IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Ingénierie des Systèmes d'Information
Vol. 31, No. 7, July, 2026, pp. 2425-2434
Journal homepage: http://iieta.org/journals/isi

Parallel Optimization of the SSABS Exact String-Matching Algorithm for Shared-Memory Multicore Systems Using POSIX Threads

Atheer Akram AbdulRazzaq[1*], Ahmed Eskander Mezher[1,2], Ahmed Sabah Ahmed AL-Jumaili[1], Huda Kadhim Tayyeh[3]

[1] Department of Business Information Technology, Business Informatics College, University of Information Technology and Communications, Baghdad 10045, Iraq
[2] Department of Information Technology, Awang Had Salleh Graduate School of Arts and Sciences, Universiti Utara Malaysia (UUM), Sintok 06010, Malaysia
[3] Department of Informatics Systems Management, Business Informatics College, University of Information Technology and Communications, Baghdad 10045, Iraq

Corresponding Author Email: athproof@uoitc.edu.iq

https://doi.org/10.18280/isi.310724

Received: 5 March 2026
Revised: 30 June 2026
Accepted: 19 July 2026
Available online: 31 July 2026

Keywords:
Sheik-Sumit-Anindya-Balakrishnan-Sekar exact string-matching algorithm, Pthreads, database types, parallel execution time, speedup, efficiency, algorithm scalability

ABSTRACT

Exact string-matching remains a fundamental operation in many data-intensive applications, where increasing dataset sizes require efficient parallel processing solutions. This study presents a parallel optimization of the Sheik-Sumit-Anindya-Balakrishnan-Sekar (SSABS) exact string-matching algorithm for shared-memory multicore systems using the Portable Operating System Interface (POSIX) threads framework. The proposed implementation divides input text into independent segments with boundary overlap handling to preserve matching accuracy while enabling concurrent execution across multiple CPU cores. The performance of the parallel SSABS algorithm was evaluated using four benchmark datasets, including Extensible Markup Language (XML), Source, Protein, and DNA, with different pattern lengths and core configurations. Experimental results demonstrate that the parallel implementation consistently reduces execution time compared with the sequential version across all tested datasets. The source dataset achieved the shortest parallel execution time due to its larger alphabet size and more effective shift operations; the DNA dataset obtained the highest scalability performance, reaching a maximum speedup of 2.97 and the highest efficiency among the evaluated datasets. Further analysis revealed that increasing the number of cores improves acceleration but introduces synchronization and thread-management overhead, leading to performance saturation at higher core counts. These findings provide insights into the relationship between dataset characteristics, algorithm behavior, and multicore scalability, demonstrating the suitability of Pthreads-based parallelization for accelerating exact string-matching operations.

1. INTRODUCTION

String-matching algorithms are algorithms used to determine the most suitable arrangement by contrasting and finding the correlation between a set of patterns and a random string [1-3]. During the comparison phase, the pattern length must equal the length of the window text. Furthermore, the pattern and text window string patterns rely on the ascertainment of their fit [4]. String-matching remains a major problem faced by multitudinous computer science applications like intrusion detection systems (IDS) search analysis [5], natural language processing, information retrieval [6, 7], artificial intelligence (AI), signal and image processing [8, 9], pattern recognition [10], web search engines [11], and operating systems [12]. String-matching is also used in the analysis of DNA patterns [13, 14] and protein sequences [15].

The world is witnessing swift database development, which emphasizes the significance of performance enhancement of exact string-matching algorithms. The brute force algorithm is perceived as the simplest algorithm technique in comparison with alternative string-matching algorithms. This is due to its ability to scan a text pattern with a text substring via left-to-right orientation. In the presence of a match or a mismatch, precise shifts take place one position to the right; hence, the brute force algorithm operates in $O(mn)$ time. Furthermore, the Boyer-Moore algorithm is a widely used algorithm due to its high efficiency; it initiates the comparison from right to left. In the presence of a match or mismatch between the pattern and the text window, the good suffix and the bad character functions greatly impact the shifting [16].

The analysis of the Boyer-Moore algorithm and the Aho-Corasick algorithm has resulted in the Commentz-Walter algorithm. Pre-processed phased patterns utilized for text-matching are the fundamentals for a state machine, with the application of the Boyer-Moore extraction in the searching stage. Notably, the matching window length must be equal to the minimum pattern length. For the pattern characters, scanning is conducted from right to left. In circumstances

where there are unpredictable irregularities, a pre-calculated shift table is used to shift the window position to the right. The algorithm's time complexity is in the order of $O(\pi + m)$ for the pre-processing phase, and $O(n*m)$ for the searching phase [17].

The Aho-Corasick algorithm uses automata, unlike the Knuth-Morris-Pratt algorithm; the state machine pattern is founded on a tree formation. At the beginning, the tree has an empty root; the states are formed when pattern matching occurs. The state machine is traversed during the search stage in anticipation of a perfect matching state probability. The time complexity is in the order of $O(nk + m)$ for the preprocessing phase, and $O(n + m + z)$ for the searching phase [18].

The Fast-On-line Hybrid Matching Algorithm (FOHM) is dual-phased; the preprocessing phase involves the creation of the Quick Search Bad Character (qsBc) table, while the search phase comprises a three-step division pattern, followed by the analysis of the three steps through the pattern and text window comparison. The text window is then compared with the pattern's first three characters, followed by the comparison of the last three characters. If a match is found, the algorithm proceeds with the comparison of the remaining characters found inside the pattern and text window. The technique shifts the window to the right by an amount derived from the qsBc table in the event of a nil match at any phase [19].

The exact string-matching algorithms are affected by the number of attempts, the number of character comparisons, and the time consumed. The downsides of one or more of these factors are influenced by the efficacy of the algorithms. The decrease in the quantity of attempts and the quantity of character comparisons in sequential performance is enabled by a preponderance of the string-matching algorithms, albeit with a weakness in consumed time. Hence, researchers have emphasized the importance of focusing on the time consumption issues by utilizing parallel processing due to its capability in reducing time consumption in exact string-matching algorithms [4]. Parallelism is dependent on the complexity and cooperation amongst the processors or computer cores to address the sequential computer issues. These parallel applications rely on a pair of approaches: data decomposition and function decomposition. Shared memory and distributed memory are the constituents of the parallel computer architecture. Therefore, the techniques used are categorized by the method they use to access memory. In the first method that uses the shared memory technique, the processors only utilize one access method to access the common memory. This shared memory technique is utilized to keep data, in addition to various functions like synchronization and communication purposes between the processors [17, 20].

