokens is typically done in the lexical analyzer portion of the compiler. A lexical analyzer is a pattern recognition engine that takes an input string of individual characters and separates it into groups of characters known as tokens. [1]. As an

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

output, a stream of tokens is generated for syntax analysis. A token is a sequence of character having a collective meaning and they are basic units of the programming language. It describes the class or category of input string such as keywords, identifiers, literal strings, constants, operators and punctuation symbols. Tokenization is a process of recognizing tokens in a character stream and it is the initial stage of any compiler of high level programming language. It is the most time-consuming operation and has a noticeable impact on the performance of a compiler. According to Lucene [2] benchmarks, tokenization consumes between 14% and 20% of a search engine indexer's time and spends 30% or more of the execution time for XML processing [3, 4]. As a consequence, compiler performance has declined precipitously. According to studies [5], the most computationally intensive phase of the compiler is the lexical analyzer, and optimizing this phase alone can significantly improve the compiler's overall performance. Tokenization necessitates the high-speed, parallel processing capability that multi-core processors are intended to provide. With the ubiquitous use of multi-core processors, developing an efficient task scheduling technique is essential to the multi-core processor's performance. The majority of existing task scheduling algorithms are optimized for single-core processors and cannot be used effectively with multi-core processor systems. Numerous studies on task scheduling have been conducted from a variety of perspectives. Existing task scheduling methods, on the other hand, have a number of shortcomings, including low processor usage, high complexity, and so forth [6].

This paper presents an efficient task allocation and auto keyword detection algorithm for achieving the lexical scanning process in parallel on multi-core processors. It is based on allocating tasks line-by-line to unused cores for detecting keyword patterns in parallel with other token types using dynamic queue. The Experimental findings indicate that the suggested algorithm has better performance in aspects of the overall time required to recognize tokens during the compilation process. The rest of this paper is structured as follows. The second section looks at many key works in this subject. Section three, proposes a parallel lexical analyzer implementation and discusses an algorithm from a theoretical and experimental perspective. Section four evaluates the suggested approach and demonstrates the performance increase using sample test cases. Section five then comes to a conclusion.

II. RELATED WORK

Numerous initiatives have been undertaken in the past to parallelize tokenization and accelerate pattern matching through the use of effective string matching algorithms. Damayanthi Herath et al. [7] examined two different designs for parallel execution. The first method splits the input strings and processes them separately in different threads for pattern matching. The second approach is to divide the pattern set into separate threads and build multiple pattern machines with the same input string. Due to the relatively large size of the pattern text in traditional bio-computing applications in comparison to the input text, the amount of time required to construct the state machine has a significant impact on the total time required for pattern matching. As a

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

consequence, the authors of [7] selected the second method, which included segmenting the massive pattern collection into smaller parts. Then, using a distinct thread, each pattern matching machine with a failure link was built individually, and each machine independently processes the same input string during runtime. The throughput of this experiment was determined by the number of patterns detected per second, and it was conducted using varying numbers of thread counts. When compared to a single thread solution on an eight-core CPU, their method resulted in a greater throughput improvement.

Lin et al. improved throughput by implementing Parallel Failure-less Aho-Corasick (PFAC) [8]. This method implements the Aho-Corasick (AC) algorithm [9] on the GPU. Because it removes all failed transitions from the state system, it minimizes the cost associated with backtracking. As a result, the algorithm's complexity and memory use are reduced. Every character in the input string is allocated to a GPU thread, and all PFAC threads pass through the same failure-free version of Aho-Corasick state machine, which quits in the absence of a valid transition.

Oreste Villa et al. implemented the AC technique using a software-based approach [10]. On the Cray XMT multithreaded shared memory architecture, it was evaluated using a dictionary. They utilized an enhanced version of the AC method known as optimized Aho-Corasick (AC-opt) [11], which constructs the state machine by substituting regular transitions for all failed ones. They demonstrated that the AC-opt method outperforms the original AC algorithm based on their results.

Daniele Paolo Scarpazza et al. [12] investigated high-performance string searching in large dictionaries. They concentrated their efforts on Network Intrusion Detection Systems (NIDS) and the Cell/B.E processor. Their objective is to parallelize AC on the IBM Cell/B.E processor by implementing an improved version that does precise string matching against massive dictionaries. The method was written in C and made use of the CBE language extension as well as an intrinsic.

