A Parallel Scanner for the Concurrent Execution of Lexical Analyzer Tasks on Multi-Core Machines using Dynamic Task Allocation Algorithm

Vaikunta Pai T. [1*], Nethravathi P. S. [2] & P. S. Aithal [3]
[1]Faculty, College of Computer Science & Information Science, Srinivas University, Mangalore-575001, India.
ORCID: 0000-0001-6100-9023; Email: vaikunthpai@gmail.com
[2] Faculty, College of Computer Science & Information Science, Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0001-5447-8673; Email: nethrakumar590@gmail.com
[3] Faculty, College of Computer Science & Information Science, Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0002-4691-8736; Email: psaithal@gmail.com

Area/Section: Computer Science.
Type of the Paper: Experimental Research.
Type of Review: Peer Reviewed as per |C|O|P|E| guidance.
Indexed in: OpenAIRE.
DOI: https://doi.org/10.5281/zenodo.6415345
Google Scholar Citation: IJMTS

How to Cite this Paper:
Vaikunta, Pai T., Nethravathi, P. S., & Aithal, P. S., (2022). A Parallel Scanner for the Concurrent Execution of Lexical Analyzer Tasks on Multi-Core Machines using Dynamic Task Allocation Algorithm. *International Journal of Management, Technology, and Social Sciences (IJMTS), 7*(1), 245-253. DOI: https://doi.org/10.5281/zenodo.6415345.

International Journal of Management, Technology, and Social Sciences (IJMTS)
A Refereed International Journal of Srinivas University, India.

CrossRef DOI: https://doi.org/10.47992/IJMTS.2581.6012.0189

Received on: 28/03/2022
Published on: 06/03/2022

A Parallel Scanner for the Concurrent Execution of Lexical Analyzer Tasks on Multi-Core Machines using Dynamic Task Allocation Algorithm

Vaikunta Pai T. [1*], Nethravathi P. S. [2] & P. S. Aithal [3]
[1]Faculty, College of Computer Science & Information Science, Srinivas University, Mangalore-575001, India.
ORCID: 0000-0001-6100-9023; Email: vaikunthpai@gmail.com
[2] Faculty, College of Computer Science & Information Science, Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0001-5447-8673; Email: nethrakumar590@gmail.com
[3] Faculty, College of Computer Science & Information Science, Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0002-4691-8736; Email: psaithal@gmail.com

ABSTRACT

Purpose: *Massive multi-core architecture is rapidly becoming the standard in digital technology due to its high and parallel computational capability and performance benefits. To fully utilize the technological capabilities of multi-core systems, system software such as compilers should be re-engineered for parallel processing. Several important contributions have been made in the past to enhance the efficiency of the lexical analysis process by leveraging the innate parallel processing capability of multi-core computers. This trend of implementation shows that a parallel lexical analyzer tends to perform lexing tasks better than a conventional sequential lexical analyzer. This article discusses the way of making the tasks in parallel during the scanning of source program in the phase of lexical analysis. The objective of this study is to explore how to perform lexical analysis in parallel. On multi-core processors, multiple processes of the lexical analyzer program can run concurrently to scan multiple lines in the input stream in parallel for token detection. This is done by allocating tasks line-by-line to the core which is not engaged yet.*
Design/Methodology/Approach: *Developing a theoretical and experimental approach for parallelizing the lexical scanning process on a multi-core system.*
Findings/Result: *According to the theoretical and experimental results, the proposed methodology significantly outperforms the sequential approach in terms of tokenization. It considerably reduces the time required for lexical analysis during the compilation process. The result establishes unequivocally that the parallel lexical analyzer's performance should scale linearly with the number of cores. It is clearly observed that the speedup is expected to increase further if the number of CPU cores increases. This enhancement would speed up the compilation process even more.*
Originality/Value: *A dynamic task allocation algorithm is developed for the concurrent execution of a lexical analyzer task on multi-core systems.*
Paper Type: *Experimental Research.*
Keywords: Multi-core machines, Lexical analyzer, Task allocation