2. RELATED WORK ON PARALLEL TECHNIQUES FOR STRING-MATCHING ALGORITHMS

String-matching algorithms are mostly used to achieve optimum outcomes during the handling of various data sets; they are also used for resolving issues related to computer applications. Currently, numerous issues are surfacing, demonstrating the inadequacies in sequential algorithms' efficacy in achieving large-scale procedures during the handling of enormous datasets, coupled with the prolonged computation time of the said algorithms. Thus, these issues have a negative impact on parallel processing, specifically those connected to large databases [21]. Various exact string-matching algorithms have been used in parallel interface applications to address the numerous issues associated with these algorithms. Certain algorithms, like the Quick-Search algorithm, utilize multi-core technology; it employs two parallelization paradigms, which are OpenMP and Pthreads, respectively [22]. The speed of the Quick-Search algorithm has been boosted in the field of intrusion detection to achieve a faster and more effective detection system.

The Knuth-Morris-Pratt (KMP) algorithm employs multi-core and multiprocessor technologies for its queries in large-sized strings. The algorithm divides these strings into segregated sections by executing the procedures for matching between the pattern and individual sections. The KMP utilizes the single instruction, multiple data (SIMD) model and employs a number of data for the parallelization process. The KMP algorithm excelled in addressing large data sizes and identifying the appropriate number of processors [23]. Additionally, the KMP demonstrated a superior execution time compared to sequential execution, showing good speedup and efficiency when addressed with just two clusters (workstations). However, the efficiency was poor when the number of utilized clusters exceeded two due to the higher communication time [24].

The Karp-Rabin algorithm employed by the graphics processing unit (GPU) parallel model is dependent on the segregation of the text into sections. The process required a collation procedure to initiate a process that segregates each section of a text with a pattern individually. A parallelized version of the Karp-Rabin algorithm demonstrated significantly faster processing time; the processing time was three times swifter when processing long text strings compared to the sequential form [25]. Advanced parallel architectures have been heavily used in recent exact string-matching developments. Researchers have used SIMD vectorization for concurrent processing of multiple characters on CPU registers [26]; some studies have also offloaded workloads onto GPUs for the maximization of massive thread-level parallelism [27]. Memory-bound constraints across heterogeneous systems [28] have recently been analyzed to support these shifts in hardware. Bit-representations have also been optimized for the reduction of loop overhead in multi-core CPU cache hierarchies [29].

This study aims to complement these frameworks by introducing an optimized multi-threaded parallelization of the Sheik-Sumit-Anindya-Balakrishnan-Sekar (SSABS) algorithm using the Portable Operating System Interface (POSIX) threads (Pthreads) on general-purpose multi-core CPUs. Table 1 clarifies that previous parallel research mainly depended on high-level compiler abstractions and has undergone severe degradation of scalability over clusters that are network-bound multi-node clusters, or concentrated on one data structure only. The execution efficiency of parallelized string-matching algorithms is controlled by a suitable parallel programming framework; the most predominant paradigms are OpenMP, Message Passing Interface (MPI), and Pthreads. The structural compromises of these frameworks, as related to issues of exact string-matching, are outlined in Table 2.

Table 2 indicates that MPI is ideal for multi-node clusters processing multi-gigabyte files, but is subject to severe communication penalties when addressing pattern overlap segments ($m - 1$). Similarly, the OpenMP high-level directives lack the necessary grain control to handle the hardware-level cache locality demands triggered by data-dependent, irregular shifting behaviors inherent to the SSABS algorithm.

Table 1. Comparative summary of parallel string-matching literature

Reference	Core Algorithm	Parallel Framework	Dataset Type or Scope	Maximum Performance or Speedup	Core Limitation Addressed in Literature
Hnaif et al. [22]	Quick-Search	OpenMP & Pthreads	Intrusion Detection Strings	High-speed pattern filtering	Abstract high-level evaluations; missed data-dependent shift influences
Rasool and Khare [23]	KMP	SIMD multi-core	Large Text Strings (251 MB)	The algorithm demonstrates a 4.33× multi-core speedup	The algorithm has resolved the missed patterns at the connection boundaries by using overlapping string connectors
Abu-Zaid and El-Rayyes [24]	KMP	Distributed workstations	Multi-workstation cluster data (The data of Arabic Strings)	Good speedup on exactly two clusters	Efficiency collapses on >2 clusters due to high network communication costs.
Shah and Oza [25]	Karp-Rabin	NVIDIA CUDA (GPU)	Large datasets	High parallel efficiency, with increased speedup	It removes sequential delays by splitting the text
This Study (Proposed)	SSABS	Pthreads (Low-level)	XML, Source, Protein, DNA	Up to 2.97× speedup (See Section 4)	Isolates alphabet-size paradox and hardware cache-locality limits

Note: KMP = Knuth-Morris-Pratt, SIMD = single instruction multiple data, XML = Extensible Markup Language, SSABS = Sheik-Sumit-Anindya-Balakrishnan-Sekar, GPU = graphics processing unit.

Table 2. Qualitative architectural comparison of parallel computing frameworks for exact string-matching

Framework	Memory Architecture	Overhead Level	Boundary Management Handling	Suitability for SSABS
MPI	Distributed	High (network latency)	High overhead due to message passing across nodes	Poor (network latency cancels out fast qsBc shifts)
OpenMP	Shared	Medium (compiler-managed)	Coarse-grained loop partitioning; rigid thread control	Fair (easy implementation, but sacrifices fine cache tuning)
Pthreads (This Study)	Shared	Low (explicit or fine-grained)	Direct pointer manipulation; localized cache scheduling	Excellent (maximizes L1 or L2 cache locality for irregular shifts)

Note: MPI = Message Passing Interface, SSABS = Sheik-Sumit-Anindya-Balakrishnan-Sekar.

The increasing rate of database development requires enhanced performance of exact string-matching algorithms. The novelty of this implementation is clarified by isolating and analyzing the architectural and data-dependent complexities of parallelizing the SSABS algorithm in shared-memory multi-core environments. The SSABS algorithm parallelization is uniquely non-trivial; its shifting mechanism relies on a qsBc preprocessing heuristic that enables long right-to-left shift distances sequentially. However, its interaction with low-overhead Pthreads changes dramatically depending on the dataset alphabet size. This study made the following main unique contributions:

1. Demonstration of an empirical paradox where smaller-alphabet datasets (DNA) underwent more severe bottlenecks of sequential execution, but exhibited peak parallel speedup scalability (2.97) and peak core efficiency (74%) across multi-core systems.
2. Mathematical analysis of the thread boundary constraints ($n/p + m - 1$); the exact point of system inflection (transitioning to eight cores) that marked the costs of hardware communication and the beginning of parallel resource efficiency cannibalization by context-switching overhead was identified.
3. Direct mapping of thread behavior onto modern multi-core CPU hardware cache topologies through fine-grained, low-level Pthread synchronization constructs; this approach is contrary to other high-level abstractions, such as OpenMP.