Daniele Paolo Scarpazza et al., [13], contributed significantly to the development of a parallel lexical analyzer. They provided a method and a set of strategies for performing parallel regular expression-based tokenization leveraging multi-core architecture capabilities such as multiple threads and Single Instruction Multiple Data instructions. As an experiment, the traditional Flex kernel was customized for execution on a multi-core cell processor. The implementation of this technology by Cell/B.E. processors boosted throughput by a maximum of 14.30 Gbps per Cell chip and 9.76 Gbps on Wikipedia input.

Umarani Srikanth [14] studied the possibility of parallelizing a lexical analyzer for use with cell processors. The technique is based on segmenting the original source code into a specified number of parts and executing lexical analysis operations concurrently. The Aho-Corasick algorithm is used to recognize tokens in that experiment. When the parallel version of

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

the lexical analyzer method is run on a system equipped with an IBM Cell Processor, a performance gain is noticed.

Amit Barve et al. [15] attempted to parallelize a lexical analyzer using the concept of processor affinity. The method is based on detecting and identifying tokens in parallel in for-loop statements in source code using memory block. The buffer is used in this algorithm to store for-loop statements detected in source code via a pivot location file comprising the beginning and ending position of each for-loop. The Round Robin scheduling approach is used to allocate tasks to various CPUs and tokens are recognized by processing each line in buffer in parallel using OpenMP programming.

In a computer language, identifier is a token that name the language entities such as variable, function, or label. Reserved words (a.k.a. keywords) are defined with predefined meaning in the language syntax definition. In fact keywords are subset of identifiers and these words cannot be used as an identifier. To look for keyword patterns before identifiers, the keyword pattern specifications are listed before identifiers, while constructing LEX specification. From the regular expression patterns specified in the lexical analyzer generator specification, the lexical analyzer generator generates a transition table for a finite automaton. The lexical analyzer's finite automaton simulator searches the input buffer for regular expression patterns using this transition table.

As noted in related work, the original source code is divided into chunks of a fixed size based on certain criteria. The blocks are then distributed among the cores for parallel processing of lexical analysis. Hence the work tries to introduce a dynamic task allocation strategy. This article demonstrates how to accelerate the lexical scanning process by allocating tasks line-by-line to unused cores to run in parallel on multi-core processors. It also discusses the way to recognize the keywords without specifying the keyword patterns in LEX specifications by developing auto keyword detection method. It minimizes memory requirements by reducing the transition tables.

III. METHODOLOGY

This section demonstrates how the data parallelization technique can be used to accelerate the process of lexical scanning. Is split into two parts:

- Theoretical approach to an algorithm
- Experimental approach

A. Theoretical approach to an algorithm

a) Parallelizing the lexical scanning process on a multi-core system

In this approach theoretical proof for the faster process is achieved.

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

Let "k" be the number of lines in the program and t_i be the time taken for i^{th} line for the recognition of the token fully. Let n be the number of processor working for recognition of the token in parallel. Let T be the total time taken for the recognition of the token for full programme.

Then the average time taken for the recognition of the token for a single line is

$$\bar{t} = \left[\sum_{i=1}^{k} t_i\right]/k \qquad (1)$$

And Standard deviation

$$\sigma_t = \frac{1}{k}\sqrt{\sum_{i=1}^{k}(t_i{}^2 - \bar{t}^2)} \qquad (2)$$

When the average time $\bar{t}$ is equal to the time taken for each line to recognising the token, then the standard deviation is zero. This is indicated by the equation (3)

$$T = \bar{t}k \qquad (3)$$

In case Standard deviation not equal to zero then the total time taken for recognising all tokens will be as shown in (4)

$$T = k(\bar{t} + \sigma_t) \qquad (4)$$

If the number of processors n, working in parallel for recognising all tokens considering each lines of the program for single core will be

$$T = \frac{k}{n}(\bar{t} + \sigma_t) \qquad (5)$$

This is theoretical value for recognising token for 23 line program with average time taken for the recognition of the token =3.66522E-05 and standard deviation = 2.71203E-05 full program in parallel. By evaluating the vales for two three and four cores are as in the table shown below

Time taken (2 cores)	Time taken (3 cores)	Time taken (4 cores)
0.000733383	0.000488922	0.000366692

b) Auto Keyword recognition

To expedite auto recognition of keywords from the formal description of an identifier, keywords are initially recognized as identifiers on the internal level. The pattern is then looked up in a keyword dictionary to return the appropriate keyword token. A pattern of keyword is a finite set of symbols drawn from a language's alphabet that denotes a keyword token, and dictionary is a

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

collection of keyword patterns K = {k_1, k_2, ..., k_n}. On the basis of the dictionary, the keyword detection method generates an automaton. In array approach of automaton, for each character of the text string T, in the worst case, it scans serially over all n total characters in the patterns. Time Complexity of searching is O(mn), where m be the text length and n be the total length of the pattern strings. To improve search performance, rather than searching the keyword pattern strings in serial, search them in parallel. This will eliminate a significant amount of redundant rescanning work.

