Ensino que te conecta

BACHARELADO EM

# CIÊNCIA DA COMPUTAÇÃO

## UM TÍTULO MUITO INTERESSANTE PARA UM TRABALHO

**LINUS TORVALDS**

Brasília - DF, 2025

**LINUS TORVALDS**

# UM TÍTULO MUITO INTERESSANTE PARA UM TRABALHO

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção de grau de Bacharel em Ciência da Computação, pelo Instituto Brasileiro de Ensino, Desenvolvimento e Pesquisa (IDP).

**Orientador**
Dr. Alan Turing

Brasília - DF, 2025

**LINUS TORVALDS**

# UM TÍTULO MUITO INTERESSANTE PARA UM TRABALHO

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção de grau de Bacharel em Ciência da Computação, pelo Instituto Brasileiro de Ensino, Desenvolvimento e Pesquisa (IDP).

Aprovado em 08/05/2025

**Banca Examinadora**

---

Dr. Alan Turing- Orientador

---

Dr. Dom Quixote de la Mancha- Examinador interno

---

Dra. Anna Karienina- Examinadora Externa

# DEDICATÓRIA

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

# AGRADECIMENTOS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

# ABSTRACT

O Abstract é a tradução do resumo para a língua inglesa. Ele tem exatamente a mesma função e estrutura: apresentar, de forma breve e precisa, os elementos essenciais do trabalho — tema, objetivo, metodologia, resultados e conclusões. Assim como no resumo, o abstract deve ser elaborado em um único parágrafo, sem citações, com linguagem formal, objetiva e impessoal. Ao final do abstract, também devem ser inseridas as keywords, que são as palavras-chave em inglês.

**Keywords:** LaTeX, metodologia científica, trabalho de conclusão de curso, paper, technology.

# RESUMO

O Resumo é uma apresentação concisa do conteúdo do trabalho. Deve sintetizar, de forma clara e objetiva, os principais elementos do texto: tema, objetivo, metodologia, resultados e conclusões. O objetivo do resumo é permitir que o leitor compreenda rapidamente o que foi desenvolvido no trabalho, sem precisar lê-lo na íntegra. De acordo com as normas da ABNT (NBR 6028), o resumo deve ser elaborado em um único parágrafo, sem citações, com a utilização de linguagem impessoal e verbo na voz ativa. Geralmente, possui entre 150 e 500 palavras, dependendo do regulamento da instituição. Além do texto, o resumo deve apresentar de três a cinco palavras-chave, separadas por ponto e finalizadas com ponto final, que representem os principais assuntos abordados no trabalho.

**Palavras-chave:** LaTeX, metodologia científica, trabalho de conclusão de curso, artigo, tecnologia.

# LIST OF FIGURES

# LIST OF TABLES

# CONTENTS


1

# 1 INTRODUCAO

## 1.1 TESTE

### 1.1.1 Testando

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Table 1: Tabela com largura fixa

| Coluna 1 | Coluna 2 | Coluna 3 |
|---|---|---|
| Dado A | Dado B | Dado C |

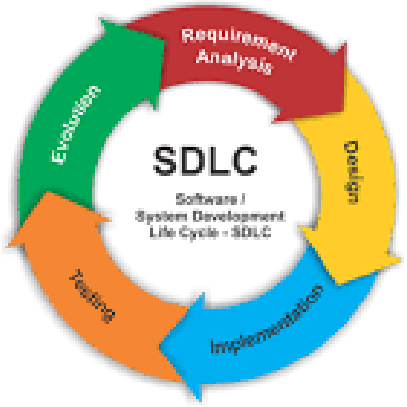


Figure 1: Legenda da figura explicando seu conteúdo.

# 2

# 2 FUNDAMENTAÇÃO TEÓRICA

## 2.1 PARALELISMO EM COMPUTAÇÃO

Advancements in semiconductor technology have historically driven computational power growth. For decades, the performance of single-core processors followed the trend predicted by Moore's Law, which forecast the doubling of transistors on a chip every eighteen months. However, in the mid-2000s, the hardware industry encountered significant physical barriers, such as excessive power consumption and challenges in heat dissipation, which limited the increase in processor clock frequency. The response to these technological limitations was a change in processor design: instead of focusing on making a single core faster, the emphasis shifted to integrating multiple processing cores onto a single chip. Thus emerged multi-core processors, which became the dominant architecture in modern computers. This transition opened up new possibilities for performance enhancement; however, to take advantage of the performance gains offered by multi-core processors, programs must be intentionally developed or adapted to execute tasks in parallel. [1].

