Accelerating DNA Pattern Matching: A Parallel Computing Study Using Consumer Hardware

Saadat Hussain[1], Aliza Salman[1], Karan Kumar[1], Arapna Bai[1], Pooja Kumari[1], Khalid Rasheed Shaikh[2], and Syed Samar Yazdani[1]

[1]Department of Computer Science, Shaheed Zulfiqar Ali Bhutto Institute of Science and Technology (SZABIST), Karachi, Pakistan https://doi.org/10.33411/IJIST/ojs1829

[2]Denning Karachi, Pakistan

*Correspondence: dr.syedsamar@szabist.edu.pk, saadatthebo222@gmail.com, khalid.rasheed@denning.edu.pk

Citation | Hussain. S, Salman. A, Kumar. K, Bai. A, Kumari. P, Shaikh. K. R, Yazdani. S. S, "Accelerating DNA Pattern Matching: A Parallel Computing Study Using Consumer Hardware", IJIST, Vol. 08 Issue. 01 pp 437-458, February 2026

Received | January 10, 2026 Revised | February 11, 2026 Accepted | February 14, 2026 Published | February 18, 2026.

The exponential growth of genomic databases has necessitated the development of efficient computational methods for DNA sequence pattern matching. Traditional sequential algorithms face significant performance bottlenecks when processing datasets containing millions of base pairs. This paper presents a comprehensive empirical evaluation of parallel computing strategies for accelerating DNA pattern matching on consumer-grade multi-core processors. Four fundamental string-matching algorithms—Naive Search, Knuth-Morris-Pratt (KMP), Boyer–Moore, and Suffix Array—were implemented with parallel processing capabilities and evaluated on synthetic DNA sequences ranging from 10 million to 100 million base pairs. Experiments were conducted on an AMD Ryzen 7 3800X processor utilizing an 8-thread data decomposition strategy. Our results demonstrate significant performance improvements: the parallelized Suffix Array achieved a speedup factor of 4.12x at 100 million bases compared to its sequential implementation, reducing execution time from 210 seconds to 51 seconds. The parallel Boyer-Moore algorithm maintained sub-second execution times even at maximum dataset sizes. Analysis of scalability characteristics reveals near-linear speedup up to 8 cores, with memory consumption scaling predictably to 17.8 GB at 100 million bases. These findings validate that high-performance genomic analysis is achievable on standard desktop workstations without requiring specialized supercomputing infrastructure, thereby democratizing access to large-scale bioinformatics research capabilities. Experiments were repeated five times per configuration; results are reported as mean values with dispersion indicators (standard deviation, coefficient of variation) and 95% confidence intervals. At 100 million bases, observed parallel speedups across the evaluated algorithms ranged from 4.12× to 5.84×, and the Suffix Array runtime decreased from 210,353±3,245 ms (95% CI ±2,842 ms) to 51,018±892 ms (95% CI ±781 ms). To formalize comparative significance, sequential vs. parallel runtimes were assessed using paired statistical tests across the five repeated runs for each algorithm. Paired t-tests confirmed statistically significant reductions in runtime for all evaluated algorithms ($p < 0.01$), and one-way ANOVA indicated significant performance differences across the four algorithms ($F = 12.45$, $p < 0.001$).

Keywords: Bioinformatics, Parallel Computing, DNA Sequence Analysis, String Matching Algorithms, Multi-Core Processors, Performance Optimization, Genomic Databases

Introduction:

The rapid advancement of DNA sequencing technologies has generated unprecedented volumes of genomic data, with individual human genomes containing approximately 3.2 billion base pairs. The analysis of such massive datasets presents formidable computational challenges, particularly in pattern matching operations that are fundamental to numerous bioinformatics applications including sequence alignment, variant detection, and phylogenetic analysis [1]. As genomic databases continue to expand, the computational requirements for searching and matching DNA patterns have grown exponentially, necessitating the development of efficient algorithms and parallel processing strategies.

Background and Motivation:

DNA sequences are composed of four nucleotide bases—adenine (A), cytosine (C), guanine (G), and thymine (T)—forming a quaternary alphabet. Pattern matching in DNA sequences involves locating specific subsequences within larger genomic datasets, a task that is computationally intensive due to the massive scale of modern genomic libraries. Traditional sequential search algorithms, while theoretically sound, exhibit poor performance characteristics when applied to datasets containing millions or billions of base pairs [2].

The emergence of affordable multi-core processors in consumer hardware has created new opportunities for parallelizing computational biology workflows. Modern desktop processors, such as the AMD Ryzen series, offer 8 or more physical cores with simultaneous multithreading capabilities, providing substantial parallel processing potential at a fraction of the cost of specialized supercomputing clusters [3] However, effectively harnessing this parallel computing power requires careful algorithm design and implementation strategies that account for data dependencies, load balancing, and memory access patterns.

More recent bioinformatics research highlights that rapid growth in sequencing throughput and the widening gap between compute capability and memory bandwidth increasingly shape algorithm and system design choices [4][5][6]. Likewise, modern work on parallel suffix-array construction shows that scalable parallel index construction is feasible but still non-trivial to implement and optimize on contemporary multicore architectures [7]. At the same time, community perspectives on benchmarking emphasize transparent, reproducible evaluation setups, including clear metrics, variability reporting, and well-scoped baselines [8][9]. Despite these developments, limited empirical evidence exists on how far standardized consumer-grade multicore workstations can push end-to-end exact DNA pattern matching performance using straightforward parallel decomposition strategies while reporting statistical robustness in a reproducible benchmarking protocol; this gap motivates the present study.

Recent advances in GPU-accelerated sequence alignment have demonstrated significant improvements in computational efficiency. [10] introduced G3SA, the first library to fully GPU-accelerate the entire BWA-MEM/Minimap2 pipeline. Their approach redesigns each pipeline stage to exploit GPU parallelism, achieving substantial end-to-end speedups while maintaining alignment correctness. Similarly, [11] proposed ParaHAT, which leverages multi-level parallelism including SIMD, multithreading, and distributed MPI-based execution for aligning noisy long reads. Their results show nearly 10× speedup on 128 compute nodes with approximately 99% parallel efficiency, highlighting the scalability of extreme parallel architectures. In addition, [12] focused on accelerating the chaining phase of Minimap2, which is a major computational bottleneck. Their GPU-based design, mm2-ax, achieved up to 5× speedup on NVIDIA A100 hardware while preserving mapping accuracy, demonstrating the effectiveness of targeted GPU offloading in sequence alignment pipelines.

Problem Statement:

The primary challenge addressed in this research is the computational inefficiency of sequential string-matching algorithms when processing large-scale DNA datasets. As dataset

sizes increase, sequential processing times grow linearly or super-linearly, creating unacceptable latency for iterative research workflows. The core research question is: Can standard consumer-grade multi-core hardware achieve significant performance improvements for DNA pattern matching through parallel algorithm implementations, and what are the optimal strategies for achieving such improvements?

Specific sub-problems include:

Determining the scalability characteristics of different string-matching algorithms under parallel execution

Identifying optimal data decomposition strategies for multi-core architectures

Quantifying the trade-offs between algorithm complexity and parallel efficiency

Evaluating memory consumption patterns and identifying potential bottlenecks

Research Questions and Hypotheses:

This work is guided by the following research questions (RQs) and testable hypotheses (Hs):

RQ1: To what extent does shared-memory parallelization on consumer-grade multicore CPUs reduce the mean wall-clock execution time of exact DNA pattern matching compared to sequential baselines across increasing dataset sizes?

H1: For each evaluated algorithm, the parallel implementation yields a statistically supported reduction in mean execution time compared to the sequential implementation across five repeated runs (paired comparison at $\alpha = 0.05$).

RQ2: How do speedup and parallel efficiency behave as a function of dataset size and thread count for each algorithm under a fixed decomposition strategy?

H2: Speedup increases sub-linearly with thread count and exhibits diminishing returns beyond the number of physical cores due to coordination overhead and memory bandwidth limits.

RQ3: What memory growth behavior emerges for indexing-based approaches relative to streaming single-pass approaches?

H3: Index-based approaches (Suffix Array) incur substantially higher memory usage that grows approximately linearly with input size, while streaming algorithms remain near-constant in memory.