In this study, the exact string-matching algorithm (SSABS algorithm) was reconstructed using a parallel model to minimize the execution time and improve execution time. In this case, the performance of the algorithm was evaluated using various parameters, such as on different databases, number of threads, and number of cores. Section 3 details the algorithm and its implementation, specifically the SSABS algorithm method. Subsection 3.1 explains the Sequential SSABS Algorithm; subsection 3.2 describes the parallel SSABS algorithm using the Pthreads technique, and the generation of Pthreads code for the SSABS algorithm. Subsection 3.3 details the parallel implementation of the proposed algorithm; Subsection 3.4 describes the employed performance metrics; Subsection 3.5 details the theoretical performance and complexity analysis; Subsection 3.6 provides the experiment design. Section 4 covers the results and discussions, while Section 5 concludes the study.

3. THE PROPOSED METHOD

The fundamental characteristics and operational behavior of the SSABS algorithm are analyzed in this section. The principal aim is to determine the computationally intensive code components via the examination of the SSABS algorithm's sequential version that includes the preprocessing and searching stages. The overall improvement and algorithm efficiency are achieved through understanding of these crucial performance aspects, as they are the enablers of efficient parallelization using Pthreads.

3.1 Sequential SSABS algorithm

The SSABS algorithm is a string-matching algorithm that combines the Boyer–Moore and Sunday's Quick Search heuristics. Its main objective is to decrease the number of shifts and optimize the average shift distance during pattern searching. During the preprocessing stage, the SSABS formulates the qsBc table, which delegates a shift value to individual alphabet characters. The value represents the character distance from the pattern terminal, or $m + 1$ if lacking in the pattern. The qsBc offers constant, favorable, and typically larger shift values, which are independent of the comparison order; this allows for faster and more predictable searching in comparison to the Boyer–Moore bad-character rule.

Throughout the searching stage, the SSABS performs pattern arrangements and matching with a text segment; it conducts comparisons mainly from the right-to-left positions. Firstly, the terminal pattern character and the window are compared; if there is a match, the first pattern character and the corresponding window character are contrasted to ascertain the main similarities between the pattern and the window. But if there is no match in the characters, the residual characters are contrasted successively. The algorithm uses the qsBc table to calculate the needed extent to move the pattern; this extent is based on the character immediately following the current text window once there is a match or mismatch. This procedure is repeated until the text is scanned [30].

3.2 Parallel Implementation of Sheik-Sumit-Anindya-Balakrishnan-Sekar algorithm using Pthreads

The threads are implemented autonomously using available implementation components that function inside a shared memory space; hence, it is an effective method for addressing simultaneous functions. These threads are widely used in shared-memory multiprocessor and multi-core systems to achieve true parallelism performance. The Pthreads library provides a harmonized and limited *C* program thread management interface on UNIX-based platforms following the IEEE POSIX 1003.1c standard. This library enables multiple-thread generation and coordination within an individual procedure, supporting the realization of an effective parallel computation. The attainment of parallelism in UNIX environments is enabled through the fork system call, which generates a new process that operates autonomously from its parent. Nevertheless, experimental studies have demonstrated that Pthreads provide outstanding performance in comparison to fork-based parallelism. The value-added edge stems from the substantially reduced operating system overhead that is linked to the generation of threads and context switching due to the sharing of the same address space; this is contrary to autonomous processes produced by fork.

The implementation of the parallel SSABS algorithm is achieved via a succession of explicit stages that align thread performance and data processing. The initial step involves the formation and management of threads, and the preparation for parallelism. A determined peak quantity of threads is specified, with individual threads assigned a specific exclusive identifier from 0 to $P - 1$, where P signifies the total number of active threads. This was followed by the creation of threads and the initiation of dedicated thread tasks. Before the implementation, mutex mechanisms are activated using a predetermined static initializer to regulate access to shared variables. Mutex lock and unlock operations are employed to guarantee that only one thread can access shared data at any instance, thus averting race conditions. Furthermore, control functions are employed for execution synchronicity by awaiting the completion of a designated thread or all threads before advancing further.

In the second stage, the first step is setting up important basic input parameters and their assignment to several threads. Global access is provided to the core SSABS algorithm variables involving text length (n), text array (y), pattern length (m), and pattern array (x). The algorithm's preprocessing function is conducted simultaneously, producing heuristic structures such as the qsBc table; this enables effective search window changes throughout the pattern matching process.

In the third stage, the input text is split into smaller sections to achieve parallel processing. A designated thread is allocated to each section of the text. Each segment's size is expanded to incorporate an overlap based on the pattern length determined as $n/p + m - 1$; this is aimed at preventing missing pattern matches at the segment edges. If synchronized access to shared data is required, mutex lock and unlock procedures are employed, and threads process their allotted segments concurrently. Each thread individually conducts the SSABS search on the text segment assigned to it in the final stage using the identical pattern and shift rules adopted in the Quick Search method. This enables the threads to find precise pattern matches effectively. Each thread tracks local performance metrics during the query, such as the number of character comparisons, the overall matching attempts, and the segments' processing time. The final stage combines the outcomes from the search trajectory of each thread. The partial outputs from each thread, including execution times and performance metrics, are merged using a reduction mutex; this merging creates a global number(s) that signify the general performance of the parallel SSABS algorithm. The mutex is released after the aggregation to guarantee appropriate synchronization and to preserve data integrity. This phase, as depicted in Figure 1, offers an extensive assessment of the algorithm's scalability and efficiency.