1. INTRODUCTION :

Phase one of the multi-phase compiler is referred to as scanning or lexical analysis, during which the input stream is scanned and tokenized [1]. A lexical analyzer, alternatively referred to as a lexer, is a pattern recognition engine that is simulated using a mathematical computational model called a Finite-State Machine (FSM) [2]. It reads a string of individual characters from the source code and clusters them into constructive sequences called lexemes using the token pattern. As an outcome, it generates a stream of tokens for syntax analysis. It is the most time-consuming operation and has a noticeable impact on the performance of a compiler. Modern processors have evolved to become increasingly multi-core in nature. Due to the fact that the majority of compilers are single-threaded, they are unable to take advantage of the resources available on modern computers. As a consequence, compiler performance has declined precipitously. According to studies [3], 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. This paper presents an efficient method for achieving the tasks in parallel, in the lexical analysis phase. This has been processed on multi-core processors. These processors tokenize the strings in parallel from the different lines of the source file. The process makes use of all available processing resources to achieve high performance in a compiler. The result demonstrates an improvement in performance in aspects of the overall time required to recognize the tokens during the compilation process. In comparison with sequential lexical analysis on a single processor system this will be faster. The remainder of the article is structured as follows. The Section 2 examines some seminal works in this field. Section 3 discusses the objectives of this work. Section 4 discusses an algorithm from a theoretical and experimental perspective. Section 5 evaluates the proposed algorithm using representative test cases and illustrates the performance improvement. Finally, Section 6 concludes.

2. RELATED WORK :

Numerous seminal works have been published in the past about determining which, tasks a compiler can perform in parallel. Mickunas et al. [4] identified for the first-time distinct areas within the compilation process that are amenable to parallel processing. They advanced a novel theoretical framework in which lexical analysis is divided into 2 levels: scanning as well as screening. They opined that text scanning could be performed concurrently. Daniele Paolo Scarpazza et al. [5] made significant contributions to the advancement of a parallel lexical analyzer. As an experiment, the original Flex kernel was optimized to execute on a multi-core cell processor. Umarani Srikanth [6] explored the possibility of parallelizing a lexical analyzer for cell processors. The approach is founded on segmenting the original source code into a predetermined number of chunks and performing lexical analysis tasks in parallel. The Aho-Corasick algorithm is used to recognize tokens in that experiment. When the parallel version of the lexical analyzer process is run on a system equipped with an IBM Cell Processor, a performance boost is observed. Amit Barve et al. [7] 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. The Round Robin scheduling algorithm is used to allocate tasks to various CPUs. The result shows a decrease in speedup due to a CPU being unable to obtain sufficient work after completing the current task assigned to it. Additionally, due to the single read head on disk, each processor in the I/O queue experiences a significant delay. Amit Barve et al. [8] tried to improve the method for parallel lexical analysis using the memory block. The buffer is used in this algorithm to store for-loop statements detected in source code via a pivot location file containing the starting and ending points of each for-loop. The tokens are recognized by processing each line in buffer in parallel using OpenMP programming.
In order to perform parallel lexical analysis, the research works presented in [6-8] divided original source code into fixed-size chunks based on certain criteria. The blocks are then distributed among the cores for parallel execution of lexical analysis. None of the experiments used a dynamic task allocation strategy. This article demonstrates how to accelerate a parallel lexical analyzer algorithm by allocating tasks line-by-line to unused cores on multi-core processors. As a result, none of the cores that are available are idle.

3. OBJECTIVES :

The main objective of this research is to design and develop theoretical and experimental approach for parallelizing the lexical scanning process on a multi-core system. The objectives of this research work have been

(1) To design and develop a dynamic task allocation algorithm for a parallel lexical analyzer on multi-core systems for achieving the tasks in parallel.
(2) To analyze the performance enhancements of the parallel lexical analyzer using a dynamic task allocation algorithm on multi-core systems with increased numbers of CPU cores.
(3) To compare the performance of a sequential lexical analyzer running on a single core to a parallel lexical analyzer running on a multi-core system utilizing a dynamic task allocation algorithm.