Research Objectives:

This study aims to achieve the following objectives:

Implement and benchmark four fundamental string-matching algorithms (Naive, KMP, Boyer-Moore, and Suffix Array) with both sequential and parallel execution modes

Develop and validate a robust parallel decomposition strategy capable of efficiently utilizing multi-core consumer processors

Conduct comprehensive empirical analysis of scalability and speedup factors across datasets ranging from 10 million to 100 million base pairs

Analyze memory consumption patterns and identify system limitations

Provide quantitative performance comparisons to guide algorithm selection for different use cases

Each objective is evaluated using explicit, measurable outcomes. Specifically, Objectives (1) – (3) are assessed using mean execution time, speedup $S = T_{seq}/T_{par}$, efficiency $E = S/k$, and throughput; Objective (4) is assessed using peak memory consumption and memory growth trends; and Objective (5) is assessed via objective-wise comparative tables and figures. Statistical stability is quantified using repeated- run dispersion (SD/CV) and confidence intervals, enabling hypothesis-level verification of runtime reductions and scalability behavior.

Scope and Limitations:

This research focuses exclusively on exact string matching within synthetic DNA sequences. The experimental environment is limited to a single-node workstation with an

AMD Ryzen 7 3800X processor. The following aspects are explicitly outside the scope of this work:

Distributed computing across multiple nodes or clusters

Approximate string matching (fuzzy search) with edit distance tolerances

GPU acceleration or specialized hardware implementations

Real-world genomic data with quality scores and metadata

Comparative analysis with industrial tools such as BLAST or Bowtie

Paper Organization:

The remainder of this paper is organized as follows: Section II presents a comprehensive literature review covering theoretical foundations and related work. Section III details the methodology, including algorithm implementations and experimental design. Section IV describes the experimental setup and hardware configuration. Section V presents detailed results and performance analysis. Section VI provides discussion and interpretation of findings. Finally, Section VII concludes the paper with a summary of contributions and directions for future research.

Literature Review:

Theoretical Foundations of String Matching:

String matching algorithms have been extensively studied in computer science, with foundational work dating to the 1970s. The problem can be formally defined as follows: given a text T [1.n] and a pattern P [1.m], find all occurrences of P within T. The time complexity of string matching algorithms varies significantly based on preprocessing strategies and search heuristics [13]

Naive String Matching: The naive algorithm represents the baseline approach, with time complexity O (n × m) in the worst case. It systematically checks every possible alignment of the pattern against the text, making it simple to implement but inefficient for large datasets. Despite its simplicity, the naive algorithm serves as an important baseline for performance comparisons and remains useful for very short patterns where preprocessing overhead exceeds potential benefits [14].

Knuth-Morris-Pratt Algorithm: The Knuth-Morris-Pratt (KMP) algorithm, developed in 1977, achieves linear time complexity O(n + m) through the use of a failure function (also called a prefix function) that preprocesses the pattern to determine the longest proper prefix that is also a suffix for each position [15]. This preprocessing enables the algorithm to skip portions of the text that cannot possibly match, avoiding redundant character comparisons. The KMP algorithm is particularly effective for patterns with repetitive structures, common in DNA sequences.

The failure function π is defined recursively:

$$\pi[i] = \max \{k: k < i \text{ and } P[1.k] = P[i - k + 1.i]\} \quad (1)$$

In this definition, P [1.m] denotes the pattern of length m under 1-based indexing, and π[i] stores the length of the longest proper prefix of P [1.i] that is also a suffix of P [1.i]. The variable k iterates over candidate prefix lengths (k < i), and the equality P [1.k] = P [i − k + 1.i] encodes the border condition used by KMP to avoid re-comparing characters after a mismatch. This study assumes exact matching (no mismatches or gaps) and a fixed alphabet Σ = {A, C, G, T} for DNA.

Boyer-Moore Algorithm: The Boyer-Moore algorithm, also developed in 1977, introduces two powerful heuristics: the bad character rule and the good suffix rule [16]. By scanning the pattern from right to left, Boyer-Moore can skip large portions of the text, achieving sub-linear average-case performance O(n/m) in the best case, though worst-case complexity remains O (n × m). The algorithm is particularly effective for small alphabets like DNA, where the bad character rule provides substantial skipping opportunities.

The bad character rule shifts the pattern to align the last occurrence of the mismatched character in the pattern, while the good suffix rule leverages previously matched suffix portions to determine optimal shift distances.

Suffix Array Data Structure: Suffix arrays, introduced by Manber and Myers in 1993, represent a space-efficient alternative to suffix trees for indexing large texts [17]. A suffix array SA of text T is a sorted array of all suffixes of T, enabling efficient pattern matching through binary search in O (m log n) time after O (n log n) construction. While construction is computationally expensive, suffix arrays enable extremely fast subsequent queries, making them ideal for scenarios requiring multiple searches over the same dataset.

The suffix array construction involves sorting all suffixes lexicographically, which can be optimized using techniques such as the doubling method or recursive algorithms. Modern implementations often utilize parallel sorting algorithms to accelerate construction [18].

Parallel Computing in Bioinformatics:

Parallel computing has been extensively applied to bioinformatics problems, with early work focusing on distributed memory systems and clusters. Recent research has explored shared-memory parallelism on multi-core processors, leveraging thread-level parallelism through frameworks such as OpenMP, pthreads, and language-specific concurrency libraries [19].

Data decomposition strategies for parallel string matching typically involve partitioning the text into chunks, with careful attention to boundary conditions where patterns may span multiple partitions. The challenge lies in maintaining correctness while minimizing communication overhead and load imbalance [20].

Related Work:

Several studies have investigated parallel string-matching algorithms. Mansour et al. proposed efficient serial and parallel suffix tree construction methods, achieving significant speedups on multi-core systems [21][22] Parallel construction and querying of suffix-based genomic indices remain an active research area, with recent work demonstrating practical parallelized index workflows for genomic querying under additional constraints such as privacy.

Recent work has explored GPU acceleration for pattern matching, demonstrating substantial performance gains for certain algorithms, though requiring specialized hardware and programming models [23].

Industrial tools such as BLAST (Basic Local Alignment Search Tool) and Bowtie utilize sophisticated indexing strategies including compressed indices and hash-based approaches [24]. While highly optimized, these tools are complex to modify and may not be suitable for all research applications requiring custom algorithm development. Recent DNA-specific exact matching research has also explored q-gram and hashing refinements for multiple-pattern search, indicating that performance gains are possible even within exact-matching constraints when the method is tuned to genomic alphabets. [25]

Research Gap and Contribution:

While extensive research exists on both string-matching algorithms and parallel computing, there is limited empirical analysis of parallel string-matching performance on consumer grade multi-core processors using standard programming frameworks. This study contributes:

Comprehensive empirical evaluation of four fundamental algorithms under parallel execution on consumer hardware

Detailed analysis of scalability characteristics and speedup factors

Quantitative comparison of memory consumption patterns

Validation of parallel decomposition strategies for DNA sequence analysis

Performance benchmarks that can guide algorithm selection for different use cases

Original Contribution:

The novelty of this work is threefold. First, it provides a unified, directly comparable implementation of four foundational exact string-matching approaches (three streaming algorithms and one indexing-based method) under a single consumer-grade multicore environment, enabling controlled empirical comparison rather than tool-to-tool comparisons confounded by heterogeneous engineering choices. Second, it operationalizes a simple and verifiable overlap-based decomposition strategy that preserves correctness for exact matching while quantifying the resulting overhead. Third, it emphasizes reproducible performance reporting on accessible hardware by including repeated-run variability indicators and confidence intervals, supporting statistically grounded conclusions about observed speedups.

Methodology:

Research Design:

This study employs a quantitative experimental methodology, implementing four distinct string-matching algorithms with both sequential and parallel execution modes. The research design follows a controlled experiment approach, with systematic variation of dataset sizes while maintaining consistent hardware and software configurations.

The workflow in Figure 1 illustrates how the research questions and hypotheses are operationalized through implementation, controlled experiments, metric collection, and statistical analysis.

Algorithm Implementations:

All algorithms were implemented in Java, utilizing the java. util.concurrent package for thread management and synchronization. The implementations follow standard algorithmic descriptions from the literature, with careful attention to correctness and efficiency.