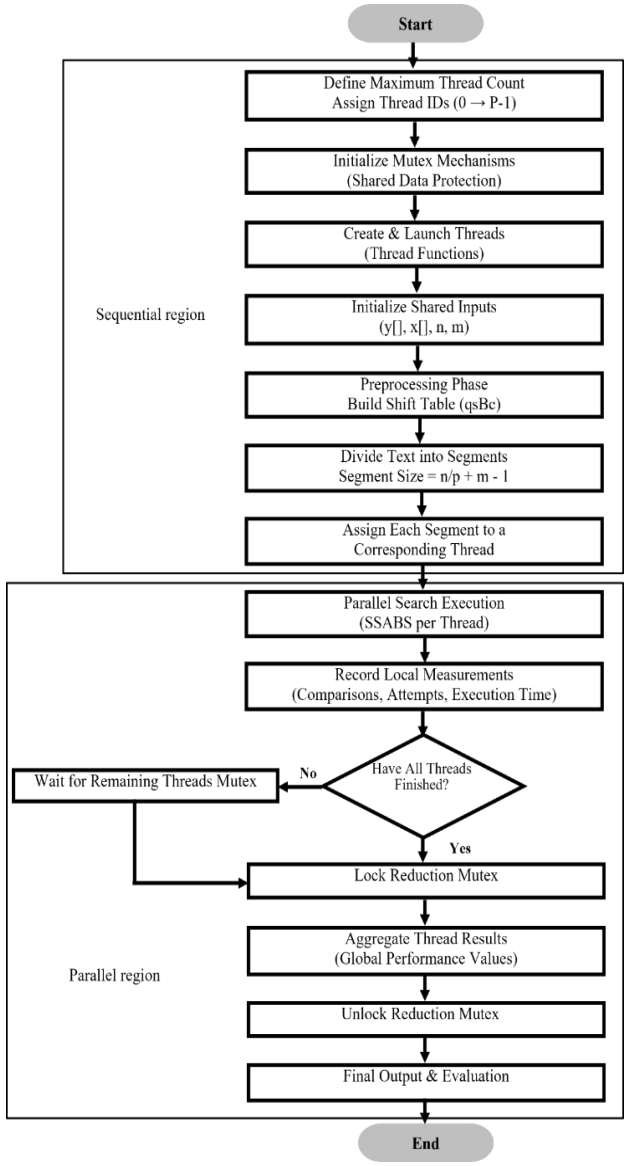

Figure 1. Flowchart of the parallel execution of the Sheik-Sumit-Anindya-Balakrishnan-Sekar (SSABS) algorithm using Pthreads

3.3 Parallel implementation of the proposed algorithm

A personal computer with an Intel® Core™ i7-6600U CPU, 8 GB of RAM, and a 256 GB SSD, running Ubuntu 18.04.6 LTS under Windows 10 through WSL2, was used to assess the parallel SSABS algorithm. GCC 7.5.0 was used to compile the Source code with optimization flags for native Pthreads. The WSL2 environment functions through a lightweight Type-1 hypervisor virtual machine, but has a structurally insignificant effect on multi-threaded performance. WSL2 differs from virtualization by running a native Linux kernel wherein execution streams can be mapped by the Pthreads library directly onto physical host CPU cores. The execution paths avoided virtualization translation overhead since the parallel SSABS implementation is strictly CPU-bound and functions wholly inside pre-allocated RAM buffers and bypasses virtual disk I/O bottlenecks. Thus, the virtualization layer introduced negligible noise latency, which ensures that the trends for empirical speedup and efficiency continuously reflect the performance of the native hardware.

3.4 The performance metrics

The parallel version of the suggested algorithm was achieved using Pthreads. Several evaluation metrics, such as execution time, speedup, and efficiency, were used to benchmark the performance of the parallel implementation with the sequential counterpart [31]. The overall time needed to complete a task is termed the execution time. It represents the time span needed for a single processor to run the algorithm from the beginning (initialization) to its termination (completion), represented as T_s. The execution time (T_p) for the parallel version is the time span from the start of the first thread to the end of the final thread's execution.

The enhanced performance derived from parallel execution is quantified using speedup; it is defined as the ratio of the sequential execution time to the parallel execution time (where S is the calculated speedup, T_s is the time spent in the sequential version, and T_p is the time spent in the parallel version). Eq. (1) is used for time calculation (measured in milliseconds).

$$Speedup(S) = \frac{T_s}{T_p} \tag{1}$$

Efficiency refers to the effective utilization of available processing cores during parallelism; it is calculated as the ratio of the speedup attained to the total number of cores in totality used. Improved resource use with less thread idle time demonstrates higher efficiency. Efficiency is calculated using Eq. (2), where P is the number of processing cores, and S is the speedup.

$$Efficiency(E) = \frac{S}{P} \tag{2}$$

3.5 Theoretical performance and complexity analysis

The scalability constraints of the multi-core SSABS application were clarified by modelling its execution profile using formal parallel complexity theory.

3.5.1 Parallel time complexity

The SSABS algorithm requires a preprocessing step to build the qsBc shift table; this requires $O(m + \sum)$ time, where m is the pattern length, and the alphabet size is indicated by $\sum$. The parallel model executed this phase sequentially by the master thread to circumvent redundant table allocations. For the searching phase, the length of the text n was split among p processing cores. Given overlaps of pattern boundaries, the chunk size processed by each core was exactly $\frac{n}{p} + m - 1$ characters. Assuming that the load was equally distributed across all threads, the parallel execution time complexity per thread is bounded as follows:

$$O\left(\left(\frac{n}{p} + m - 1\right).m\right) \tag{3}$$

In the best-case shifting scenario (large alphabet sizes, such as the Source dataset), the inner pattern matching cost is decreased, and the search bounds per core are reduced to $O(\frac{n}{p.m})$.

3.5.2 Theoretical speedup bounds ('Amdahl's Law')

Amdahl's Law controls the maximum achievable speedup (S) of the parallel framework in this study as follows:

$$S(p) = \frac{1}{f + \frac{1-f}{p}} \tag{4}$$

where, f is the strictly sequential fraction of the application, and p is the number of cores.

In this implementation, f consisted of the following:

1. The sequential initialization and pre-processing of the qsBc table;
2. The latency of the operating system needed to spawn and context-switch the Pthreads.

As $f > 0$, the speedup scaling curve naturally flattened as p transitioned from four to eight cores, which clarified the sub-linear scaling curves observed empirically (see Section 4).

3.5.3 Synchronization and scalability costs

The text searching phase ended with the introduction of the synchronization overhead. The total match positions across chunks were aggregated into a global shared vector by applying a mutual exclusion lock (pthread_mutex_t) or an explicit barrier synchronization. A serialization overhead restricted by $O(p.k)$ was introduced by this important section, where the match hits frequency is denoted by k. The overhead of this synchronization barrier management began to compete with the decreasing computational chunk size ($\frac{n}{p}$) when p increased to 8, which indicated the scalability of the algorithm.

3.6 Experiment design

The experiment data for this study was sourced from the Pizza & Chili Corpus repository. Four distinct dataset categories were chosen (Extensible Markup Language (XML), Source, DNA, and Protein), each having a 200 MB file size. Statistical reliability was ensured, and anomalies were eliminated from background thread noise by performing each experiment five times and averaging the results. These iterations yielded negligible variance, which confirmed the high stability and environmental consistency of the profiles of execution.