The keyword recognition automaton is based on the trie of the given keyword dictionary. Trie is a data structure based on tree structure, which can be used to efficiently retrieve a key from a large collection of patterns. Each keyword pattern k_i in the keyword dictionary corresponds to a path in the trie that begins at the root node and includes edges labelled with the pattern's symbols. Each edge that leaves a node has a unique label. Figure 1 illustrates the trie that corresponds to the keyword dictionary {if, int, long, union, unsigned}. A node's label L(v) is the concatenation of all edge labels encountered on the path leading to node v. For each keyword pattern $k_i \in K$, there is a node v in the trie with label L(v) = k_i (in the figure 1, grey colour nodes represent nodes holding valid keywords in C language), and the label L(v) of any leaf node v in the trie equals some pattern $k_i \in K$. The time complexity of building a trie-based pattern matching machine is linear to the total length of given keyword patterns n in dictionary i.e. O(n).

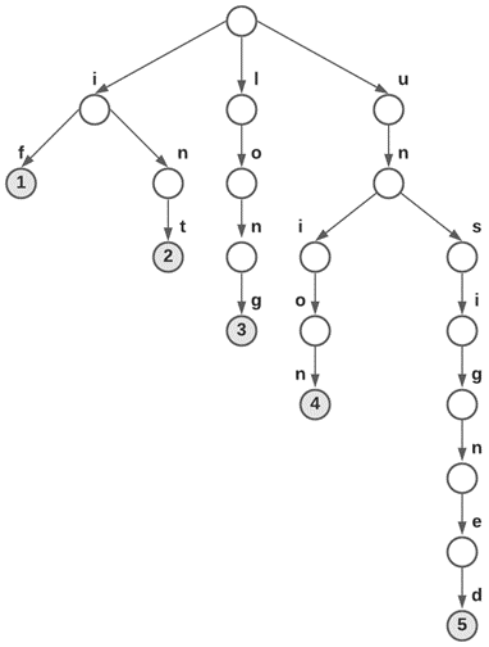

Fig 1: Trie corresponding to some keywords in C – if, int, long, union and unsigned

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

A trie can be used to determine whether or not a given lexeme is of token type keyword, as follows. To find a lexeme s, begin at the root node and extend the path labelled as s. If the search path leads to a node with an identifier i, then the lexeme included in the keyword dictionary and it is keyword pattern k_i. For the search action the complexity will be linear to the length of the lexeme. For each character of lexeme, in the worst case, it scans as most as many characters as exist in the deepest branch of the trie. Time Complexity of searching is $O(mL_{max})$, where m be the length of the lexeme and L_{max} is the length of the longest pattern string, which is a good runtime over array approach of implementation.

B. Experimental approach

a) Auto keyword detection using hashed trie

There are various ways to design a trie, each with a particular trade-off between memory consumption and performance of operations. The basic trie structure stores a set of words over an alphabet A as a linked set of nodes, each of which has an array of child pointers, one for each symbol in the alphabet. The space requirement to store n strings $w_1, \ldots, w_n$ in this basic form is $O(|A|\sum_{i=1}^{n} \text{length}(w_i))$. It can be made space-efficient by implementing trie using hash map to store children of a node. Allocate memory only for alphabets in use, and it will not waste space storing null pointers. It minimizes memory requirements of trie and performs efficient lookups. Space used here with every node here is proportional to number of children which is much better than proportional to alphabet size, especially if alphabet is large. Trie memory optimization using hash map can be very effective in addressing performance optimization. The memory optimization implementation of a dictionary lookup using trie can be accomplished by the following algorithm. It serves the primary purpose of keyword recognition in lexical analyzer.

Function HASHTRIE_SEARCH(LEXEME). Given LEXEME, a currently matched lexeme and ROOT, a pointer to the root element of constructed hashed trie whose typical node contains CHILDREN, which is of type hash map data structure to map hash value to key of a child node. NODE is a pointer to the node being currently processed. Variable CHILD holds hash value mapped to a particular key. Function SUB returns the substring. This function returns "true" if given LEXEME is found in hash trie, "false" otherwise.