4. METHODOLOGY :

Is split into two parts:
(1) Theoretical approach to an algorithm
(2) Experimental approach

4.1 Theoretical approach to an algorithm:

In a multi-core system, let us consider that *K* numbers of cores are available. These *K* numbers of cores are working in parallel to process the lexical scanning in parallel. This is done without disengaging any of the processors. When each processor completes their process it will inform that it is idle now and consequently it gets the next work at the same time. In this work, it is called as dynamic allocation of the task to the processors. These processors are happening without a gap of work scheduling.
The above process can be explained theoretically as follows. Let *"k"* be the number of lines in the program which is to be processed for lexical analysis. Let us consider t_i as the time taken for *i*th line for the recognition of the token of that line. As said earlier, *K* is the number of cores of the processors. Let *T* be the total time taken for the recognition of the token for the full program, which is in consideration.
The time required for the complete lexical analysis depends on three parts.

1. Case 1: If the number of cores is greater than the number of lines in the program. ($k < K$)
2. Case 2: If both are equal ($k = K$)
3. Case 3: If the number of cores is less than the number of lines in the program. ($k > K$)

In the first case, it is requirement of number of cores are more than that of number of lines of the program. Hence, definitely the time taken for the longest line is the time taken for the whole process. Let us consider ith line is the longest line; therefore, the total time taken for the whole process is t_i. However, there are two more cases in this. They are

a. All the lines of the program are of equal length.
b. All the lines in the program are of different length.

In case 'a', all lines of the program is of equal length; in this case, the total time taken is the average time for the execution as per equation number (1) or the time taken for a single line for the execution.

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

Example 4.1: A program with 8 numbers of lines, each line having 6 tokens. This program is given to an octal core computer to evaluate the lexical analysis in parallel. The total time taken for this is the average time taken for the evaluation or time taken for a single line.
In the second case 'b', with all the lines of the program are of different lengths, then the simply it may be considered as the time taken for the line which takes the maximum time for the analysis.
Example 4.2: A program with 8 numbers of lines. This program is given to an octal core computer to evaluate the lexical analysis in parallel. The average time taken for recognizing token is 13ms. Line number 1 and 4, takes 11ms, 2 and 3 takes 12ms, 7 and 8 takes 14 ms, 5 and 6 takes 15ms. With the standard deviation of these values is 14.57ms. Total time taken for the evaluation of full program can be calculated as maximum time taken for the execution of a single line that is 15ms.
Again, in the second case, that is, case 2, also the same situation arrases since both, that is, the number of cores and the number of lines to be analyzed, are the same. Then the above explanation and set of equations hold good for this case also. Again, there are two cases: 'a' and 'b'. In case of 'a', the time taken is $\bar{t}$ or time taken for a single line, and in the case 'b', the total time taken is the time taken for the line which takes the maximum time for the execution.
In the third case, the number of lines in the program is more than the number of cores. Here all cores are to be working with its fullest extent. Since total number of lines to be executed is more than the core. There are two cases:

A. The number of lines in the program is integer multiples of the number of cores GCD(k,K) ≠1
B. The number of lines of program is not an integer multiple of numbers of core GCD(k,K) = 1

That is, as said earlier, let k be the number of lines in the program and K be the number of cores in the system. Then the case 'A' is defined as GCD (k,K) ≠1case 'A' is defined asGCD (k,K) = 1and , in both cases k > K.

Now define the total time taken by the ith core to analyze lexically is T_i. Therefore, the total time taken for lexical analysis of a given program is given for case A which can be handled as two more cases.

Ά. All the lines of the program are of equal length (as per the token is concerned)

β. All the lines of the program are of different lengths.

For the case Ά the total time taken is given by the equation (2) as follows

$$T = \bar{\bar{T}} = \frac{1}{K}\left[\sum_{i=1}^{K} T_i\right] \qquad (2)$$

That is an average of the time taken by the core to analyze the program.

Example for case Ά: Example 4.3: A program with 16 numbers of lines. This program is given to an octal core computer to evaluate the lexical analysis in parallel. The average time taken for recognizing token is 12ms for all lines. As per equation (1), the time taken can be evaluated as 24ms.