Configurations with 2, 4, and 8 processing cores were

employed for parallel studies. The execution times were indicated as follows: Sequential implementation is denoted as seq, whilst executions utilizing 2, 4, and 8 processor cores were denoted by C2, C4, and C8, respectively. Improvement was assessed through a performance comparison between the sequential version and the parallel version. The averaged values were analyzed to enable a fair comparison of datasets and algorithm performance. Throughout the trials, the pattern length ranged from 10 to 100 characters. The performance data indicated the impact of different pattern sizes on execution time, speedup, and efficiency using varying colors.

4. RESULTS AND DISCUSSION

The dataset was partitioned into numerous sections and disseminated among processing cores using Pthreads to achieve SSABS algorithm parallelism. Variable pattern lengths were used on a 200-MB dataset to evaluate the performance of the algorithm through comparisons of speedup, efficiency, and parallel and sequential execution times. The utilization of varied database alphabet sizes enabled analysis of the behavior of the algorithm under several alphabet configurations. The comparison of the execution times under differing pattern lengths on a 200-MB dataset demonstrated that the parallel execution consistently outperformed the sequential execution. The source databases achieved ideal execution times for most of the pattern lengths. Conversely, the poorest performance for all pattern lengths was recorded by the DNA database (see Figure 2). The varied cross-dataset performance was based directly on the alphabet size (∑) and data entropy interaction, which determined the SSABS algorithm's shifting efficiency. Redundancies in structural formatting or alphabet symbol distributions (dense and low) caused the bad-character shift table (qsBc) to frequently produce minimal shift distances during a mismatch in the XML and DNA datasets; this forced threads into shorter execution loops. The source dataset allowed highly optimized execution scaling, and these execution times could be framed as processing throughput indicators. For example, only 28 ms was needed to complete the execution of the 200-MB source dataset across four cores with an m = 80 pattern length requirement; this can be paralleled to a peak sustained algorithmic processing throughput of about 7142.86 MB/s.

The speedup findings showed that the algorithm achieved optimal performance on the DNA database for most of the pattern lengths. When conducted on two and four cores, the other databases showed significant speedup; moreover, the DNA database specifically attained higher speedup when using four and eight cores. However, the protein database recorded the maximum speedup when using two cores. Figure 3 demonstrated that the source database achieved the weakest speedup when running on eight cores; the XML database recorded the lowest speedup when running on two and four cores.

The efficiency findings demonstrated that the algorithm operated optimally on the DNA database (Figure 4) compared to the XML, Source, and Protein datasets. The Protein database demonstrated optimal efficiency on two cores, while the DNA database performed optimally on four and eight cores. The Source database exhibited the lowest efficiency when running on eight cores, while the XML database showed the lowest efficiency on two and four cores.

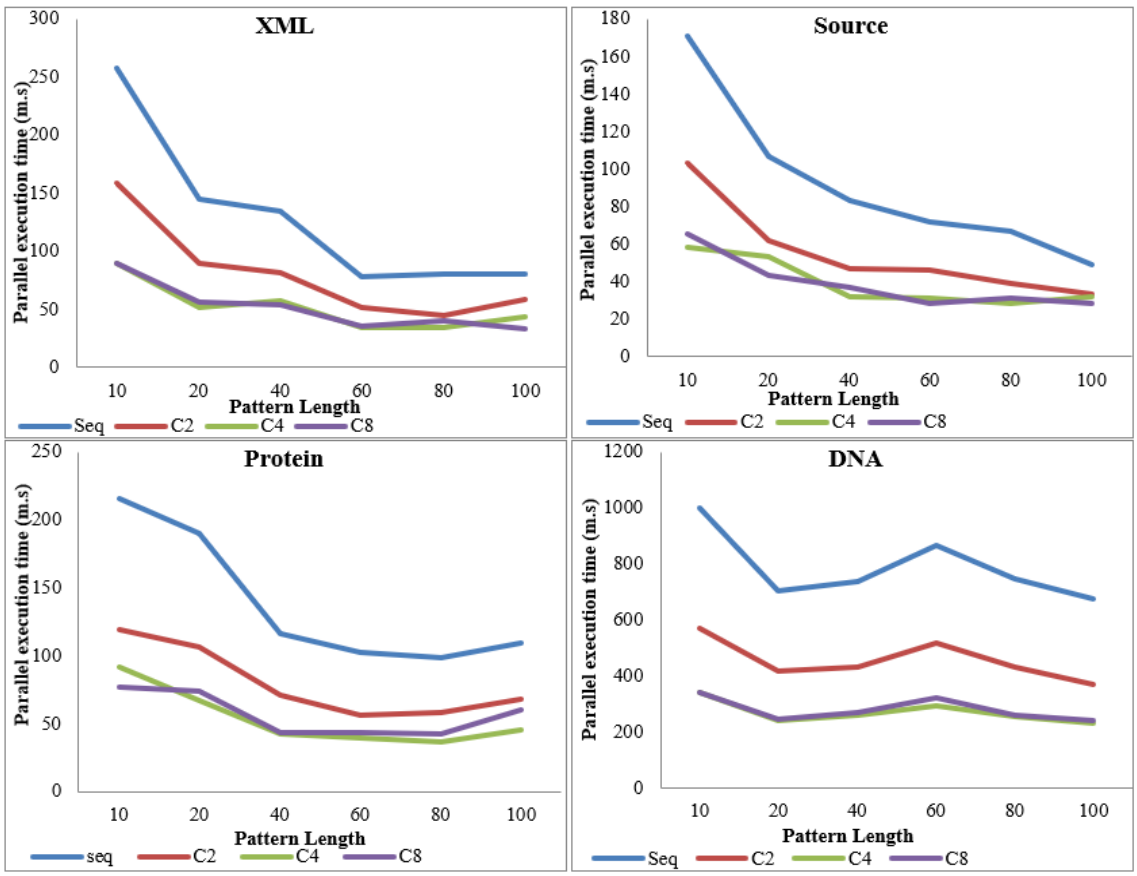

Figure 2. Parallel time using different data types: (a) Extensible Markup Language (XML), (b) Source code, (c) Protein, (d) DNA