Method:

1. [Initialization]
NODE ← ROOT

2. [Perform the search]
Repeat for K = 1, 2, ..., LENGTH(LEXEME)

CHILD ← NODE.CHILDREN[SUB(LEXEME,k,1)]

if CHILD = NULL

then Return(false) (There is no such keyword)

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

else NODE ← CHILD

3. [Keyword found]

Return(true)

b) Parallel lexical scanning using dynamic task allocation

To facilitate parallel tokenization across multiple cores, integrate the dynamic task allocation algorithm, which assigns tasks line-by-line to the core that are not currently engaged. This algorithm uses a set of techniques that take advantage of multi-core features such as multi-processing and processor affinity. To process a specific source file is taken. To facilitate efficient file access within the algorithm, the source file is first mapped to a process address space. Memory mapping is a method of loading a file directly into computer memory. This prevents processor in the I/O queue from experiencing a delay, resulting in a significant improvement in file I/O performance. The algorithm creates a new child process for each line read from the buffer to run the instance of LEX program. The same is then assigned it to the core that is not currently engaged. As a result, assigned core concurrently tokenizes string from each read line. A CPU queue maintains the status of availability of cores. Initially, the CPU queue is filled with all available cores. When a process demands a core, the core at the head of the CPU queue is removed and the core is re-added at the tail of the CPU queue when it becomes free. While the queue is not empty, the scheduler continues to remove cores from the CPU queue's head and allocate them to processes. If the queue is empty, the scheduler receives a wait signal. The line-by-line task allocation to the cores that are not currently engaged and auto detection of keywords can be accomplished by the flowchart shown in Fig. 2.