### 2.1.1 Fundamentals of Parallelism

Parallelism in computing refers to the simultaneous execution of multiple tasks or parts of the same task, with the primary goal of reducing a program's total execution time. It is crucial to distinguish parallelism from concurrency. Concurrency occurs when multiple tasks are in progress at a given time, but may not be executing simultaneously. A single-core system can exhibit concurrency by rapidly switching between the execution of different tasks, simulating parallelism through preemption. True parallelism, on the other hand, requires hardware with multiple processing units. [2].

### 2.1.2 Parallel Computing Architectures

Parallel computer architectures are classified according to various criteria. A prominent taxonomy, proposed by Flynn, categorizes these architectures based on instruction and data streams [3]:

- **SISD (Single Instruction, Single Data)**: Represents a conventional single-processor

architecture, where a single instruction operates on a single data stream.

- **SIMD (Single Instruction, Multiple Data)**: Involves multiple processing units that execute the same instruction concurrently on different data elements.
- **MISD (Multiple Instruction, Single Data)**: Characterizes systems where multiple processing units apply distinct instructions on the same data stream.
- **MIMD (Multiple Instruction, Multiple Data)**: Involves multiple processing units that execute distinct instructions on distinct data streams independently. This paradigm is directly related to multi-core architectures.

Within the MIMD category, systems are further distinguished based on their memory organization [1]:

- **Shared-Memory Systems**: In these systems, multiple processors share a physical memory address space. Inter-processor communication (IPC) is typically achieved by reading from and writing to shared variables in memory or through message passing mechanisms. Multi-core processors are the most common and widely available example of this architecture, where all cores on a chip share access to main memory.
- **Distributed-Memory Systems**: Each processor in these systems possesses their own locally accessible memory, and communication between processors is usually done by exchanging messages across an interconnection network. Computer clusters are examples of this architectural model.

The present work focuses on the acceleration of algorithms in environments with multi-core processors, which fall into the shared memory MIMD category. A multi-core processor (MCP) is a single chip that contains two or more central processing units (CPUs) called ‘cores’. Each of these cores is capable of reading and executing program instructions independently, essentially functioning as a complete processor.

### 2.1.3 Parallel Programming With Threads

Within shared-memory systems, such as multi-core processors, software-level parallelism is commonly achieved through the execution of multiple threads within a single process.

**Processes vs. Threads**

A process is an instance of an executing program, an entity possessing its own dedicated virtual address space, file descriptors, environment variables, and other Operating System (OS) resources, all isolated from other processes [1].

A thread, on the other hand, is a lighter unit of execution started by a process. The same process can control, concurrently or in parallel, multiple threads. A key characteristic is that all threads within a process share its virtual address space, which encompasses executable code, global data, and the heap segment. However, each thread maintains its private resources, which are essential for its independent execution context:

- A unique identifier (thread ID).
- A set of CPU registers, including the program counter (PC).
- A separate execution stack, used for local variables and function call control.
- Thread-specific state, such as signal masking and scheduling priority.

**POSIX Thread Model (Pthreads)**

The POSIX Thread Model, commonly referred to as “Pthreads”, is a standard that defines an application programming interface (API) for thread creation and management, as specified by IEEE Std 1003.1c-1995. Pthreads provides a portable suite of data types, functions, and constants that allows developers to create multithreaded applications that are compilable and executable across diverse POSIX-compliant operating systems (e.g., Linux, macOS, and various Unix variants). The Pthreads API provides functionality for managing the thread lifecycle—including creation, termination, and joining—and, critically, incorporates synchronization primitives. These primitives, such as mutexes and condition variables, are indispensable for orchestrating access to shared resources within shared-memory environments [1, 2].