In the second case β, since the GCD (k,K) ≠1 there is a mismatch between the lines of program and the total number of cores. In this case, the total time taken by the cores is defined as

$$T = \frac{1}{K}\left\{\left[\sum_{i=1}^{K} T_i\right] + \left[\sum_{i=1}^{K} \sqrt{\bar{\bar{T}}^2 - T_i^2}\right]\right\} \qquad (3)$$

Example for case β: Example 4.4: A program with 16 numbers of lines. This program is given to an octal core computer to evaluate the lexical analysis in parallel. The average time taken for recognizing token is 13ms. Line number 1, 4, 6 and 14 is takes 11 ms, 2, 3, 5 and 13 is takes 12ms, 7, 9, 10 and 16 is takes 14 ms, 8, 11, 12 and 15 is takes 15ms. With the standard deviation of these values is 10.303ms. The total time taken for the evaluation of the full program can be calculated by adding these two values, which is average and standard deviation. Hence approximately, will get, the total time taken for the evaluation of the full program that is 23.303ms. This includes an average of 13ms and a standard deviation of 10.303ms.

4.2 Experimental approach:

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 [9]. 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. The line-by-line task allocation to the cores that are not currently engaged can be accomplished by the following algorithm.

Algorithm DYNAMIC TASK ALLOCATION. Given source file SOURCE written in C programming language, this algorithm allocates tasks to the core that are not currently in use, in a line-by-line fashion. As a result, assigned core concurrently tokenizes string and places generated tokens in global table. The array BUFFER is used to store source file line by line and each read line is stored in variable LINE. The number of available cores is stored in variable CPU_COUNT and core that are not currently engaged are stored in array CPU_QUEUE. The variables FRONT and REAR are used to process the queue. The variable C_FLAG indicates child process's completion flag and variable PA_FLAG represents processor affinity.

```
Input:  Source file say source.c
Output: Parallel token generation
Method:
1.  [Open file]
        Open SOURCE file in READ mode for input
2.  [Loop to read each line from source file]
        Repeat while not end of SOURCE file
```

```
          Read from SOURCE file into BUFFER
3.  [Close file]
        Close SOURCE file
4.  [Initialize cpu core counter]
        CPU_COUNT ← number of available cores
5.  [Initialize cpu queue to available cores ]
        Repeat for i = 1, 2, .., CPU_COUNT
            CPU_QUEUE[i] ← i
6.  [Initialize front and rear variables]
        FRONT ← 1
        REAR ← CPU_COUNT
7.  [Process each line from the buffer and do the following in parallel]
        Repeat while not end of BUFFER
            Read from BUFFER into LINE
             if CPU_COUNT ≠ 0 (check cores are free?)
             then    PA_FLAG ← CPU_QUEUE[FRONT]
                     Repeat for i = 1, 2, ..,, REAR-1 (Shift all the available cores in cpu queue
                         CPU_QUEUE[i] ← i + 1      from index 2 till rear to the left by one)
                     REAR ← REAR – 1
                     CPU_COUNT ← CPU_COUNT – 1
                       Pi ← fork()  (Create a child process to run instance of LEX program)
                     sched_setaffinity(Pi, PA_FLAG)   (function to set processor affinity)
                     Set C_FLAG ← 0 (to indicate core has assigned a task)
                     yy_scan_string(LINE)  (function to scan the line)
                     yylex()  (function to tokenize the line and  stores tokens in global table)
                     Set C_FLAG ← 1 (to indicate core has completed assigned task)
                     if C_FLAG =1
                     then    CPU_QUEUE[REAR]= PA_FLAG
                               CPU_COUNT ← CPU_COUNT + 1
             else    Repeat from previous LINE from BUFFER (when cores are not available)
8.   [Finished]
    Exit
```

5. EXPERIMENTAL RESULTS :

The above algorithm has been tested on a machine equipped with an Intel Core i5-9400F 6-Core 2.90 GHz Processor and 8GB RAM. The test is conducted on a machine with operating system Ubuntu 18.04.5 LTS installed. 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 CPU affinity. Whereas the conventional LEX tool generated sequential lexical analyser 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 Figure 1.
The result establishes unequivocally that the parallel lexical analyzer's performance should scale linearly with the number of cores. The effect on speedup using sequential and parallel lexical analyzer is shown in table 2. Figure 2 illustrates the performance enhancements associated with increasing the number of CPU cores. The speedup is defined as the ratio of serial to parallel execution time, which is calculated using the following equation (4).