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

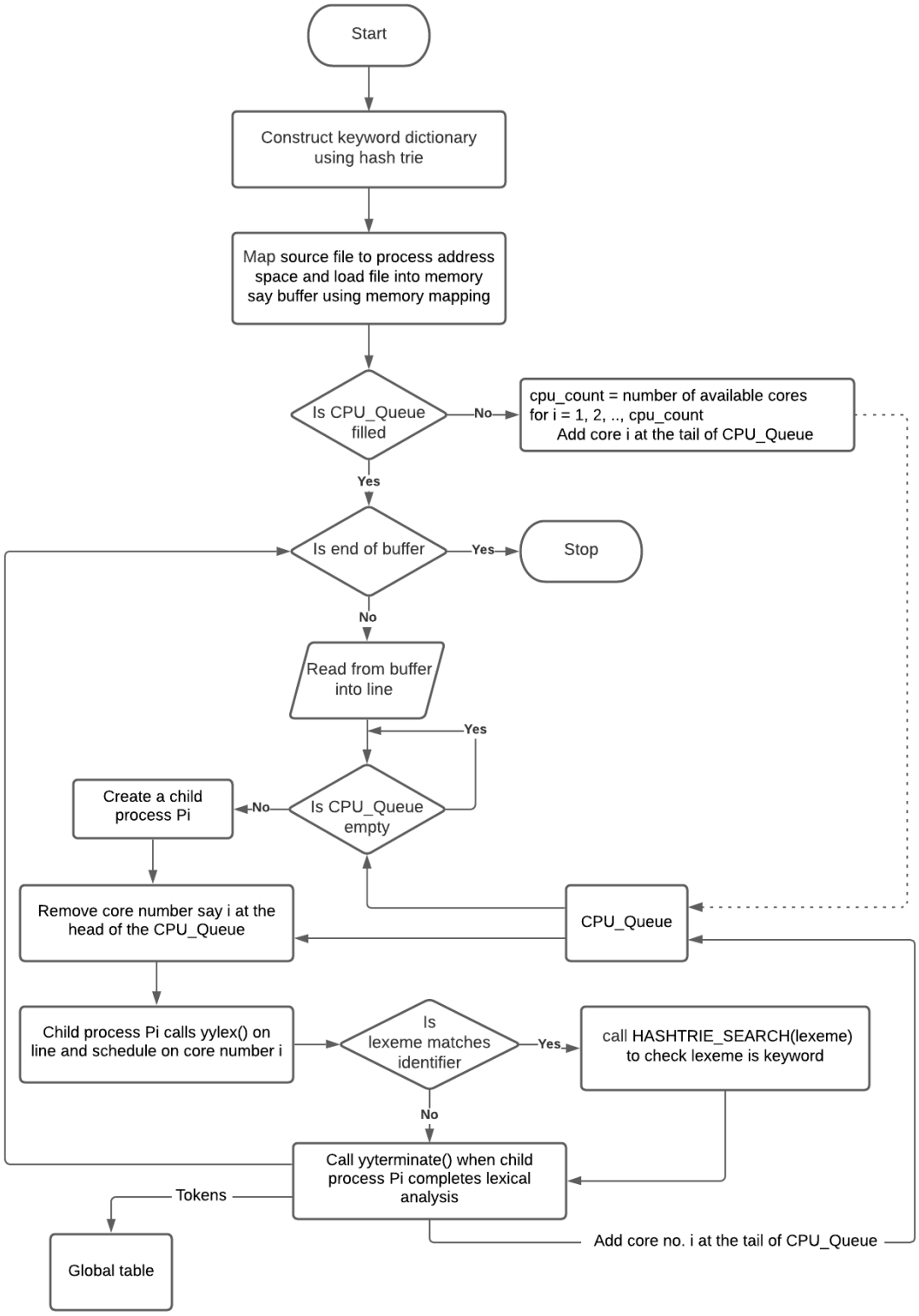

Fig 2: Flowchart of hybrid model for parallel lexical scanning using dynamic task allocation and auto keyword detection method

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

IV. EXPERIMENTAL RESULTS

The experimental system is configured with an Intel Core i7-950 processor, an ASUS P6T-SE motherboard, and 12-GB DDR3 memory running the Ubuntu 18.04.5 LTS operating system. Multiple source files of varying sizes are considered to validate the algorithm. The algorithm is implemented using C programming language with fork system call to create new child processes to run the multiple instances of LEX program on different cores in parallel. Each new child process created is bound to specific CPU by using processor affinity. Whereas the conventional LEX tool generated sequential lexical analyzer runs on a single core.

Experimental results for a variety of CPU core counts and source files of varying sizes have been compiled and summarized in Table 1 and illustrated in Fig. 3. The experimental results show that hybrid model for parallel lexical analyzer on multi-core achieves significant speedup over the sequential lexical analyzer implementation on single core. The result establishes unequivocally that the parallel lexical analyzer's performance should scale linearly with the number of cores.

The result analysed mainly under the following aspects:

a) Speedup: The speedup is defined as the ratio of serial to parallel execution time, which is calculated using the following equation (6).

$$\text{Speed up} = \frac{\text{Time taken for sequential lexical analysis using a single core}}{\text{Time taken for parallel lexical analysis using multiple cores}} \qquad (6)$$

Speedup efficiency in parallel lexical analysis using 4 cores for recognising tokens for 348 line program is shown below

$$\text{Speed up} = \frac{\text{Time taken for sequential lexical anaysis using a single core}}{\text{Time taken for parallel lexical analysis using 4 cores}} = \frac{0.001952}{0.000501} = 2.3765405\text{X}$$

The effect on speedup using sequential and parallel lexical analyzer is shown in table 2. Fig. 4 illustrates the performance enhancements associated with increasing the number of CPU cores.

b) Throughput: The number of bytes being processed per unit of time, which is calculated using the following equation (7).

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

$$\text{Data Throughput} = \frac{\text{Number of bytes in the input}}{\text{Total time taken for sequential / parallel lexical analysis using multiple cores}} \quad (7)$$

The data throughput of parallel lexical analyzer using 4 cores over an input file of size 5.6 KB is shown below

$$\text{Data Throughput for 4 core implementation} = \frac{5{,}628 \text{ bytes}}{0.0055230 \text{ Secs}} = 0.0081521 \text{Gbps}$$

Table 3 compares the performance of the proposed parallel lexical analyzer utilizing multiple cores to that of the sequential method using input files of various sizes. The first and second columns indicate the size of the input and the number of tokens produced, respectively. The maximum data throughput of the parallel lexical analyzer method utilizing 2, 3, and 4 cores is shown in the third, fourth, fifth, and sixth columns, respectively. In Table 3, for processing the input file of size 5.6 KB, parallel lexical analyzer using 4 cores achieves 0.0081521 Gbps of data throughput while sequential approach achieves 0.0034302 Gbps. As shown in Fig. 5, parallel lexical analyzer using 4 cores achieves 2.376566964 times faster than sequential approach.

Table 1: Time required for performing sequential and parallel lexical analysis on C programs with varying numbers of CPU cores

Experiment Number	File Size (Lines)	# of Tokens Generated	Time Taken for Sequential Lexical Analysis using a single core	Time Taken (in seconds) by Parallel Lexical Analyzer		
				2 Cores	3 Cores	4 Cores
1	67	483	0.0032186	0.0026919	0.0017858	0.0013162
2	78	499	0.0034874	0.0028892	0.0019370	0.0013411
3	80	333	0.0027882	0.0023139	0.0015559	0.0011727
4	130	494	0.0045738	0.0037044	0.0024231	0.0019050
5	138	637	0.0051238	0.0040631	0.0029343	0.0021806
6	149	522	0.0049501	0.0040115	0.0027013	0.0019503