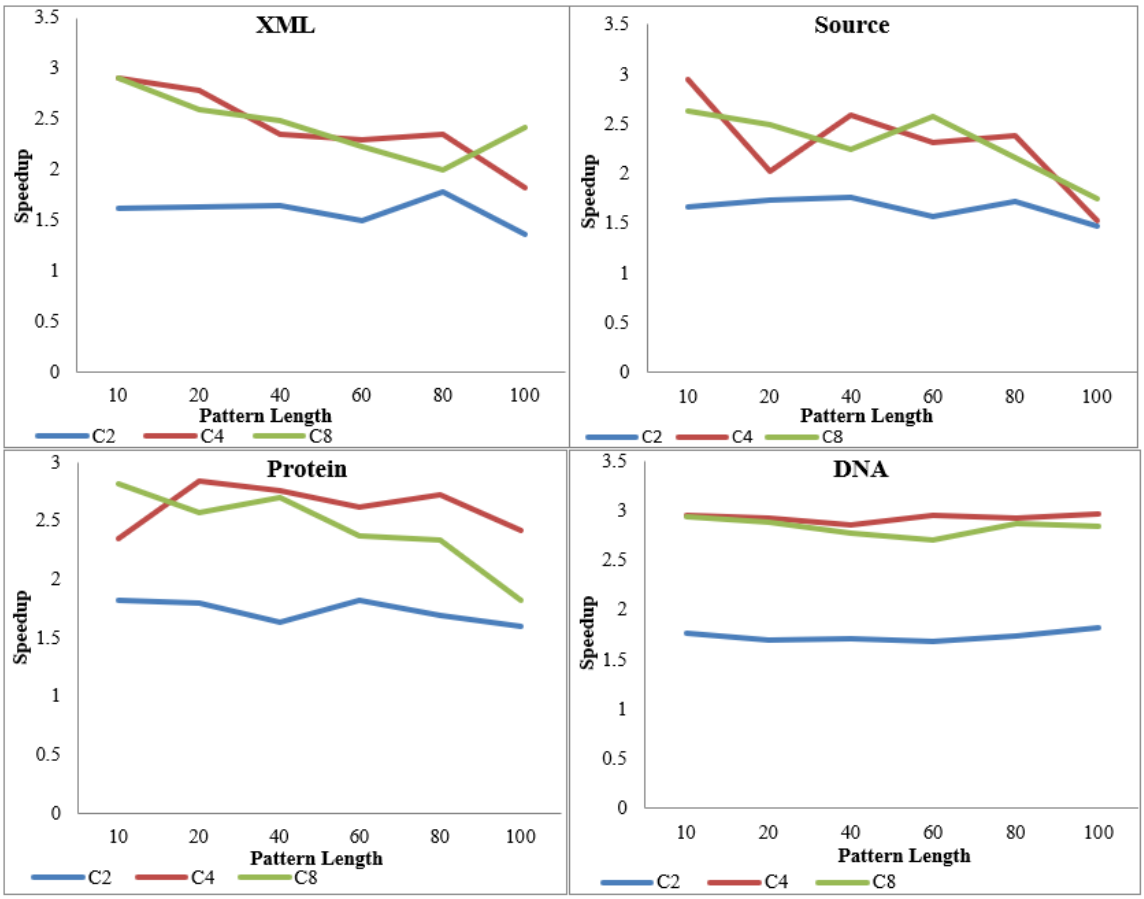

Figure 3. The speedup using different data types: (a) Extensible Markup Language (XML), (b) Source code, (c) Protein, (d) DNA

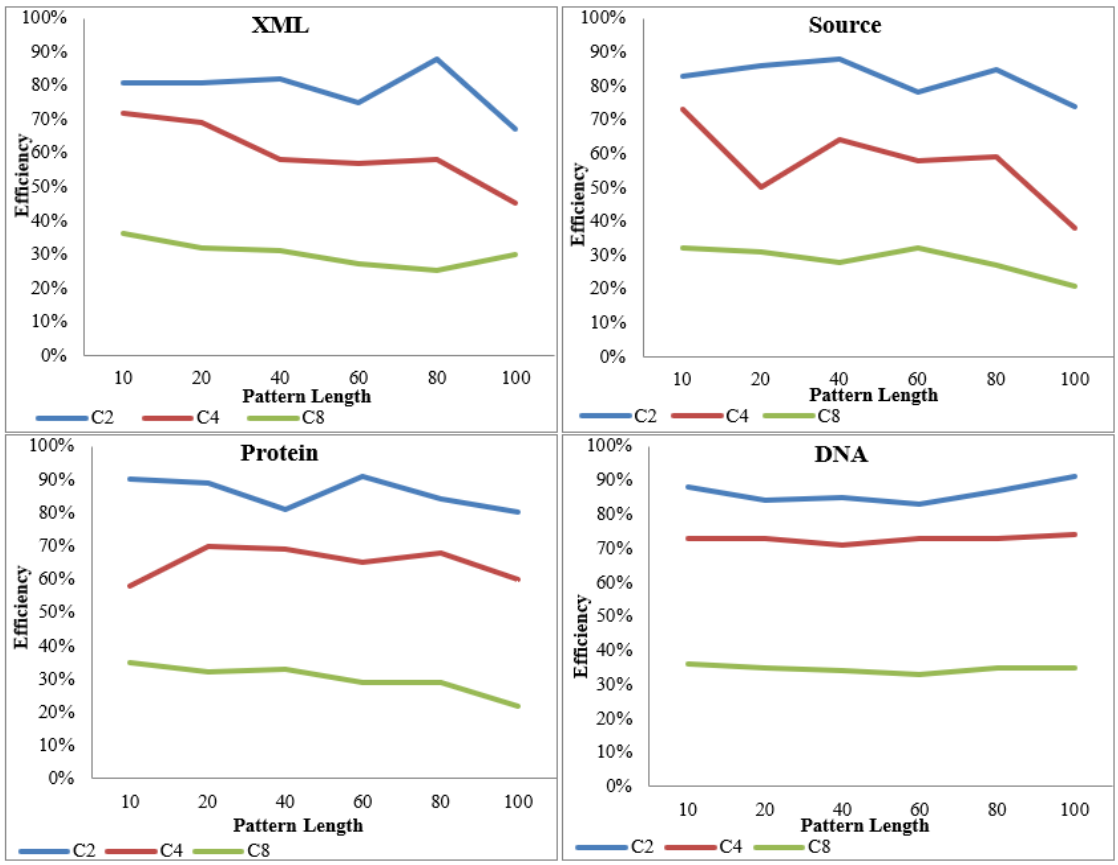

Figure 4. Efficiency using different data types: (a) Extensible Markup Language (XML), (b) Source code, (c) Protein, (d) DNA

The performance differences on the Source and DNA databases were examined to highlight the distinctness of the multi-core SSABS profile. The Source dataset demonstrated faster execution times (decreasing to 28 ms on eight cores for a 100-pattern length) given that its extensive alphabet allowed maximum qsBc shifts. Nevertheless, scaling up to eight processing cores decreased its parallel efficiency from 74% to 21%. The small-alphabet DNA dataset maintained a 33–36% efficiency profile under loads of eight cores. This result demonstrated that the parallelized SSABS right-to-left scanning steps could remain localized in individual CPU core L1 or L2 caches; this could possibly be due to the high rate of character discrepancies in small-alphabet strings, which caused minimal global thread synchronization delays compared to datasets with large alphabets.