### 2.1.4 Parallel Performance Evaluation Metrics

Performance evaluation is fundamental to understanding the effectiveness of a parallel program. The following metrics compare the execution time of the parallel program with its equivalent sequential counterpart [4].

**Execution Time and Overhead**

The performance of a program is measured by its execution time. For parallel programs, the following terms are defined:

- **Ts**: The execution time of the fastest known sequential algorithm for a given problem, executed on a single processor unit.
- **Tp**: The execution time of the parallel program utilizing p processors.

Ideally, `Tp` is expected to be substantially less than `Ts`. However, parallel execution introduces *overhead*. This encompasses the time spent by processors on tasks that do not directly contribute to problem resolution, including communication, synchronization, load balance, and other computations inherent to the parallel implementation.

**Speedup**

Speedup (`S`) measures the performance gain achieved through parallel execution relative to sequential execution and it if referenced as equation 1. It is defined as the ratio of sequential execution time to parallel execution time:

$$S = \frac{T_s}{T_p}$$

An ideal, or linear, speedup would be `S = p`. Amdahl's Law, establishes a theoretical upper bound on the speedup achievable by parallelizing a program with a fixed problem size. `s` represents the fraction of a program's sequential execution time that is inherently serial (non-parallelizable), and `f` is the parallelizable fraction (`s + f = 1`), the maximum speedup on `p` processors is given by:

$$S \leq \frac{1}{s + f/p}$$

In the limit, as the number of processors (`p`) tends to infinity, the maximum speedup is bounded by `1/s`. This implies that even a small sequential fraction (`s` large) can severely limit the total speedup, regardless of the number of processors available. A complementary perspective is offered by the Gustafson-Barsis Law. This law argues that, that a speedup approximately linear

$$S_{\text{escalado}} = s + (1 - s) \cdot p$$

is achievable for problems that se beneficiam desta escalabilidade. This formulation suggests that for problems that can be scaled, it is possible to obtain speedup approximately linear with the number of processors, even in the presence of a serial fraction. Both Amdahl's and Gustafson's laws are important for understanding the limits and potential of parallelism in different scenarios.

**Efficiency**

Efficiency (`E`) measures the effectiveness of processor utilization in a parallel program. It is defined as the ratio of speedup to the number of processors `p`:

$$E = \frac{S}{p} = \frac{T_s}{p \cdot T_p}$$

Efficiency values range between 0 and 1 (inclusive), often expressed as a percentage (0% to 100%). An efficiency of 1 (or 100%) signifies ideal, linear speedup, implying that all processors are contributing productively throughout the execution. In practice, efficiency typically diminishes as the processor count increases, primarily due to the escalating impact of parallel overhead.

**Scalability**

Scalability, in the context of a parallel program and its underlying architecture, denotes its capacity to sustain performance (often measured by efficiency) as the number of processors and/or the problem size increases.

**Strong Scalability**: Measures how execution time varies with an increasing number of processors for a fixed total problem size. The objective is to maintain relatively constant efficiency as $\mathtt{p}$ increases for a fixed $\mathtt{W}$. Amdahl's Law is particularly relevant to this scenario.

**Weak Scalability**: Evaluates performance as both the number of processors $\mathtt{p}$ and the total problem size $\mathtt{W}$ increase, such that the problem size per processor $\mathtt{W/p}$ remains constant. The goal is to maintain efficiency as $\mathtt{p}$ and $\mathtt{W}$ scale proportionally. Gustafson's Law is most pertinent in this context.

A system is considered scalable if the efficiency remains above a certain reasonable limit as the platform (number of processors) and the problem grow.

## 2.2 TRIES (PREFIX TREES)

This section elucidates the Trie data structure, detailing its defining properties, core operational mechanics, performance aspects, and its significance as a foundational element in advanced string processing algorithms, particularly the Aho-Corasick algorithm.

### 2.2.1 Definition and Basic Structure

A Trie—also known as a prefix tree—is a specialized tree-like data structure optimized for the efficient storage and retrieval of a dynamic set of strings. Edward Fredkin introduced the term "trie" in 1960, highlighting its primary application in information retrieval [5, 6].