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

7	173	875	0.0065838	0.0051914	0.0035454	0.0025732
8	174	836	0.0095566	0.0072449	0.0047305	0.0036860
9	298	1044	0.0115459	0.0088811	0.0056277	0.0045716
10	348	1672	0.0131256	0.0099716	0.0072673	0.0055230

Table 2: The effect on speedup using sequential and parallel lexical analyzer with different no. of CPU cores

Experiment Number	File size (Lines)	Speedup		
		Using 2 Cores	Using 3 Cores	Using 4 Cores
1	67	1.1956415	1.8023598	2.4453860
2	78	1.2070294	1.8004195	2.6003600
3	80	1.2049604	1.7920405	2.3775294
4	130	1.2346839	1.8876132	2.4009663
5	138	1.2610457	1.7462043	2.3497634
6	149	1.2339675	1.8325179	2.5381458
7	173	1.2682030	1.8569732	2.5585882
8	174	1.3190742	2.0202110	2.5926750
9	298	1.3000491	2.0516029	2.5255654
10	348	1.3162983	1.8061086	2.3765405

Table 3: Throughput Comparisons over an input file of different sizes

File Size	# of Tokens Generated	Data Throughput (Gbps)			
		Sequential Lexical Analyzer using a single core	Parallel Lexical Analyzer		
			2 Cores	3 Cores	4 Cores
1.7 KB	499	0.0039732	0.0047958	0.0071533	0.0103318
1.7KB	333	0.0049925	0.0060158	0.0089466	0.0118700
2.2KB	483	0.0055900	0.0066838	0.0100750	0.0136697
2.4KB	637	0.0037878	0.0047766	0.0066142	0.0089003
2.8KB	494	0.0049447	0.0061052	0.0093335	0.0118719
2.9KB	522	0.0046819	0.0057774	0.0085796	0.0118833
3.2 KB	875	0.0039442	0.0050021	0.0073244	0.0100917
3.6 KB	836	0.0030186	0.0039818	0.0060983	0.0078264
5.0 KB	1044	0.0034360	0.0044670	0.0070494	0.0086779
5.6 KB	1672	0.0034302	0.0045152	0.0061954	0.0081521

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

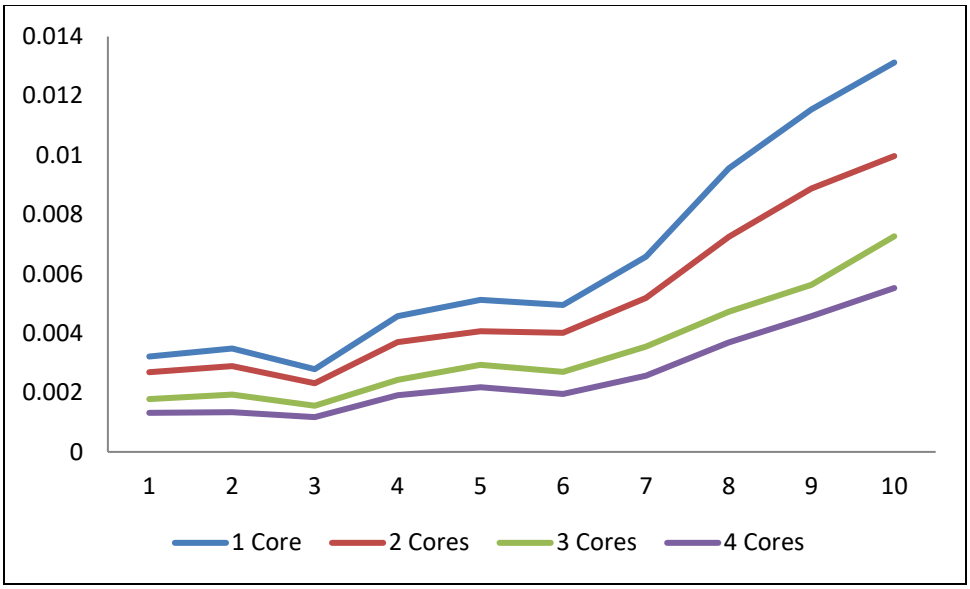

Fig 3: Graph of experiment number v/s time taken in sequential and parallel lexical analysis with increase in number of CPU cores. Horizontal axis represents experiment number and Vertical axis represents the execution time in seconds.