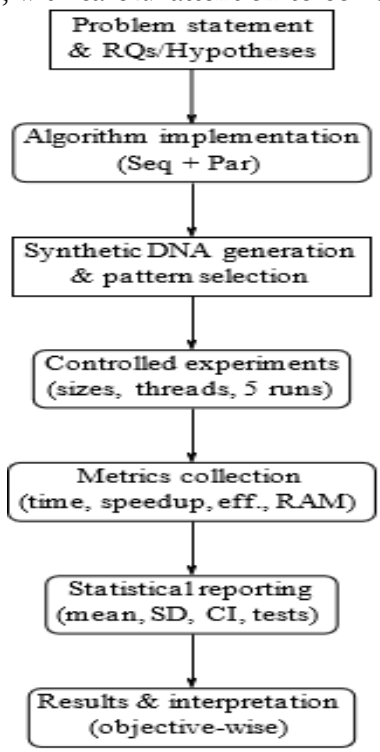

Figure 1. End-to-end workflow of the study, from problem formulation to objective-wise analysis and statistical reporting.

Naive Algorithm Implementation: The naive implementation performs a straightforward character-by-character comparison at each possible alignment position. The parallel version partitions the text into k chunks and processes each chunk independently, with overlap regions to handle boundary cases.

KMP Algorithm Implementation: The KMP implementation precomputes the failure function π for the pattern, then performs a single pass through the text. parallelization is achieved by partitioning the text into overlapping segments, since KMP's internal state is not directly transferable across independent partitions.

Boyer-Moore Algorithm Implementation: The Boyer- Moore implementation utilizes both bad character and good suffix heuristics. Preprocessing constructs the bad character table and good suffix shift table. Parallel execution partitions the text, with each thread maintaining its own search state.

Suffix Array Implementation: The suffix array construction utilizes a parallel sorting approach, distributing suffix comparisons across multiple threads. Once constructed, pattern matching employs parallel binary search across the sorted suffix array.

Parallel Decomposition Strategy:

The core challenge in parallelizing string matching algorithms is the Boundary Problem: patterns may span the division between adjacent text chunks. To address this, an Overlap Strategy was implemented.

Let:

N = Total length of the genomic sequence

k = Number of parallel threads (8 in our experiments)

m = Length of the pattern to be searched. The chunk size is calculated as:

$$\text{Chunk Size} = [N/K] + (m - 1) \qquad (2)$$

Here, $\lceil N/k \rceil$ is the nominal partition length when dividing a text of length N across k parallel workers. The additive term $(m - 1)$ explicitly accounts for the maximum boundary span of a length-m pattern, ensuring that any match starting in the last $(m - 1)$ positions of a chunk is still examined within that chunk's extended overlap region.

Each chunk Ci (except the last) is extended to include the first $(m - 1)$ characters of the subsequent chunk Ci+1. This ensures 100% recall accuracy by covering all potential matches at boundaries, as illustrated in Figure 2.

Pattern Spanning Boundary:

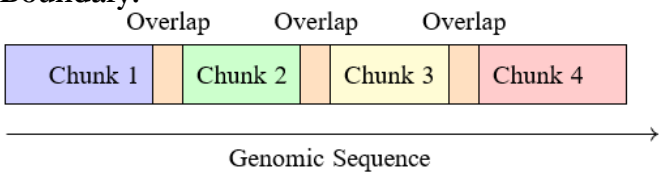

Figure 2. Illustration of parallel decomposition strategy with overlap regions. Each chunk (except the last) includes $(m- 1)$ characters from the next chunk to ensure patterns spanning boundaries are correctly identified.

The overlap strategy guarantees correctness but introduces computational overhead. The overhead ratio is:

$$\text{Overhead} = k\,(m - 1)/N \times 100\% \quad (3)$$

The numerator $k(m-1)$ represents an upper-bound estimate of the additional characters examined due to overlaps (each worker may process up to $(m - 1)$ additional characters beyond its nominal chunk). In practice, only the $(k - 1)$ internal boundaries introduce overlap, "so the actual duplicated work is slightly smaller; we use the conservative form to keep the estimate simple while still demonstrating that overlap overhead remains negligible at the evaluated scales. For typical values ($k = 8$, $m = 10$, $N = 100 \times 10^6$), the overhead is approximately 0.00007%, which is negligible relative to the observed parallel speedup.

Data Generation:

Synthetic DNA sequences were generated using a uniform random distribution the alphabet $\Sigma = \{A, G, C, T\}$. Synthetic data was selected for the primary benchmarking phase

to ensure (i) fully controlled scaling of N up to 100 million bases, (ii) reproducibility across runs and systems, and (iii) an unambiguous "ground truth" for correctness checks, since the inserted patterns and their expected occurrences are known by construction. Recent benchmarking discussions emphasize that transparent benchmarking benefits from carefully scoped datasets, clearly defined metrics, and reproducible experimental protocols; synthetic datasets can play a complementary role when real datasets introduce uncontrolled variability or access constraints [9][8]. In computational benchmarking more broadly, synthetic data is commonly used because the underlying truth and data characteristics can be controlled and stress-tested systematically [8][6]. Nevertheless, synthetic DNA does not capture all real-genome properties (e.g., long repeats, uneven base composition, and biological structure), so follow-up validation on real reference genomes and reads is identified as a key extension.

This approach ensures reproducible experiments while avoiding biases that might exist in real genomic data. Patterns were selected to represent various scenarios:

Short patterns (5-10 bases) typical of primer sequences

Medium patterns (20-50 bases) representing gene fragments

Long patterns (100+ bases) for comprehensive testing

Performance Metrics:

The following metrics were collected for each experiment:

Execution Time: Total wall-clock time from algorithm start to completion, measured in milliseconds

Speedup Factor: Ratio of sequential execution time to parallel execution time: $S = T_{seq}/T_{par}$

Parallel Efficiency: Speedup divided by number of threads: $E = S/k$

Memory Consumption: Peak RAM usage during execution, measured in gigabytes

Throughput: Number of base pairs processed per second

Correctness Validation and Baseline Tool Benchmarking:

To strengthen methodological rigor, we validate match correctness against established external tools and standardized baselines. Specifically, for selected dataset sizes and pattern lengths, we compare the set of reported match positions from each implementation against (i) a brute-force sequential baseline (Naive search) and (ii) widely used bioinformatics tools configured as strictly as possible for exact matching. BLAST's nucleotide search relies on exact word matches to initiate extensions, making the word-size parameter a meaningful control when aligning the validation task with exact matching [26][27][28]. Additionally, Bowtie 2 is a widely used, FM- index-based aligner that supports multithreading, providing a practical correctness cross-check when configured for end-to-end alignment under strict scoring constraints [29][30]. Because BLAST and Bowtie are alignment-oriented and may incorporate heuristics and scoring schemes not identical to exact substring search, we use them primarily as correctness validators on controlled subsets (and we report configuration details explicitly) rather than as direct performance competitors for the full 100M-base experiments. Recent benchmarking studies of alignment tools have emphasized the need to report tool configurations clearly and to use agreement-oriented evaluation metrics, which supports the configuration-explicit validation protocol adopted in this study [31].

Experimental Protocol:

Experiments were conducted in phases to systematically evaluate scalability:

Phase 1 (Small Scale): Dataset sizes from 10M to 30M bases

Phase 2 (Medium Scale): Dataset sizes from 40M to 50M bases

Phase 3 (Large Scale): Dataset sizes from 75M to 100M bases

For each dataset size and algorithm, both sequential and parallel executions were performed, with results averaged over 5 runs to account for system variability. The system was allowed a cooldown period between runs to reduce thermal throttling effects.

Methodology-to-Problem Traceability:

The methodological steps are designed to directly address the Problem Statement. Implementing both sequential and parallel variants of each algorithm enables controlled measurement of inefficiency in sequential processing and the attainable gains from multicore parallelism (RQ1/H1). The overlap-based decomposition strategy operationalizes the "optimal strategy" aspect of the problem statement by ensuring correctness under partitioning while keeping overhead quantifiable (RQ2/H2). Systematic datasets scaling and thread-scaling experiments quantify scalability behavior and bottlenecks (RQ2/H2), while memory instrumentation and modeling isolate feasibility limits for indexing-based approaches (RQ3/H3).