$$\text{Speed up} = \frac{\begin{array}{c}\text{Time taken for}\\ \text{sequential lexical analysis}\\ \text{using a single core}\end{array}}{\begin{array}{c}\text{Time taken for}\\ \text{parallel lexical analysis}\\ \text{using multiple cores}\end{array}} = \frac{\begin{array}{c}\text{Time taken for}\\ \text{sequential lexical anaysis}\\ \text{using a single core}\end{array}}{\begin{array}{c}\text{Time taken for}\\ \text{parallel lexical analysis}\\ \text{using 4 cores}\end{array}} = \frac{0.0131256}{0.0056938} = 2.3052443\text{X} \qquad (4)$$

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

Experiment Number	File Size (Lines)	No. 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	0.0027752	0.0018410	0.0013569
2	78	499	0.0034874	0.0029786	0.0019969	0.0013826
3	80	333	0.0027882	0.0023855	0.0016040	0.0012090
4	130	494	0.0045738	0.0038190	0.0024980	0.0019639
5	138	637	0.0051238	0.0041888	0.0030250	0.0022480
6	149	522	0.0049501	0.0041356	0.0027848	0.0020106
7	173	875	0.0065838	0.0053520	0.0036551	0.0026528
8	174	836	0.0095566	0.0074690	0.0048768	0.0038000
9	298	1044	0.0115459	0.0091558	0.0058018	0.0047130
10	348	1672	0.0131256	0.0102800	0.0074921	0.0056938

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	1.1597723	1.7482890	2.3720245
2	78	1.1708185	1.7464069	2.5223492
3	80	1.1688116	1.7382793	2.3062035
4	130	1.1976434	1.8309848	2.3289373
5	138	1.2232143	1.6938182	2.2792705
6	149	1.1969484	1.7775424	2.4620014
7	173	1.2301570	1.8012640	2.4818305
8	174	1.2795019	1.9596047	2.5148947
9	298	1.2610476	1.9900548	2.4497984
10	348	1.2768093	1.7519254	2.3052443

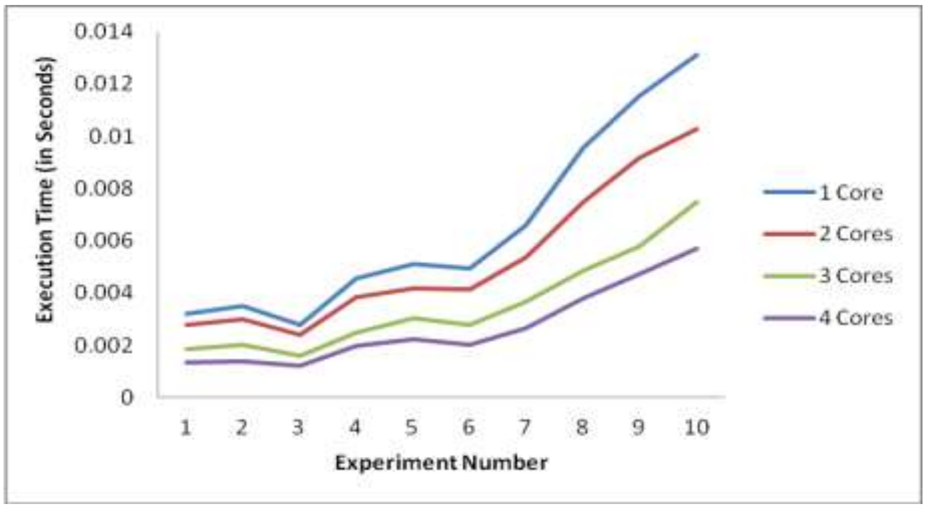

Fig. 1: Graph of experiment number v/s time taken in sequential and parallel lexical analysis with a different number of CPU cores