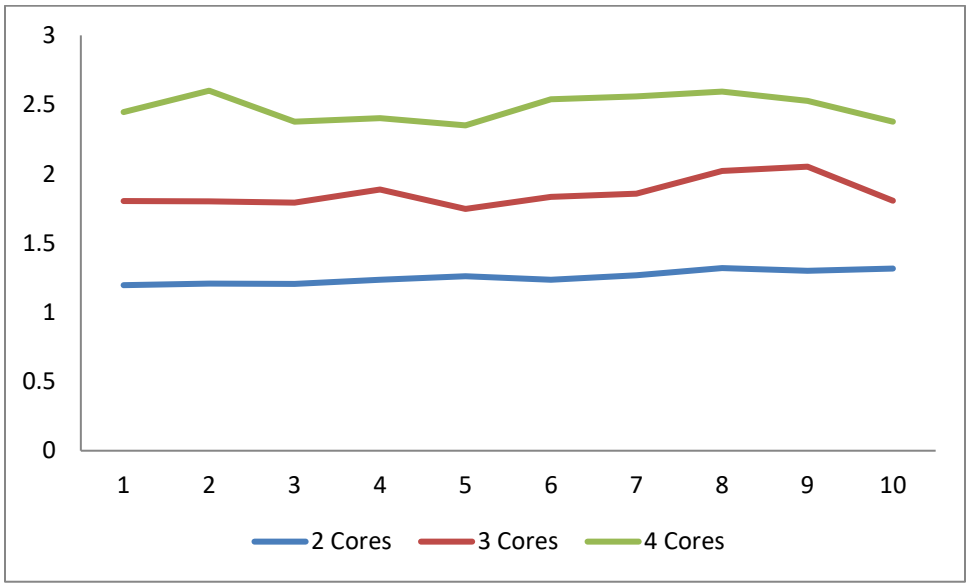

Fig 4: Graph of speedup efficiency in parallel lexical analysis for various numbers of CPU core. Horizontal axis represents experiment number and Vertical axis represents the speedup.

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

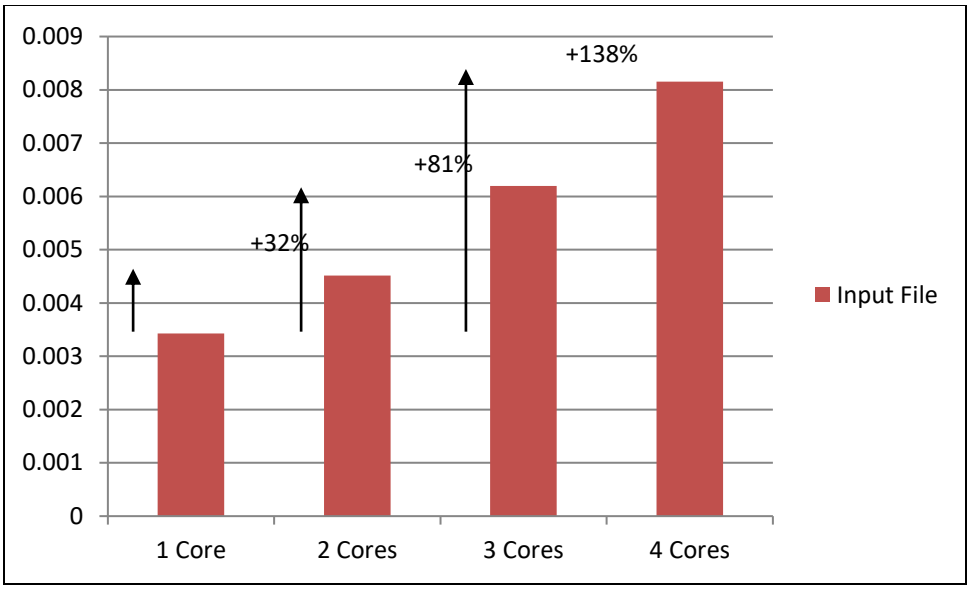

Fig 5: The data throughput of sequential lexical analyzer using a single core and parallel lexical analyzer using multiple cores over an input file of size 5.6 KB. Horizontal axis represents number of cores employed and Vertical axis represents the data throughput in Gbps.

V. CONCLUSION

This article discusses how to accelerate the lexical scanning process by allocating tasks line-by-line to unused cores to run in parallel on multi-core processors with concurrent keyword recognition. It also discusses the way to recognize the keywords without specifying the keyword patterns in LEX specifications by developing auto keyword detection method. As a result, the assigned core tokenizes the string and detects the keywords concurrently. The parallelization approach significantly enhances the lexical analyzer's performance. As per experimental results, when four processes and cores are used, the proposed methodology significantly outperformed the sequential approach by recognizing tokens in 0.0055230 seconds. The speedup of the parallel application is 2.3765405X. For processing the input file of size 5.6 KB, parallel lexical analyzer using 4 cores achieves 0.0081521 Gbps of data throughput, which provides a 138 percent improvement on sequential approach. When compared to the parallel process of lexical analysis line by line, this hybrid process is comparatively better. It is clearly observed that the speedup should increase at or close to the same rate as the number of cores increases. Implementations of this algorithm can be enhanced to generate lexical analyzers.