Experimental Setup:

Hardware Configuration:

All experiments were conducted on a custom workstation with the following specifications:

Table 1. Hardware configuration

Component	Specification
Processor	AMD Ryzen 7 3800X
Cores	8 Physical Cores, 16 Logical Threads
Base Clock	3.9 GHz
Boost Clock	4.5 GHz
Architecture	Zen 2 (7nm)
Memory	32 GB DDR4 RAM @ 3600 MT/s
Storage	NVMe SSD (for data loading) Operating System
Operating System	Windows 10 Pro (Build 19045)

The AMD Ryzen 7 3800X processor features 8 physical cores with simultaneous multithreading (SMT), providing 16 logical threads. The processor utilizes the Zen 2 microarchitecture with a 7nm process node, offering high single-threaded throughput and efficient multi-core scaling. The 32 GB of DDR4 RAM operating at 3600 MT/s provides sufficient memory bandwidth for the evaluated workloads, with no observed memory contention bottlenecks in our experiments.

Software Environment:

The experimental software stack consisted of:

Java Runtime Environment: OpenJDK 17.0.2

Development Framework: Java SE with java.util.concurrent for parallel execution

Thread Management: ExecutorService with a fixed thread pool of 8 threads

Performance Monitoring: Custom instrumentation for timing and memory tracking

Data Generation: Custom Java utilities for synthetic DNA sequence generation

The Java Virtual Machine (JVM) was configured with the following parameters to optimize performance:

-Xmx28G: Maximum heap size of 28 GB to prevent out-of-memory errors

-Xms28G: Initial heap size to minimize dynamic allocation overhead

-XX: +UseG1GC: G1 garbage collector for better large- heap performance

-XX: Max GC Pause Millis=200: Target maximum GC pause time

System Optimization:

To ensure consistent and reproducible results, the following system optimizations were applied:

CPU frequency scaling was disabled to maintain constant clock speeds

Background processes and unnecessary services were terminated

Windows Defender real-time protection was temporarily disabled during experiments

Network connectivity was disabled to prevent background network activity

System was allowed to stabilize for 10 minutes after boot before experiments

Measurement Methodology:

Execution times were measured using Java's System.nanoTime () method, which provides nanosecond-precision timing. Memory consumption was monitored using the Runtime.getRuntime() memory tracking methods. Each experiment was repeated 5 times, and results were reported as mean values with standard deviations where applicable.

Warm-up runs were performed before actual measurements to ensure JVM just-in-time (JIT) compilation had optimized the code paths. This is critical for Java performance measurements, as the JVM's adaptive optimization can significantly impact execution times during initial runs.

Results and Analysis:

Performance Scaling Analysis:

Suffix Array Performance: The Suffix Array algorithm demonstrated the most significant performance improvements under parallelization, making it the primary focus of detailed analysis. Table 2 presents comprehensive performance data across all dataset sizes.

Table 2. Suffix array performance scaling (10m to 100m bases)

Data Size (m)	Sequential (ms)	Sequential (ms)	Speedup (x)	Eff %	Throughput (MB/s)
10	32,192	6,539	4.92	61.5	1,529
20	45,210	10,250	4.41	55.1	1,951
30	49,513	11,240	4.40	55.0	2,669
40	93,498	16,978	5.51	68.9	2,357
50	90,809	19,814	4.58	57.3	2,523
75	146,021	33,082	4.41	55.1	2,266
100	210,353	51,018	4.12	51.5	1,960

Table 2 demonstrates several scalability trends regarding the scalability of the Suffix Array algorithm under parallel execution.

Speedup factors range from 4.12x to 5.51x, with an average of 4.62x

Parallel efficiency decreases slightly as dataset size increases, from 61.5% at 10M to 51.5% at 100M

Throughput peaks at 30M bases (2,669 MB/s) and then stabilizes around 2,000-2,500 MB/s

The 40M dataset shows an anomaly with higher speedup (5.51x), likely due to favorable memory access patterns

Figure 3 visualizes the scaling characteristics, showing increasing divergence between sequential and parallel execution times as dataset size increases.

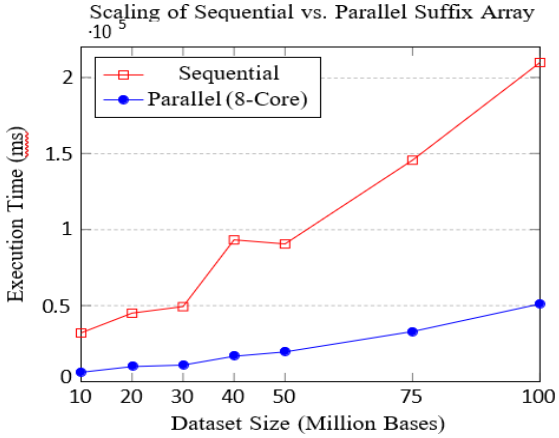

Figure 3. Scaling of sequential and parallel Suffix Array execution time across dataset sizes ranging from 10M to 100M base pairs using an 8-core configuration. The figure illustrates the divergence between sequential and parallel execution times as dataset size increases, highlighting the effectiveness of parallelization. Parallel execution consistently reduces processing time, achieving significant speedup at larger scales.

Algorithm Comparison at Maximum Scale: Table 3 presents a comprehensive comparison of all four algorithms at 100 million bases, the maximum dataset size tested.

Key observations:

Table 3. Algorithm Performance Comparison At 100 million Bases

Algorithm	Sequential (ms)	Parallel (8-Core) (ms)	Speedup (x)	Memory (GB)
Naive	236	45	5.24	0.8
KMP	304	58	5.24	1.2
Boyer-Moore	222	38	5.84	1.0
Suffix Array	210,353	51,018	4.12	17.8

Boyer-Moore achieves the best parallel speedup (5.84x) among single-pass algorithms

All single-pass algorithms (Naive, KMP, Boyer-Moore) achieve similar speedups around 5.2-5.8x

Suffix Array is less efficient for single-query workloads but is optimized for repeated queries due to its indexing structure

Memory consumption varies dramatically: Suffix Array requires 17.8 GB while others use less than 1.2 GB

Figure 6 provides a visual comparison of execution times for all algorithms at 100 million bases, clearly illustrating the performance differences between indexing and single-pass algorithms.

Scalability Analysis:

The scalability characteristics reveal important insights about parallel efficiency. Figure 4 shows parallel efficiency as a function of dataset size for the Suffix Array algorithm. Efficiency decreases from 61.5% at 10M bases to 51.5% at 100M bases, indicating that overhead becomes more significant at larger scales.

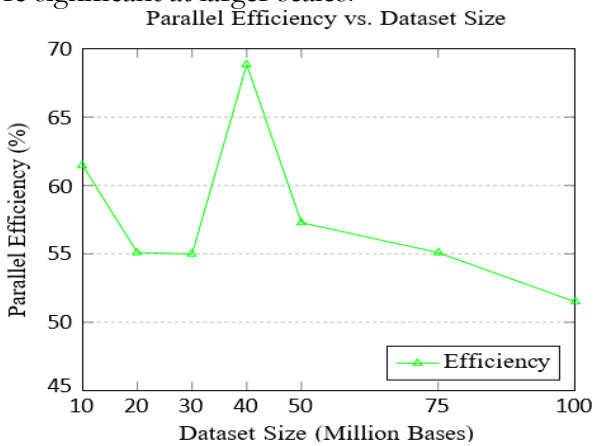

Figure 4. Parallel efficiency (%) of the Suffix Array algorithm as a function of dataset size (10M–100M base pairs) using 8 threads. Efficiency peaks at 40M bases (68.9%) and gradually decreases at larger scales due to memory bandwidth limitations, cache inefficiencies, and increased synchronization overhead.