Tries store strings by organizing nodes so that each path from the root to a node represents a unique prefix. In a standard $R$-way trie, each node can branch to $R$ children, where $R$ denotes the size of the character alphabet. The existence of a parent-child edge highlights two key aspects of tries:

- The edge's position (index) within the parent node's array of $R$ possible child links directly corresponds to a unique character in the alphabet. For instance, index 0 might correspond to 'a', index 1 to 'b', and so on, according to a predefined mapping.
- The character associated with this edge extends the prefix represented by the parent; this implies that at least one word (or key) in the trie contains this character sequence.

To signify that such a prefix also constitutes a complete key present in the trie, the

node reached at the end of this path is typically marked by storing an associated non-null value or a boolean flag. The characters themselves are implicitly encoded by the trie's structure rather than being explicitly stored within each node.

### 2.2.2 Core Operations

Tries support several primary operations, including search, insertion, and deletion [7].

**Search**: Locating a key involves traversing the trie starting from the root, following the successive characters of the target key. Each character dictates which child link to follow. The search operation succeeds if a complete path corresponding to all characters of the key exists and the node reached at the end of this path is appropriately marked as representing a complete key. The search fails if, at any point, a required link is null (i.e., no path for the current character exists) or if the path is fully traversed but the final node is not marked as a complete key.

**Insertion**: The insertion process begins similarly to a search, by traversing the trie according to the characters of the key to be added. If the path for the key does not extend to its full length (i.e., a null link is encountered), new nodes are created and linked to form the remainder of the path. The node corresponding to the final character of the inserted key is then marked to indicate that it is a complete key in the set.

**Deletion**: Deleting a key first involves locating the node corresponding to the key's final character and removing its end-of-word marker. If there are no child links (meaning it is not a prefix for any other key), it can be deleted. This deletion may proceed recursively: when a child node is removed, if its parent has no other children and the parent node is not marking a key, the parent node becomes redundant and can be deleted. This process continues up to the root as long as parent nodes become redundant.

### 2.2.3 Advantages and Performance Characteristics

Tries provide distinct advantages for operations on string collections.

**Time Complexity**: A primary advantage is that the time required for search and insertion operations depends primarily on the length of the key ($L$), not on the total number ($N$) of keys stored. Specifically, these operations typically require examining a number of nodes proportional to $L$. Sedgewick & Wayne (2011) state that the number of array accesses (node visits) is at most $L + 1$ [7]. This $O(L)$ performance makes tries highly efficient for prefix-based searches (e.g., retrieving all keys starting with a given prefix) and ensures that search performance does not substantially degrade as the dataset size increases.

**Prefix Sharing**: When stored strings share common prefixes, tries inherently exploit this by having those strings share the initial path from the root. This sharing can lead to significant space savings compared to storing each string independently, particularly if there is substantial prefix overlap among keys [5].

### 2.2.4 Space Complexity Considerations

While prefix sharing can save space, the overall space complexity of $R$-way tries requires careful consideration. Each node in an $R$-way trie conceptually provides $R$ potential links. If the alphabet size $R$ is large (e.g., for Unicode) and the actual branching factor at most nodes is low (a sparse trie), allocating space for all $R$ links per node can lead to considerable memory consumption due to numerous unused (null) links [6, 7].

### 2.2.5 Foundational Role in String Processing

The trie data structure is a cornerstone for numerous advanced string processing algorithms. Critically for this work, the Aho-Corasick algorithm, engineered for the efficient, simultaneous matching of multiple patterns within a text, employs a trie—constructed from the set of patterns (the dictionary)—as its initial structural framework. This pattern trie is subsequently augmented with specialized ”failure links” to form a finite automaton, enabling text processing in time linear to the text's length [8]. A thorough grasp of trie principles is thus indispensable for developing and analyzing the Aho-Corasick algorithm.

## 2.3 PATTERN MATCHING

This section introduces the fundamental problem of pattern matching within strings, explores key algorithmic approaches to its solution, and underscores its significance across various computational domains. This foundational discussion provides the essential context for understanding advanced multi-pattern matching algorithms, such as the Aho-Corasick algorithm.