These results showed that parallel execution improved performance compared to sequential execution at the initial stage. Nevertheless, overhead increased with increasing number of cores, with the extra communication expense having an adverse impact on the parallel execution time. A significant decrease in parallel and sequential execution times was noted across the various pattern lengths with increasing number of cores and pattern length [9, 17]. Since the SSABS algorithm employed the effective qsBc function, the Source dataset generated the optimal sequential and parallel execution times for the different pattern lengths. The DNA dataset had the smallest alphabet size compared to the other datasets, resulting in prolonged processing times.

In comparison to the DNA dataset, the slower speedup of the XML and Source datasets throughout the core counts, and the slower speedup of the Protein dataset on four and eight cores, could be due to their higher alphabet size. Furthermore, increasing the number of cores significantly reduced the parallel execution time, which impacts the total speedup. These elements result in decreased speedup values for datasets with huge alphabets when used in combination. The DNA dataset is not considerably impacted due to its small alphabet count; hence, the searching phase shifts more frequently, which extends the execution time but enhances speedup performance.

This multi-core setup highlighted three important design choices that optimized the parallel SSABS algorithm:

1. Low communication overhead (Pthreads vs. MPI): Heavy communication penalties were avoided by choosing Pthreads over a distributed framework (MPI). Computer nodes are required to manage pattern boundaries ($m - 1$) by continuously sending data to and from over network cables in MPI; however, the Pthreads model involved all threads sharing the same physical memory. Hence, communication was restricted to rapid internal hardware cache updates (L3 cache) and minimal synchronization locks. Even when locking slightly reduced performance at eight cores, the delay was minimal in comparison to sluggish network transfers.

2. Smart thread allocation (preventing oversubscription): System efficiency was maintained by matching the number of software threads precisely to the physical CPU core numbers (P = Threads = Cores) rather than creating more threads than physical cores to force the operating system to continuously switch threads in and out of the CPU (context switching). For the data-heavy SSABS algorithm, the CPU cache performance was impacted, scheduling was delayed, and the final result-gathering phase was hindered by this continuous switching. Maximum speed was guaranteed by a strict 1:1 mapping.

3. Balanced workloads (static partitioning): The dataset was divided evenly among the available cores to ensure that each core processes an equal block size n/p. The processing times theoretically varied as the SSABS algorithm shifted by different distances due to different text blocks. Nonetheless, the relatively large test datasets ensured the uniform spread of characters across the files. In the DNA and Protein datasets, highly symmetric thread execution times were maintained by this balanced distribution of characters; this prevented single cores from idling while waiting for other cores to complete execution.

This multi-core parallel framework is characterized by the following technical features:

1. Dataset size (200 MB): A 200 MB dataset was selected as it exceeded characteristic sizes of CPU caches (<32 MB); this forced the system to directly stream data from the main RAM and isolated true parallel performance. Runtime would increase linearly with a larger file (1 GB), with the efficiency curves remaining unchanged; the cache would accommodate a smaller file and real memory delays would be hidden.

2. Memory footprint: Pthreads was used to load the 200 MB dataset into RAM as a shared and read-only segment. Assigned sections were accessed by worker threads through low-overhead memory pointers instead of replicating data; this was aimed at maintaining a minimal additional memory footprint that consisted only of thread-local variables and a single mutex lock and strictly bound total memory to $O(n + m + p)$ bytes.

3. The eight-core performance plateau: The speedup curve flattened as the core counts scaled to eight owing to two main issues: i) individual text chunks became so small that the costs for thread management of the operating system competed with the actual duration of scanning; ii) multiple threads that found matches simultaneously were required to wait to access the shared mutex lock (pthread_mutex_t); this created a synchronization issue that limited additional acceleration.

Efficiency increased with speedup as the two variables are positively correlated; consequently, efficiency was reduced by using eight cores. Compared to the XML, Source, and Protein datasets, the DNA database demonstrated the highest global efficiency.

5. CONCLUSION

The parallelization of the SSABS exact string-matching algorithm was focused on in this study; the parallel version of the algorithm was implemented using the Pthreads package. Within the specific assessment scope (using 200-MB benchmarking datasets across conformations of up to eight processing cores), the findings demonstrated that the parallelized program performed better than the sequential version across different data types and pattern lengths. The performance of the algorithm was optimal on the Source dataset in terms of parallel execution time. However, the DNA dataset recorded the longest execution time owing to its limited shifting distances. The DNA dataset achieved the highest overall speedup and efficiency in terms of performance scaling. Larger-alphabet datasets, such as Source and XML, on the other hand, achieved lower speedup metrics despite their rapid absolute runtimes. This setup demonstrated the primary architectural issue of parallel synchronization overhead: the costs of hardware messaging and context switching counteracted gains in computational efficiency as the thread counts scaled to eight cores.

An important limitation of this study is its dependence on datasets of fixed size in a unified shared-memory architecture. These issues may be mitigated by parallelizing the SSABS algorithm preprocessing and searching stages to remove sequential checkpoints during optimization. Hence, future work will concentrate on extending the algorithm to other message-passing multiprocessor models (such as MPI) and heterogeneous frameworks to assess its scalability for gigabyte- and terabyte-scale databases.

REFERENCES

[1] Tarigan, D.A., Buaton, A.O., Briyandana, B., Safitri, E.R., Rosnelly, R. (2024). Analysis of string matching application on serial number using Boyer Moore algorithm. CNAHPC, 6(1): 237-246. https://doi.org/10.47709/cnahpc.v6i1.3410

[2] Jargalsaikhan, D., Hendrian, D., Yoshinaka, R., Shinohara, A. (2022). Parallel algorithm for pattern matching problems under substring consistent equivalence relations. In 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022), Prague, Czech Republic, pp. 28:1-28:21. https://doi.org/10.4230/LIPIcs.CPM.2022.28

[3] Faro, S., Lecroq, T. (2013). The exact online string matching problem: A review of the most recent results. ACM Computing Surveys, 45(2): 1-42. https://doi.org/10.1145/2431211.2431212

[4] AbdulRazzaq, A.A., Fadhel, M.A., Alzubaidi, L., Al-Shamma, O. (2023). Parallel processing of E-Atheer algorithm using Pthread paradigm. Indonesian Journal of Electrical Engineering and Computer Science, 30(3): 1624. https://doi.org/10.11591/ijeecs.v30.i3.pp1624-1633

[5] Al-Dabbagh, S.S.M., Abdal, Y.M. (2021). Parallel hybrid string matching algorithm using CUDA API function. In 2021 International Conference on Computing and Communications Applications and Technologies (I3CAT), Ipswich, UK, pp. 66-70. https://doi.org/10.1109/i3cat53310.2021.9629415