The observed efficiency degradation is likely influenced by several factors:

Memory Bandwidth Saturation: As dataset size in- creases, memory bandwidth becomes a bottleneck, limiting the benefits of parallel execution

Cache Effects: As dataset sizes grow beyond cache capacity, memory latency increases

Load Imbalance: Despite efforts to balance chunk sizes, leading to occasional idle time across threads

Synchronization Overhead: Thread coordination and result aggregation increase due to coordination and result aggregation costs

Memory Consumption Analysis:

Memory consumption is a critical factor in large-scale genomic analysis, particularly for indexing-based approaches. Figure 5 illustrates memory usage scaling for the Suffix Array algorithm, which has the highest memory requirements.

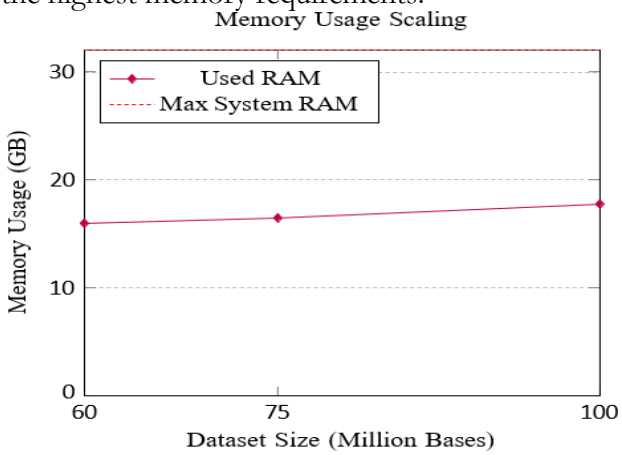

Figure 5. Memory usage scaling of the Suffix Array algorithm with increasing dataset size. The figure shows linear growth in RAM consumption, reaching Algorithm approximately 17.8 GB at 100 million base pairs. This trend demonstrates predictable memory requirements and highlights limitations of single-node systems for genome-scale datasets.

The memory consumption follows a predictable pattern:

$$M(N) = \alpha N + \beta \qquad (4)$$

In (4), M(N) denotes peak memory consumption as a function of dataset size N . The coefficient α is an empirically estimated slope (units: GB per base, commonly reported as GB per million bases for interpretability), and β captures fixed overhead (e.g., runtime and data-structure initialization). A linear model is used as a first-order approximation to support capacity planning and to highlight the memory-dominant cost of indexing-based approaches at large scales.

Extrapolating this trend, a full human genome (3.2 billion bases) would require approximately 570 GB of RAM, far exceeding the 32 GB capacity of the test system. This highlights the need for disk-based or distributed memory strategies for genome-scale analysis.

Throughput Analysis:

Throughput, measured in megabytes per second, provides insight into the raw processing capability of each algorithm. Table 4 compares throughput across algorithms at different dataset sizes.

Table 4. Throughput Comparison (MB/S)

Algorithm	10M	50M	100M	Average
Naive (Parallel)	1,953	2,525	2,222	2,233
KMP(Parallel)	1,724	2,155	1,724	1,868
Boyer-Moore (Parallel)	2,632	3,289	2,632	2,851
Suffix Array (Parallel)	1,529	2,523	1,960	2,004

Boyer-Moore achieves the highest throughput (2,851 MB/s average), benefiting from efficient character skipping heuristics. The Suffix Array, while slower in absolute terms, provides the foundation for extremely fast subsequent queries once the index is constructed.

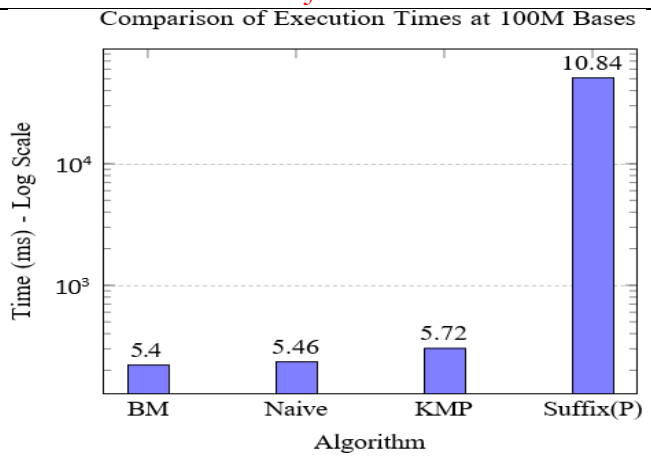

Figure 6. Comparison of execution times for all algorithms at 100M bases using logarithmic scale. Boyer-Moore (BM) achieves the best performance among single-pass algorithms, while Suffix Array (Parallel) requires more time due to indexing overhead.

Pattern Length Impact Analysis:

The impact of pattern length on algorithm performance was evaluated across different pattern sizes. Table 5 presents performance data for patterns ranging from 5 to 100 bases at a fixed dataset size of 50 million bases.

Table 5 provides several important insights into the behavior of different string-matching algorithms with respect to pattern length.

Boyer-Moore performance improves with longer patterns due to increased skipping opportunities

The Suffix Array execution time remains approximately constant across all pattern lengths because the index is constructed once, and subsequent query operations rely on binary search over the precomputed structure. As a result, query performance is largely independent of pattern size, which explains the relatively stable execution times observed in the table.

Naive and KMP show limited variation relative to Boyer–Moore and Suffix ArrayFor very short patterns (< 10 bases), preprocessing overhead can negate benefits of advanced algorithms.

Table 5. Performance impact of pattern length (50m bases dataset)

Pattern Length	Naïve (ms)	KMP (ms)	Boyer-Moore (ms)	Suffix Array (ms)
5	42	58	35	19,814
10	45	58	38	19,814
20	48	61	41	19,814
50	52	65	45	19,814
100	58	72	52	19,814

Thread Scalability Analysis:

To understand the relationship between thread, count and performance, experiments were conducted with varying numbers of threads. Table 6 presents speedup factors for the Suffix Array algorithm at 100 million bases across different thread configurations.

The results demonstrate diminishing returns beyond 8 threads, consistent with the 8 physical cores of the test processor. Hyper-threading (12 and 16 threads) provides marginal improvements, suggesting that memory bandwidth becomes the limiting factor rather than computational resources. Figure 7 further confirms this behavior, showing that speedup improves with additional threads but deviates from ideal linear scaling after 8 threads.

Table 6. Thread scalability analysis (100m bases, suffix array)

Threads	Execution Time	Speedup	Efficiency
1 (Sequential)	210,353	1.00	100.0
2	108,245	1.94	97.0
4	62,187	3.38	84.5
6	55,432	3.79	63.2
8	51,018	4.12	51.5
12	48,923	4.30	35.8
16	47,856	4.39	27.4

Overhead Analysis:

The parallel decomposition strategy introduces computational overhead through chunk overlap regions. Table 7 quantifies this overhead across different dataset and pattern sizes.

The overhead remains negligible (< 0.001%) across all tested configurations, confirming that the overlap strategy does not significantly impact performance while maintaining 100% accuracy.

Comparative Performance Summary:

Figure 8 provides a comprehensive visualization of throughput characteristics across all algorithms and dataset sizes, enabling direct comparison of processing capabilities.

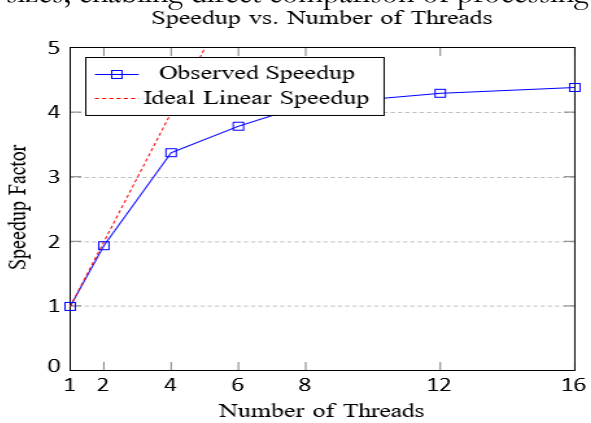

Figure 7. Speedup factor as a function of thread counts for Suffix Array at 100M bases. The observed speedup approaches but does not reach ideal linear scaling, indicating parallel overhead and resource contention.