### 2.3.1 The Pattern Matching Problem

Pattern matching involves identifying all instances where a given pattern $P$, which is a string of length $M$, appears as a contiguous substring within a larger text $T$, which has a length of $N$. Beyond simply finding the first match, the problem often extends to locating all occurrences, enumerating them, or extracting surrounding contextual information for each match [7].

### 2.3.2 Importance and Applications

Pattern matching stands as a cornerstone problem in computer science with diverse applications. In general information processing, it underpins the functionality of text editors and web search engines for keyword searches. The field of computational biology extensively utilizes pattern matching for analyzing genetic sequences [7].

Furthermore, in the field of information security, pattern matching is crucial: Network Intrusion Detection Systems (NIDS) utilize it to identify malicious signatures in network traffic. Tools like YARA, for instance, define rules based on textual or binary patterns to classify and identify malware (`https://virustotal.github.io/yara/`). Later in this

work, we'll explore the role of pattern matching for NIDS in greater detail.

### 2.3.3 Pattern Matching Algorithms

Pattern matching algorithms are generally categorized based on the number of patterns they process and the exactness of the match. This research focuses on Exact Matching, where the pattern must correspond precisely to a segment of the text. A key distinction for this study lies between algorithms designed for Single-Pattern Matching and those optimized for Multi-Pattern Matching, where the objective is to identify all occurrences of any pattern from a predetermined set of patterns within a given text [8, 9].

### 2.3.4 Fundamental Algorithms for Exact Single-Pattern Matching

**The Naive Algorithm**

The naive or brute-force algorithm represents the most straightforward method for exact pattern matching. Its operation involves iteratively aligning the pattern $P$ with the text $T$ at every possible starting position $i$ (from 0 to $N - M$). For each alignment, $P$ is compared character by character, from left to right, against the corresponding substring. If all characters match, an occurrence is reported [9].

Following this comparison (whether a match or a mismatch), the algorithm shifts $P$ one position to the right relative to $T$ and proceeds to the next alignment. Despite its simplicity, Knuth, Morris, and Pratt (1977) state that this method can be highly inefficient [10]. Its worst-case time complexity is $O(M \cdot N)$ character comparisons. This occurs, for example, when searching for a pattern like $P = a^n b$ within a text $T = a^{2n} b$, where many partial matches occur.

**Optimized Approaches via Pattern Preprocessing**

To address the limitations of the naive algorithm, more advanced pattern matching algorithms include a preprocessing phase for the pattern $P$. This phase extracts structural information about $P$ that is then utilized during the search, enabling larger shifts of the pattern relative to the text and thus reducing the total number of character comparisons.

**Knuth-Morris-Pratt (KMP) Algorithm**: The KMP algorithm [10] efficiently resolves the pattern matching problem with an optimal worst-case time complexity of $O(N+M)$. Unlike the naive approach, KMP avoids re-checking characters that have already been matched. To achieve this, KMP first preprocesses the pattern $P$ by constructing a special array known as the failure array (or prefix array), typically denoted by $p[1..M]$. This array indicates the length of the longest prefix that is also a suffix for each prefix of $P$. With this information, when a mismatch occurs during the search, the algorithm utilizes the failure array to determine how far the pattern should shift to the right, bypassing unnecessary comparisons and preventing redundant checks.

**Boyer-Moore (BM) Algorithm**: The Boyer-Moore (BM) algorithm [11] is widely recognized for its excellent practical performance, often outperforming linear-time methods

in typical cases. Unlike the KMP algorithm, Boyer-Moore typically compares characters starting from the end of the pattern and moves backward. Its efficiency primarily stems from two rules (or heuristics):

- **Bad Character Rule**: When a mismatch occurs, this rule uses the character from the text that caused the mismatch to determine how far the pattern can safely shift. The algorithm shifts the pattern to align this mismatched text character with its last occurrence in the pattern or moves completely past the mismatched character if it does not appear in the pattern.
- **Good Suffix Rule**: When a mismatch happens after matching one or more characters from the end of the pattern, this rule uses the matched suffix to decide how far to shift the pattern. It searches for another occurrence of the matched suffix within the pattern, ensuring that no previously matched characters are unnecessarily compared again.

These two rules combined enable the Boyer-Moore algorithm to skip larger portions of the text, significantly reducing the total number of comparisons.