REFERENCES

1. Aho, A. V., Lam, M. S., & Sethi, R. (2009). Compilers Principles, Techniques and Tools, 2nd ed, PEARSON Education.
2. T. A. S. Foundation. Lucene, http://lucene.apache.org.
3. Nicola, M., & John, J. (2003, November). Xml parsing: a threat to database performance. In Proceedings of the twelfth international conference on Information and knowledge

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

management (pp. 175-178). ACM.
4. Perkins, E., Kostoulas, M., Heifets, A., Matsa, M., & Mendelsohn, N. (2005, November). Performance analysis of XML APIs. In XML Conf. and Exposition.
5. Pai T, V., & Aithal, P. S. (2020). A Systematic Literature Review of Lexical Analyzer Implementation Techniques in Compiler Design. International Journal of Applied Engineering and Management Letters (IJAEML), 4(2), 285-301.
6. Chen, Y. S., Liao, H. C., & Tsai, T. H. (2012). Online real-time task scheduling in heterogeneous multicore system-on-a-chip. IEEE Transactions on Parallel and Distributed Systems, 24(1), 118-130.
7. Herath, D., Lakmali, C., & Ragel, R. (2012, August). Accelerating string matching for bio-computing applications on multi-core CPUs. In 2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS) (pp. 1-6). IEEE.
8. Lin, C. H., Tsai, S. Y., Liu, C. H., Chang, S. C., & Shyu, J. M. (2010, December). Accelerating string matching using multi-threaded algorithm on GPU. In 2010 IEEE Global Telecommunications Conference GLOBECOM 2010 (pp. 1-5). IEEE.
9. Aho, A. V., & Corasick, M. J. (1975). Efficient string matching: an aid to bibliographic search. Communications of the ACM, 18(6), 333-340.
10. Villa, O., Chavarria-Miranda, D., & Maschhoff, K. (2009, May). Input-independent, scalable and fast string matching on the Cray XMT. In 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1-12). IEEE.
11. Watson, B. W. (1994). The performance of single-keyword and multiple-keyword pattern matching algorithms. Eindhoven University of Technology, Department of Mathematics and Computing Science, Computing Science Section.
12. Scarpazza, D. P., Villa, O., & Petrini, F. (2008, April). High-speed string searching against large dictionaries on the Cell/BE processor. In 2008 IEEE International Symposium on Parallel and Distributed Processing (pp. 1-12). IEEE.
13. Scarpazza, D. P., & Russell, G. F. (2009, June). High-performance regular expression scanning on the Cell/BE processor. In Proceedings of the 23rd international conference on Supercomputing (pp. 14-25).
14. Srikanth, G. U. (2010, June). Parallel lexical analyzer on the cell processor. In 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (pp. 28-29). IEEE.
15. Barve, A., & Joshi, B. K. (2015). Improved Parallel Lexical Analysis using OpenMP on Multi-core Machines. Procedia Computer Science, 49(1), 211-219.
16. Wang, Z., & O'Boyle, M. (2018). Machine learning in compiler optimization. Proceedings of the IEEE, 106(11), 1879-1901.
17. Yang, Y., Xiang, P., Kong, J., & Zhou, H. (2010). A GPGPU compiler for memory optimization and parallelism management. ACM Sigplan Notices, 45(6), 86-97.

ISSN: 0011-9342 | Year 2021
Issue: 7 | Pages: 14176-14191

18. Yao, X., Geng, P., & Du, X. (2013, December). A task scheduling algorithm for multi-core processors. In 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies (pp. 259-264). IEEE.
19. Wolfe, M. J. (1995). High performance compilers for parallel computing. Addison-Wesley Longman Publishing Co., Inc..
20. Paxson, V. (1995). Flex–Fast Lexical Analyzer Generator. Lawrence Berkeley Laboratory, Berkeley, CA.
21. Man7org. (2021). Man7org. Retrieved 2 January, 2021, from https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html
22. Aldea, S., Llanos, D. R., & González-Escribano, A. (2012). Using SPEC CPU2006 to evaluate the sequential and parallel code generated by commercial and open-source compilers. The Journal of Supercomputing, 59(1), 486-498.