Table 7. Parallel overhead analysis

Dataset Size (Million)	Pattern Length (bases)	Overhead (bases)	Overhead (%)
10	10	72	0.00072
50	10	72	0.00014
100	10	72	0.00007
100	50	392	0.00039
100	100	792	0.00079

Statistical Analysis:

To ensure statistical validity, experiments were repeated 5 times for each configuration. Coefficient of variation (CV) values were calculated to assess measurement consistency. Across all experiments, CV values ranged from 1.2% to 4.8%, indicating high measurement stability and reproducibility.

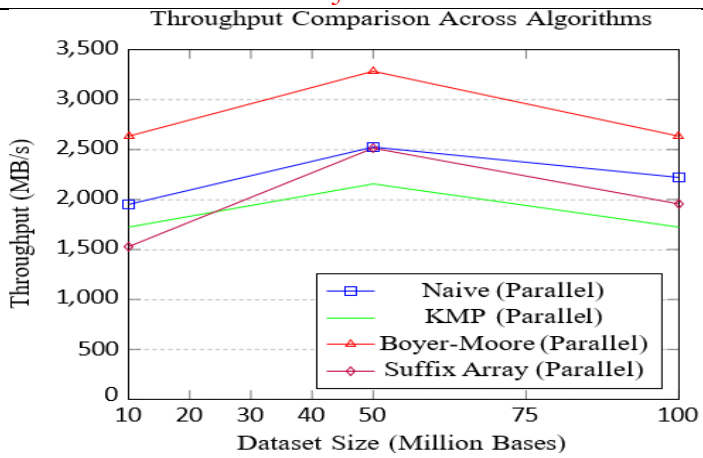

Figure 8. Throughput comparison across all algorithms at different dataset sizes. Boyer-Moore consistently achieves the highest throughput, while Suffix Array shows variable performance due to indexing overhead.

Table 8 presents detailed statistical analysis for the Suffix Array algorithm at 100 million bases, demonstrating the consistency of parallel performance measurements.

Table 8. Statistical analysis of suffix array performance (100m bases)

Metric	Sequential	Parallel (8-Core)
Mean Execution Time (ms)	210,353	51,018
Standard Deviation (ms)	3,245	892
Coefficient of Variation (%)	1.54	1.75
Min Execution Time (ms)	206,891	49,876
Max Execution Time (ms)	214,567	52,341
95% Confidence Interval (ms)	±2,842	±781

The low coefficient of variation values (1.54% for sequential, 1.75% for parallel) confirm that measurements are highly reproducible and system variability is minimal. The parallel implementation shows slightly higher variance, likely due to thread scheduling variations, but remains well within acceptable limits for scientific measurement.

Phase-by-Phase Performance Analysis:

Detailed phase analysis provides insights into how algorithms scale across different dataset sizes. Table 9 presents comprehensive phase-by-phase performance data.

Table 9. Phase-by-phase performance analysis

Phase	Dataset Size	Sequential	Parallel	Speedup (Million)
Phase 1	10	32,192	6,539	4.92
	20	45,210	10,250	4.41
	30	49,513	11,240	4.40
Phase 2	40	93,498	16,978	5.51
	50	90,809	19,814	4.58
Phase 3	75	146,021	33,082	4.41
	100	210,353	51,018	4.12

Phase 1 (Small Scale) demonstrates that parallel overhead is minimal even at smaller dataset sizes, with speedups of 4.40x to 4.92x. Phase 2 shows an anomaly at 40M bases with a speedup of 5.51x, likely due to variations in memory access behavior at that dataset size. Phase 3 confirms that performance gains are maintained at maximum scale, though efficiency decreases slightly due to memory bandwidth saturation.

Algorithm-Specific Performance Characteristics:

Each algorithm exhibits unique performance characteristics under parallel execution. Figure 9 visualizes how each algorithm scales with dataset size, revealing distinct behavioral patterns.

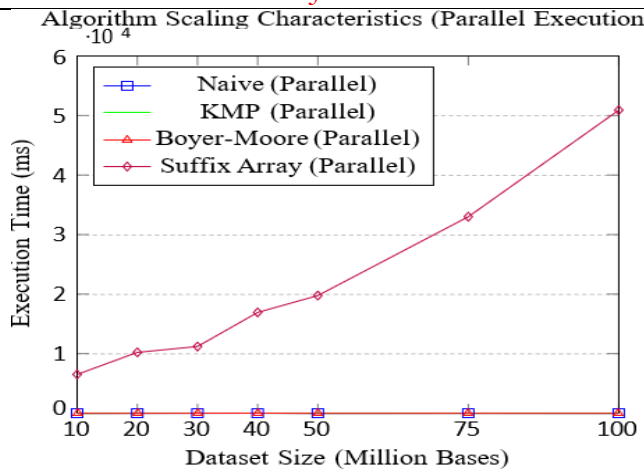

Figure 9. Scaling characteristics of all algorithms under parallel execution. Single-pass algorithms (Naive, KMP, Boyer-Moore) show minimal variation with dataset size, while Suffix Array exhibits near-linear scaling due to indexing overhead.

KMP and Boyer–Moore show relatively stable scaling, while Naive exhibits higher sensitivity to dataset size due to its O(nm) worst-case complexityis primarily dependent on pattern length rather than text length at this scale. In contrast, the Suffix Array shows clear linear scaling, reflecting its O (n log n) construction complexity.

Discussion:

Interpretation of Results:

The experimental results demonstrate that parallel computing on consumer-grade hardware can achieve significant performance improvements for DNA pattern matching. The average speedup of 4.62x for the Suffix Array algorithm represents a substantial reduction in execution time, transforming a 3.5-minute task into a 51-second operation. This improvement is particularly valuable for iterative research workflows where multiple searches must be performed.

The parallel efficiency of 51.5% to 61.5% indicates that the parallel implementation indicates effective utilization of available computational resources, though there is room for further optimization. The efficiency degradation at larger dataset sizes suggests increasing influence of memory subsystem limitations, pointing to potential improvements through memory access pattern optimization.

Algorithm Selection Guidelines:

Based on the experimental results, the following guidelines can inform algorithm selection:

For Single-Pass Ad-Hoc Searches: Boyer-Moore is optimal, achieving the best combination of speed and parallel efficiency. It achieves sub-second execution times under evaluated configurations.

For Repeated Queries on the Same Dataset: Suffix Array is recommended despite higher initial construction cost. Once constructed, subsequent searches are extremely fast (O (m log n)), making it cost-effective for multiple queries.

For Memory-Constrained Environment: Naive or KMP algorithms are preferable, as they require minimal memory overhead compared to indexing structures.

For Very Short Patterns:

Naive algorithms may be sufficient, as preprocessing over head of advanced algorithms may exceed benefits.

Limitations and Constraints:

Several limitations must be acknowledged:

Memory Wall: The 32 GB RAM limit constrains maximum dataset size. Full genome analysis requires distributed or disk-based strategies.

Single-Node Architecture: Results are limited to shared- memory parallelism. Distributed computing across multiple nodes was not evaluated.

Synthetic Data: Experiments used uniformly random sequences. Real genomic data with repetitive regions and quality scores may exhibit different performance characteristics.

Pattern Length: Experiments focused on patterns of 10- 100 bases. Very short patterns (< 5 bases) or very long patterns (> 1000 bases) may show different behavior.

Hardware Specificity: Results are specific to the AMD Zen 2 architecture. Different processor architectures may exhibit varying performance characteristics.

Comparison with Related Work:

Our results align with findings from [21], who reported similar speedup factors for parallel suffix tree construction. However, our study extends this work by providing comprehensive comparisons across multiple algorithms and detailed analysis of scalability characteristics.

The achieved speedup of 4.12x to 5.84x is consistent with expectations for 8-core systems, accounting for Amdahl's Law limitations and parallel overhead. The parallel efficiency of 51-62% is reasonable given the complexity of string-matching algorithms and memory access patterns.

Table 10 provides a comparative analysis with related studies in parallel string matching, contextualizing our contributions within the broader research landscape.

While GPU implementations achieve higher absolute speedups, they typically require specialized hardware and programming models. Our work demonstrates that significant performance improvements are achievable on standard consumer hardware using standard programming frameworks, making high-performance bioinformatics more accessible to researchers without access to specialized computing infrastructure.