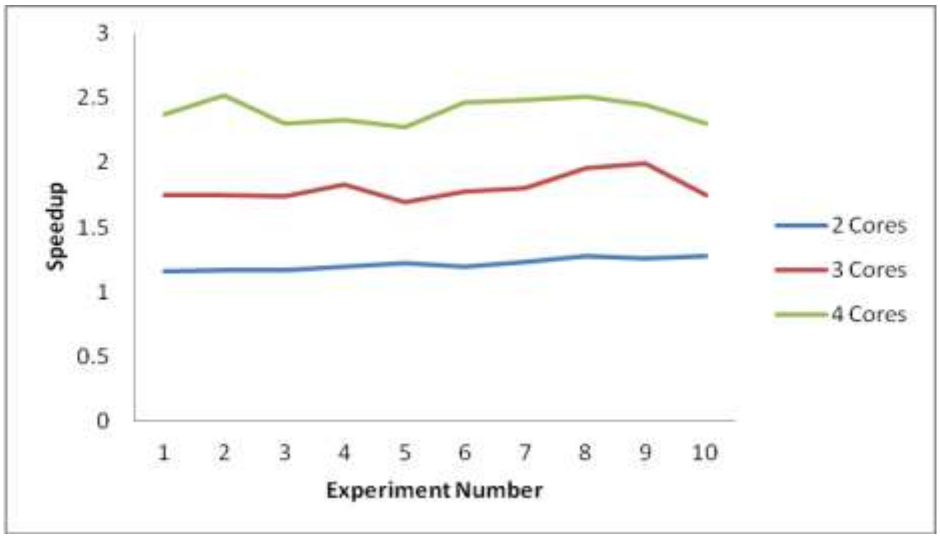

Fig. 2: Graph of speedup efficiency in parallel lexical analysis for various numbers of CPU cores

6. CONCLUSION :

This article discusses how to perform tasks in parallel while scanning the source code for the lexical analysis phase. The proposed dynamic task assignment algorithm assigns tasks line-by-line to available multiple cores that are not currently in use. As a result, the assigned core tokenizes the string concurrently. The parallelization approach significantly enhances the lexical analyzer's performance, as demonstrated by the above result analysis statistics. As per experimental results, when four processes and cores are used, the proposed methodology significantly outperformed the sequential approach by recognizing tokens in 0.0056938 seconds. The speedup of the parallel application is 2.3052443X. 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. 1-993.

[2] Paxson, V. (1995). *Flex–Fast Lexical Analyzer Generator*. Lawrence Berkeley Laboratory, Berkeley, CA.

[3] Vaikunta Pai T., Jayanthila Devi A., & 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. DOI: http://doi.org/10.5281/zenodo.4454632.
Google Scholar↗

[4] Mickunas, M. D., & Schell, R. M. (1978, December). Parallel compilation in a multiprocessor environment. *In Proceedings of the annual conference of the ACM, Washington, D.C., USA*. 241-246.
Google Scholar↗

[5] 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).
Google Scholar↗

[6] 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*, 28-29. IEEE.
Google Scholar↗

[7] Barve, A., & Joshi, B. K. (2012, September). A parallel lexical analyzer for multi-core machines. *In 2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, 1-3. IEEE. Google Scholar↗

[8] Barve, A., & Joshi, B. K. (2015). Improved Parallel Lexical Analysis using OpenMP on Multi-core Machines. *Procedia Computer Science, 49*(1), 211-219. Google Scholar↗

[9] Vaikunta Pai, T., Nethravathi, P. S., & Aithal, P. S. (2022). Improved Parallel Scanner for the Concurrent Execution of Lexical Analysis Tasks on Multi-Core Systems. *International Journal of Applied Engineering and Management Letters (IJAEML)*, *6*(1), 184-197. DOI: https://doi.org/10.5281/zenodo.6375532. Google Scholar↗

[10] Hall, M., Padua, D., & Pingali, K. (2009). Compiler research: the next 50 years. *Communications of the ACM, 52*(2), 60-67. Google Scholar↗

[11] Halstead, R. J., Villarreal, J., & Najjar, W. A. (2014). Compiling irregular applications for reconfigurable systems. *International Journal of High Performance Computing and Networking*, *7*(4), 258-268. Google Scholar↗

[12] Man7org. (2021). Man7org. Retrieved 2 January, 2021, from https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html

[13] 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. Google Scholar↗

[14] 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*, 1-12. IEEE. Google Scholar↗

[15] 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*, 1-5. IEEE. Google Scholar↗

[16] Wang, Z., & O'Boyle, M. (2018). Machine learning in compiler optimization. *Proceedings of the IEEE*, *106*(11), 1879-1901. Google Scholar↗

[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. Google Scholar↗

[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*, 259-264. IEEE. Google Scholar↗

[19] Sinya, R., Matsuzaki, K., & Sassa, M. (2013, October). Simultaneous finite automata: An efficient data-parallel model for regular expression matching. *In 2013 42nd International Conference on Parallel Processing*, 220-229. IEEE. Google Scholar↗

****