### 2.3.5 Limitations of Iterating Single-Pattern Algorithms

When the task involves searching for multiple patterns simultaneously from a set $K = \{P_1, P_2, \ldots, P_k\}$ within a text $T$, simply applying a single-pattern matching algorithm for each $P_i$ independently proves highly inefficient. This ”iterative” approach requires scanning the text multiple times, resulting in a total time complexity that can be approximately $O(k \cdot (N + M_{\text{avg}}))$ or worse, where $M_{\text{avg}}$ is the average pattern length [8]. This highlights the need for more specialized solutions.

### 2.3.6 The Need for Specialized Multi-Pattern Algorithms

To efficiently handle applications that require simultaneous searches for a large number of patterns, specialized algorithms are essential. These methods usually preprocess the entire set of patterns to create a unified data structure, which enables a single, efficient pass over the text to identify all occurrences of any pattern within the set.

The theoretical framework of *finite automata* provides a robust and elegant solution for this problem. By constructing a single *finite automaton* (FA) that can recognize all patterns in a given set, the text can be processed in a single continuous scan. Each character read from the text triggers a *state transition* within the FA. When the FA reaches an *accepting state*—a state explicitly designed to indicate the successful recognition of one or more patterns—the algorithm reports an occurrence. Algorithms like the Aho-Corasick algorithm exemplify this approach; they automate the construction of such a *pattern-matching machine*, typically based on a trie structure augmented

with specialized *failure transitions* [8]. This automaton-based paradigm forms the critical foundation for efficiently solving multi-pattern matching problems.

## 2.4 THE AHO-CORASICK ALGORITHM

This section provides a comprehensive examination of the Aho-Corasick algorithm, a highly efficient string-matching method designed to locate all occurrences of multiple patterns (keywords) within a given text. The algorithm leverages foundational concepts from *automata theory* and string data structures, particularly tries, to offer an optimized solution for the multi-pattern search problem. It achieves *linear time complexity* with respect to the combined length of the text and the number of matches found [8].

### 2.4.1 Context

The task of concurrently identifying all instances of a large collection of patterns within a text poses a significant computational challenge. A naive approach, which involves iteratively applying single-pattern matching algorithms (such as KMP or BM) for each pattern, is inefficient. This method necessitates repeated scans of the text, leading to a worst-case time complexity of approximately $O(k \cdot N)$ where $k$ is the number of patterns and $N$ is the text length, rendering it impractical for large pattern sets or long texts.

The Aho-Corasick algorithm addresses this limitation by constructing a single *finite automaton* from the entire set of patterns. This automaton processes the input text in a single pass to detect all matches. The algorithm effectively combines the *state-transition* efficiency of finite automata with the prefix-sharing advantages of tries. It generalizes the "*failure function*" concept from the Knuth-Morris-Pratt algorithm to the multi-pattern domain, enabling efficient handling of mismatches. While the basic Aho-Corasick machine utilizes *goto transitions* and *failure back-edges* (making it quasi-deterministic), a fully deterministic version can also be pre-computed if necessary, though the standard approach with *failure links* is often preferred for its construction efficiency.

### 2.4.2 Algorithm Components and Construction

The Aho-Corasick algorithm operates in two main phases: a preprocessing (construction) phase that builds the *pattern-matching automaton*, and a matching (search) phase. Three key functions, constructed sequentially define the automaton:

**The Keyword Trie**

The initial stage involves building a *keyword trie*, which is a standard trie constructed from the given set of patterns $K = \{P_1, P_2, \ldots, P_k\}$. Each node in this trie represents a state in the automaton, and each edge corresponds to a character transition. A path from the root to any state $s$ spells out a string that is a prefix of one or more keywords. This structure is formally the *goto function*, $\mathrm{g}(s, a)$, which for a state $s$ and character

$a$, returns the next state $s'$ or indicates failure if no such transition exists. The root state is typically completed with transitions (often self-loops or transitions to a dedicated "fail" state if not leading to a pattern prefix) for all alphabet characters not starting any keyword, ensuring the automaton always advances one text character per cycle [7, 8].