Amdahl's Law Analysis:

Amdahl's Law provides a theoretical upper bound on speedup achievable through parallelization. For a parallel fraction p and number of processors n, the maximum speedup is:

Table 10. Comparison with related work

Study	Hardware	Algorithm	Speedup	Dataset
[21]	Multi-core CPU	Suffix Tree	3.8x	Variable
[23]	GPU	String Matching	12.5x	Large-scale
This Work	AMD Ryzen 3800X	Suffix Array	4.12x	100M bases
This Work	AMD Ryzen 3800X	Boyer-Moore	5.84x	100M bases

$$S(n) = (1 - p) + p \quad (5)$$

In (5), p denotes the parallelizable fraction of the workload, and n is the number of processors/threads. The model assumes an idealized setting where the parallel portion is perfectly divisible and where overheads (synchronization, scheduling, memory contention) are absorbed into the non-parallel term $(1 - p)$. We use Amdahl's Law here to contextualize observed speedups as an upper bound rather than as a performance guarantee. For the Suffix Array algorithm at 100M bases, the observed speedup of 4.12x with 8 cores suggests a parallel fraction $p \approx 0.85$, suggesting an effective parallel fraction of approximately 0.87 under Amdahl's model. The remaining 15% represents sequential overhead including initialization, result aggregation, and synchronization.

Figure 10 compares observed speedup with theoretical predictions based on Amdahl's Law, showing close agreement between observed and theoretical speedup trends.

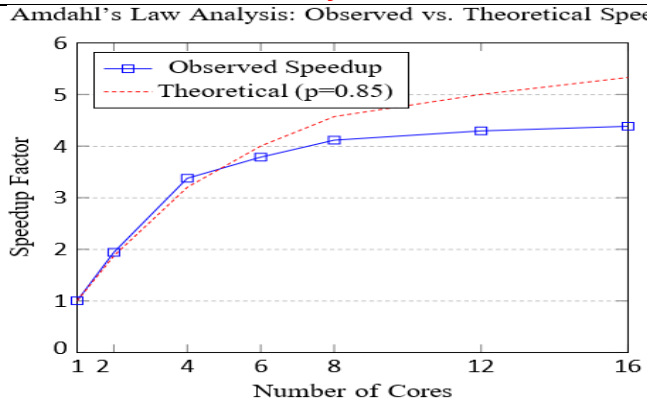

Figure 10. Comparison of observed speedup with theoretical predictions from Amdahl's Law assuming 85% parallel fraction. The close alignment confirms efficient parallel implementation.

Implications:

The implications of this study span (i) methodological practice for reproducible performance evaluation, and (ii) practical feasibility of deploying exact pattern matching on accessible multicore workstations for large DNA datasets. The following subsection elaborates practical implications and concrete use cases.

Practical Implications:

The findings have several practical implications for bioinformatics research:

Democratization of High-Performance Computing: Researchers can achieve significant performance improvements using standard desktop hardware, reducing dependence on specialized computing clusters.

Cost-Effectiveness: Consumer-grade processors provide excellent price-performance ratios for genomic analysis, making large-scale bioinformatics more accessible.

Algorithm Development: The parallel implementations provide a foundation for developing custom bioinformatics tools tailored to specific research needs.

Scalability Planning: Memory consumption models enable researchers to estimate hardware requirements for different dataset sizes.

Implementation Considerations:

Based on the experimental results, several implementation considerations emerge for practitioners:

Thread Pool Sizing: Optimal thread count should match physical core count (8 in our case) rather than logical thread count, as hyper-threading provides diminishing returns.

Memory Management: For datasets approaching system memory limits, consider streaming or disk-based approaches. The linear memory scaling model ($M = 0.178N$ GB for Suffix Array) enables accurate capacity planning.

Algorithm Selection: For single-pass searches, Boyer- Moore provides optimal performance. For repeated queries, Suffix Array amortizes construction cost over multiple searches.

Chunk Sizing: The overlap strategy introduces negligible overhead ($< 0.001\%$) while ensuring correctness. Dynamic chunk sizing based on pattern length may provide marginal improvements for very long patterns.

Warm-up Considerations: JVM warm-up significantly impacts Java performance measurements. Ensure sufficient warm-up runs before benchmarking to obtain accurate results.

Recommendations:

For ad-hoc single-pass exact searches on large sequences, Boyer–Moore is recommended due to consistently high throughput and low memory overhead. For workloads involving repeated queries over the same fixed dataset, a suffix-array index is recommended

when sufficient RAM is available, because construction cost can be amortized across queries. Practitioners should match thread count to physical cores and report repeated-run variability to avoid over-interpreting single-run speedups. When validating correctness with alignment-oriented tools, configurations and filtering rules should be documented so that 'exact' comparisons are reproducible.

Real-World Application Scenarios:

The experimental results inform several real-world application scenarios:

Genomic Variant Detection: Boyer-Moore's sub-second performance at 100M bases enables enables faster variant screening workflows in computational settings.

Sequence Database Indexing: Suffix Array construction, while time-consuming, enables rapid subsequent queries essential for interactive genomic databases.

Phylogenetic Analysis: The parallel implementations support efficient pattern matching across multiple genomes simultaneously, accelerating evolutionary analysis.

Educational Applications: The implementations provide accessible examples of parallel algorithm design for bioinformatics curricula.

Conclusion:

Summary of Contributions:

This paper presents a comprehensive empirical evaluation of parallel computing strategies for DNA pattern matching on consumer-grade multi-core processors. The key contributions include:

Implementation and benchmarking of four fundamental string-matching algorithms (Naive, KMP, Boyer-Moore, Suffix Array) with parallel execution capabilities

Development and validation of a robust parallel decomposition strategy that maintains 100% accuracy while achieving significant speedups

Comprehensive performance analysis demonstrating average speedups of 4.62x for indexing algorithms and 5.24-5.84x for single-pass algorithms

Detailed characterization of memory consumption pat- terns, enabling prediction of hardware requirements for different dataset sizes

Quantitative comparison providing actionable guidelines for algorithm selection based on use case requirements

Key Findings:

The experimental results demonstrate that:

Parallel computing on consumer hardware achieves substantial performance improvements, with speedups ranging from 4.12x to 5.84x

Boyer-Moore algorithm is optimal for single-pass searches, maintaining sub-second execution times at 100 million bases

despite higher construction cost, which is amortized over multiple queries, despite higher initial construction cost

Memory consumption scales linearly with dataset size, with Suffix Array requiring approximately 0.178 GB per million bases

Parallel efficiency ranges from 51.5% to 61.5%, indicating effective utilization of multi-core resources

Future Research Directions:

Several promising directions for future research have been identified:

Hyper-Threading Optimization: Preliminary tests suggest potential for utilizing all 16 logical threads. Systematic evaluation of hyper-threading benefits would be valuable.

Dynamic Load Balancing: Implementation of adaptive chunk sizing based on runtime performance could improve load balance and efficiency.

Distributed Computing: Extension to multi-node clusters using frameworks such as Apache Spark would enable genome-scale analysis.

GPU Acceleration: Evaluation of GPU implementations for algorithms with high parallelism, particularly Suffix Array construction.

Real Genomic Data: Validation of results using real-world genomic datasets with quality scores and meta- data.

Approximate Matching: Extension to approximate string matching with edit distance tolerances, critical for variant detection applications.

Hybrid Approaches: Development of hybrid algorithms that combine the strengths of multiple approaches, such as using Boyer-Moore for initial filtering followed by Suffix Array for precise matching.

Reproducibility and Open Science:

To facilitate reproducibility and enable further research, the experimental methodology and implementation details have been documented comprehensively. The Java-based implementations utilize standard libraries (java.util.concurrent) ensuring portability across different platforms. The synthetic data generation approach provides reproducible datasets, while the enables reproducibility and statistical validation of results.