[6] Dababat, W. (2020). Intelligent predictive string search algorithm using two sliding windows in parallel environment. International Journal of Advanced Trends in Computer Science and Engineering, 9(4): 6346-6355. https://doi.org/10.30534/ijatcse/2020/316942020

[7] Zhang, Z. (2022). Review on string-matching algorithm. SHS Web of Conferences, 144: 03018. https://doi.org/10.1051/shsconf/202214403018

[8] Hakak, S.I., Kamsin, A., Shivakumara, P., Gilkar, G.A., Khan, W.Z., Imran, M. (2019). Exact string matching algorithms: Survey, issues, and future research directions. IEEE Access, 7: 69614-69637. https://doi.org/10.1109/access.2019.2914071

[9] Ibrahim, O.A.S., Hamed, B.A., El-Hafeez, T.A. (2022). A new fast technique for pattern matching in biological sequences. The Journal of Supercomputing, 79(1): 367-388. https://doi.org/10.1007/s11227-022-04673-3

[10] Mahmud, P., Rana, M.S., Talukder, K.H. (2018). An efficient hybrid exact string matching algorithm to minimize the number of attempts and character comparisons. In 2018 21st International Conference of Computer and Information Technology (ICCIT), Dhaka, Bangladesh, pp. 1-6. https://doi.org/10.1109/ICCITECHN.2018.8631908

[11] Ozsoy, A., Nazli, M., Cankur, O., Sahin, C. (2025). CUSMART: Effective parallelization of string matching algorithms using GPGPU accelerators. Frontiers of Information Technology & Electronic Engineering, 26(6): 877-895. https://doi.org/10.1631/fitee.2400091

[12] Baloi, A., Belean, B., Turcu, F., Peptenatu, D. (2024). GPU-based similarity metrics computation and machine learning approaches for string similarity evaluation in large datasets. Soft Computing, 28(4): 3465-3477. https://doi.org/10.21203/rs.3.rs-1776657/v1

[13] Karcioglu, A.A., Bulut, H. (2021). Improving hash-q exact string matching algorithm with perfect hashing for DNA sequences. Computers in Biology and Medicine, 131: 104292. https://doi.org/10.1016/j.compbiomed.2021.104292

[14] Rexie, J.A.M., Raimond, K., Murugaaboopathy, M., Brindha, D., Mulugeta, H. (2022). Lightweight pattern matching method for DNA sequencing in Internet of Medical Things. Computational Intelligence and Neuroscience, 2022(1): 6980335. https://doi.org/10.1155/2022/6980335

[15] Nunes, L.S., Bordim, J.L., Ito, Y., Nakano, K. (2018). Parallel Rabin-Karp algorithm implementation on GPU (preliminary version). Bulletin of Networking, Computing, Systems, and Software, 7(1): 28-32.

[16] Akram AbdulRazzaq, A., Abdul Rashid, N. (2022). Parallel processing outcomes of E-Abdulrazzaq algorithm using multi-core technique. Iraqi Journal for Computers and Informatics, 48(2): 1-8. https://doi.org/10.25195/ijci.v48i2.463

[17] Pandey, R.K., Taruna, S. (2021). Prevalent exact string-matching algorithms in natural language processing: A review. Journal of Physics: Conference Series, 1854(1): 012042. https://doi.org/10.1088/1742-6596/1854/1/012042

[18] Aldwairi, M., Conte, T., Franzon, P. (2005). Configurable string matching hardware for speeding up intrusion detection. ACM SIGARCH Computer Architecture News, 33(1): 99-107. https://doi.org/10.1145/1055626.1055640

[19] Kamil AL-Jazayiri, H., AbdulRazzaq, A.A. (2024). Enhanced hybrid algorithm for E-Abdulrazzaq and fast online hybrid matching algorithms for exact string matching. Iraqi Journal for Computers and Informatics, 50(1): 20-33. https://doi.org/10.25195/ijci.v50i1.452

[20] Kouzinopoulos, C.S. (2013). Parallel and distributed implementations of multiple and two-dimensional pattern matching algorithms. Ph.D. dissertation, Department of Applied Informatics, University of Macedonia, Greece.

[21] Raju, K.B., Rao, C.S., Raju, S.V. (2013). A frame work for parallel string matching-A computational approach with Omega model. Global Journal of Computer Science and Technology, 13(2-A): 13-20.

[22] Hnaif, A.A., Alhalaiqah, M., Abouabdalla, O., Ramadass, S., Kadhum, M.M. (2009). Parallel quick search algorithm to speed packet payload filtering in NIDS. Journal of Engineering Science and Technology, 4(2): 220-230.

[23] Rasool, A., Khare, N. (2012). Parallelization of KMP string matching algorithm on different SIMD architectures: Multi-core and GPGPU's. International Journal of Computer Applications, 49(11): 26-28.

https://doi.org/10.5120/7672-0963
[24] Abu-Zaid, I.M., El-Rayyes, E.K. (2012). Parallel search using KMP algorithm in Arabic string. International Journal of Science and Technology, 2(7): 427-431.
[25] Shah, P., Oza, R. (2017). Improved parallel Rabin-Karp algorithm using compute unified device architecture. In Information and Communication Technology for Intelli, New York, USA, pp. 236-244. https://doi.org/10.1007/978-3-319-63645-0_26
[26] Faro, S., Külekci, M.O. (2012). Fast multiple string matching using streaming SIMD extensions technology. In Proceedings of the 19th International Symposium on String Processing and Information Retrieval, Cartagena de Indias, Colombia, pp. 217-228. https://doi.org/10.1007/978-3-642-34109-0_23
[27] Kouzinopoulos, C.S., Margaritis, K.G. (2009). String matching on a multicore GPU using CUDA. In 2009 13th Panhellenic Conference on Informatics, Corfu, Greece, pp. 14-18. https://doi.org/10.1109/PCI.2009.47
[28] Cali, D.S., Kalsi, G.S., Bingöl, Z. et al. (2020). GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, pp. 951-966. https://doi.org/10.1109/MICRO50266.2020.00081
[29] Zavadskyi, I.O. (2022). Fast exact pattern matching by the means of a character bit representation. SN Computer Science, 3(3): 181. https://doi.org/10.1007/s42979-022-01052-w
[30] Sheik, S.S., Aggarwal, S.K., Poddar, A., Balakrishnan, N., Sekar, K. (2004). A fast pattern matching algorithm. Journal of Chemical Information and Computer Sciences, 44(4): 1251-1256. https://doi.org/10.1021/ci030463z
[31] Kumar, V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to Parallel Computing. Redwood City, CA: Benjamin/Cummings.