**The Failure Function**

The *failure function*, f($s$), is central to the algorithm's efficiency. For any state $s$ (other than the root), f($s$) points to the state $s'$ such that the string labeling the path from the root to $s'$ is the longest proper suffix of the string labeling the path from the root to $s$ that is also a prefix of some keyword in $K$. This function enables the automaton, upon a mismatch at state $s$ with the current text character, to transition to f($s$) and re-attempt a match without rescanning text characters. The failure function is typically computed level-by-level (breadth-first search order) after the goto function is complete, leveraging already computed failure values for parent states.

**The Output Function**

The *output function*, output($s$), identifies all keywords that end at state $s$. For a given state $s$, output($s$) is the set of all patterns $P_i$ in $K$ such that the string corresponding to the path to $s$ is $P_i$. To ensure all matches are reported, including those that are suffixes of other patterns, output($s$) is augmented during the failure function computation: if f($s$) $= s'$, then output($s$) is set to output($s$) $\cup$ output($s'$). This incremental union ensures that reaching state $s$ correctly identifies all patterns ending at $s$ or at states reachable from $s$ via *failure links*.

**Matching Phase**

Once the Aho-Corasick automaton is constructed, it processes the input text $T$ character by character in a single pass. Starting at the root (initial state), for each text character $a_i$:

1. The automaton attempts to transition from the *current_state* using g(*current_state*, $a_i$).
2. If this transition leads to a state $s'$, the *current_state* becomes $s'$.
3. If g(*current_state*, $a_i$) indicates failure, the *current_state* is updated to f(*current_state*), and step 1 is repeated with the same text character $a_i$ until a valid goto transition is found or the root state is reached (from which a goto transition will always exist for $a_i$, possibly to the root itself if $a_i$ does not start any pattern).
4. After each successful transition to a new state $s'$, the algorithm consults output($s'$) and reports all keywords found, along with their ending position $i$ in the text.

The number of state transitions (goto and failure combined) during the matching phase on a text of length $N$ is bounded by $2N$.

**Time and Space Complexity**

The Aho-Corasick algorithm exhibits optimal *linear time complexity* for multi-pattern matching [8, 9].

- **Time Complexity**
  - Construction Phase (Preprocessing): The construction of the goto, failure, and output functions takes $O(L)$ time, where $L$ is the total length of all keywords in the pattern set. Suppose a dense representation for root transitions is used (e.g., an array indexed by alphabet characters). In that case, an additional $O(|\Sigma|)$ term for alphabet size $|\Sigma|$ may be incurred for initializing these root transitions. Thus, construction is generally $O(L + |\Sigma|)$.
  - Matching Phase (Search): Processing a text of length $N$ involves at most $2N$ state transitions. Reporting matches takes additional time proportional to the total number of occurrences, $O_k$. Therefore, the search phase is $O(N + O_k)$.
- **Space Complexity**
  - The space required to store the automaton (goto function, failure function, and output function) is proportional to $O(L + |\Sigma|)$. The goto function can be stored as a trie, requiring space proportional to $L$. The failure function requires $O(L)$ space (one pointer per state). The output function, in its most general form, could also require more space if many states have many outputs, but efficient representations exist.

### 2.4.3 Applications and Relevance

The Aho-Corasick algorithm is extensively used in applications demanding high-performance multi-pattern matching. Its single-pass processing capability makes it ideal for systems like Network Intrusion Detection Systems (NIDS), which scan network traffic for thousands of known malicious signatures. In cybersecurity, it is a core component of anti-malware tools and systems like YARA, where it efficiently matches literal string components extracted from more complex rules. Other applications include large-scale text analysis, bioinformatics (searching for multiple motifs in DNA or protein sequences), and spam filtering. It should be noted that while Aho-Corasick excels at exact literal string matching, systems like NIDS often layer additional capabilities (e.g., regular expression matching for parts of signatures) on top of or in conjunction with Aho-Corasick engines to handle more complex pattern definitions, which can affect overall performance characteristics. The Aho-Corasick algorithm remains a theoretically important and practically impactful example of applying *finite automata* theory to solve complex string processing problems.