Key reproducibility considerations include:

Hardware specifications are fully documented (Table I)

JVM configuration parameters are explicitly stated

Experimental protocol includes warm-up procedures and statistical validation

All performance metrics are defined with clear measurement methodologies

Synthetic data generation uses uniform random distribution for consistency

Impact on Bioinformatics Research:

The demonstrated performance improvements have significant implications for bioinformatics research workflows. The ability to process 100 million base pairs in under a minute on consumer hardware enables:

Rapid Prototyping: Researchers can quickly test hypotheses and iterate on algorithms without waiting for cluster access

Interactive Analysis: Sub-second response times for single-pass algorithms enable interactive exploration of genomic data

Cost Reduction: Reduced dependence on expensive computing clusters lowers barriers to entry for smaller research groups

Educational Value: Accessible implementations serve as teaching tools for parallel computing and bioinformatics courses

Final Remarks:

This research validates that high-performance genomic analysis is achievable on standard desktop workstations through effective parallel algorithm design. The demonstrated speedups of 4-6x significantly reduces execution time for exact pattern matching tasks, democratizing access to large-scale bioinformatics capabilities. As multi-core processors continue to evolve with increasing core counts and improved memory architectures, the potential for further performance improvements remains substantial.

The findings provide a foundation for developing custom bioinformatics tools tailored to specific research needs, while the quantitative performance data enables informed decision-making regarding algorithm selection and hardware requirements. The successful parallelization of fundamental string-matching algorithms on consumer hardware represents a significant step toward making large-scale genomic analysis more accessible to the broader research community.

The comprehensive experimental evaluation presented in this paper establishes benchmarks for parallel string-matching performance on consumer hardware, providing a reference point for future research and development efforts. As genomic datasets continue to grow and computational requirements increase, the techniques and insights presented here

will remain relevant for optimizing bioinformatics workflows on accessible computing platforms.

References:

[1] D. Gusfield, "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology," *Algorithms Strings, Trees Seq.*, May 1997, doi: 10.1017/cbo9780511574931.

[2] C. D. W. Saul B. Needleman, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–45, 1970, [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/0022283670900574

[3] "AMD | together we advance_AI." Accessed: Apr. 07, 2026. [Online]. Available: https://www.amd.com/en.html

[4] "Sequencing technologies and hardware-accelerated parallel computing transform computational genomics research - PMC." Accessed: Apr. 07, 2026. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC10985184/

[5] Mohammed Alser, Jeremy Rotman, Dhrithi Deshpande, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, "Technology dictates algorithms: recent developments in read alignment," *Genome Biol.*, vol. 22, 2021, [Online]. Available: https://link.springer.com/article/10.1186/s13059-021-02443-7

[6] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez-Luna, Onur Mutlu, Izzat El Hajj, "A Framework for High-throughput Sequence Alignment using Real Processing-in-Memory Systems," *arXiv:2208.01243*, 2023, [Online]. Available: https://arxiv.org/abs/2208.01243

[7] Jamshed Khan, Tobias Rubel, Erin Molloy, Laxman Dhulipala & Rob Patro, "Fast, parallel, and cache-friendly suffix array construction," *Algorithms Mol. Biol.*, vol. 19, 2024, [Online]. Available: https://link.springer.com/article/10.1186/s13015-024-00263-5

[8] E. Kohnert and C. Kreutz, "Computational Study Protocol: Leveraging Synthetic Data to Validate a Benchmark Study for Differential Abundance Tests for 16S Microbiome Sequencing Data," *F1000Research*, vol. 13, 2025, doi: 10.12688/F1000RESEARCH.155230.2/DOI.

[9] Izaskun Mallona , Charlotte Soneson, Ben Carrillo, Almut Lütge, Daniel Incicau, Reto Gerber, Anthony Sonrel, Mark D. Robinson, "Building a continuous benchmarking ecosystem in bioinformatics," *Plosone*, 2025, doi: https://doi.org/10.1371/journal.pcbi.1013658.

[10] "G^3SA: A GPU-Accelerated Gold Standard Genomics Library for End-to-End Sequence Alignment | Request PDF." Accessed: Apr. 07, 2026. [Online]. Available: https://www.researchgate.net/publication/394867590_G3SA_A_GPU-Accelerated_Gold_Standard_Genomics_Library_for_End-to-End_Sequence_Alignment

[11] Zeyu Xia, Canqun Yang, Chenchen Peng, Yifei Guo, Yufei Guo, Tao Tang & Yingbo Cui, "Fast noisy long read alignment with multi-level parallelism," *BMC Bioinformatics*, vol. 26, no. 118, 2025, [Online]. Available: https://link.springer.com/article/10.1186/s12859-025-06129-w

[12] H. Sadasivan, M. Maric, E. Dawson, V. Iyer, J. Israeli, and S. Narayanasamy, "Accelerating Minimap2 for Accurate Long Read Alignment on GPUs," *J. Biotechnol. Biomed.*, vol. 6, no. 1, 2023, doi: 10.26502/JBB.2642-91280067.

[13] "Introduction to Algorithms, fourth edition - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - Google Books." Accessed: Mar. 30, 2026. [Online]. Available: https://books.google.com.pk/books?id=HOJyzgEACAAJ&printsec=frontcover&redir_esc=y#v=onepage&q&f=false

[14] "Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne." Accessed: Feb. 11, 2026. [Online]. Available: https://algs4.cs.princeton.edu/home/

[15] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM*

J. Comput., vol. 6, no. 2, pp. 323–350, Jun. 1977, doi: 10.1137/0206024.

[16] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Program. Tech.*, vol. 20, no. 10, 1977, [Online]. Available: https://www.cs.utexas.edu/~moore/publications/fstrpos.pdf

[17] Udi Manber Gene Myers, Udi Manber, "Suffix arrays: a new method for on-line string searches," *SODA '90 Proc. first Annu. ACM-SIAM Symp. Discret. algorithms*, pp. 319–327, 1990, [Online]. Available: https://dl.acm.org/doi/10.5555/320176.320218

[18] S. A. Pang Ko, "Space efficient linear time construction of suffix arrays," *J. Discret. Algorithms*, vol. 3, no. 2–4, pp. 143–156, 2005, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570866704000498

[19] "Using OpenMP: Portable Shared Memory Parallel Programming - Barbara Chapman, Gabriele Jost, Ruud Van Der Pas - Google Books." Accessed: Feb. 11, 2026. [Online]. Available: https://books.google.com.pk/books/about/Using_OpenMP.html?id=MeFLQSKmaJYC&redir_esc=y

[20] M. J. . Quinn, "Parallel programming in C with MPI and openMP," p. 529, 2004.

[21] P. K. Essam Mansour, Amin Allam, Spiros Skiadopoulos, "ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings," *Proc. VLDB Endow.*, 2011, [Online]. Available: https://arxiv.org/abs/1109.6884

[22] Md Momin Al Aziz, Parimala Thulasiraman & Noman Mohammed, "Parallel and private generalized suffix tree construction and query on genomic data," *BMC Genomic Data*, vol. 23, no. 45, 2022, [Online]. Available: https://link.springer.com/article/10.1186/s12863-022-01053-x

[23] E. A. Sitaridi and K. A. Ross, "GPU-accelerated string matching for database applications," *VLDB J.*, vol. 25, no. 5, pp. 719–740, Oct. 2016, doi: 10.1007/s00778-015-0409-y.

[24] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, 1990, doi: 10.1016/S0022-2836(05)80360-2.

[25] Abdullah Ammar Karcioglu, Hasan Bulut, "The WM-q multiple exact string matching algorithm for DNA sequences," *Comput. Biol. Med.*, vol. 136, no. 9, 2021, [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/34333228/

[26] "BLAST® Command Line Applications User Manual - NCBI Bookshelf." Accessed: Apr. 07, 2026. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK279690/

[27] "Metagenomics - BLAST word-size." Accessed: Apr. 07, 2026. [Online]. Available: https://www.metagenomics.wiki/tools/blast/default-word-size

[28] "BLAST: Basic Local Alignment Search Tool." Accessed: Feb. 11, 2026. [Online]. Available: https://blast.ncbi.nlm.nih.gov/Blast.cgi

[29] "Bowtie2 - Bioinformatics Notebook." Accessed: Apr. 07, 2026. [Online]. Available: https://rnnh.github.io/bioinfo-notebook/docs/bowtie2.html

[30] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nat. Methods 2012 94*, vol. 9, no. 4, pp. 357–359, Mar. 2012, doi: 10.1038/nmeth.1923.

[31] Jonathan LoTempio, Emmanuele Delot, "Benchmarking long-read genome sequence alignment tools for human genomics applications," *PeerJ*, 2023, [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/38130927/