3

# 3 REVISÃO SISTEMÁTICA

This section outlines the methodology employed for the Systematic Literature Review (SLR) conducted to identify, evaluate, and synthesize pertinent studies concerning the parallelization and optimization of the Aho-Corasick algorithm. The review focuses on implementations targeting multi-core CPUs. The objective of this SLR is to support the development of the present research and to answer the formulated research questions.

## 3.1 RESEARCH QUESTIONS

The following Research Questions (RQs) were formulated to guide the literature search, selection, and analysis process:

1. **RQ1:** What parallelization techniques and associated challenges are documented in the literature for implementing the Aho-Corasick algorithm utilizing threads on multi-core CPUs?

2. **RQ2:** What specific optimizations for the Aho-Corasick algorithm are employed in application scenarios involving extensive pattern sets, particularly within domains of information security?

## 3.2 SEARCH STRATEGY AND STUDY SELECTION PROTOCOL

The search and selection protocol for identifying primary studies encompassed the definition of data sources, construction of search queries, and establishment of inclusion and exclusion criteria.

### 3.2.1 Data Sources

As seguintes bases de dados científicas foram selecionadas devido à sua abrangência e relevância na área de Ciência da Computação e Engenharia de Software:

- Web of Science (WoS)
- Scopus
- ACM Digital Library (ACM DL)

- IEEE Xplore

### 3.2.2 Search Queries

Two primary search strings were constructed to address the aspects of parallelization (RQ1) and optimization/application (RQ2) pertinent to the Aho-Corasick algorithm. The literature search was executed in May 2025.

- **String 1 (Focus on Parallelization):** `"Aho-Corasick" AND (parallel* OR multithread* OR multi-thread* OR thread* OR concurren* OR hpc OR high-perform* OR high perform*) AND ("multi-core" OR multicore OR "many-core" OR CPU) NOT "distribut systems")`
- **String 2 (Focus on Optimization and Applications):** `"Aho-Corasick" AND (optimi* OR accelerat* OR perform* OR evaluat* OR improve*) AND ("application" OR "implementation" OR "use case"))`

### 3.2.3 Inclusion (IC) and Exclusion (EC) Criteria

The subsequent criteria were established for the selection of primary studies:
**Inclusion Criteria (IC):**

- **IC1 (Primary Focus):** The study addresses the Aho-Corasick (AC) algorithm or presents parallelization/optimization techniques demonstrably applicable to AC.
- **IC2 (Parallelism Aspect):** The study investigates, proposes, implements, analyzes, or explicitly discusses the parallelization or concurrent execution of the Aho-Corasick algorithm.
- **IC3 (Target Platform):** The study reports empirical results or analyses about multi-core CPU platforms.
- **IC4 (RQ Relevance):** The study provides information relevant to addressing at least one RQ.
- **IC6 (Language):** Published in English or Portuguese.
- **IC7 (Accessibility):** Full-text availability of the study.
- **IC8 (Publication Window):** Published between January 2015 and May 2025, inclusive.

**Exclusion Criteria (EC):**

- **EC1 (Irrelevant Focus):** Studies with only superficial mentions of the Aho-Corasick algorithm or pattern matching, lacking a primary focus on its parallelization or optimization.

- **EC2 (Non-CPU Platform Exclusivity):** Studies exclusively focused on non-CPU platforms (e.g., GPU, FPGA) without analysis or insights transferable to multi-core CPU environments.
- **EC4 (Insufficient Detail):** Studies lacking sufficient detail regarding methodology, implementation, or reported results.
- **EC5 (Publication Type):** Opinion pieces, editorials, and non-peer-reviewed gray literature.
- **EC6 (Language Barrier):** Publications in languages other than English or Portuguese.
- **EC7 (Duplication):** Duplicate publications (the most comprehensive or recent version was retained).
- **EC8 (Inaccessibility):** Full-text unobtainable.

### 3.2.4 Selection Process and Initial Search Outcomes

The study selection was a multi-stage process. Initially, the constructed search strings were executed against the selected digital libraries. The retrieved results were subsequently refined using filters provided by each database (e.g., publication period, document type, language, subject area), as detailed below:

- **ACM Digital Library:** String 1 yielded 122 articles, reduced to 29 following filter application
  String 2 yielded 274 articles, reduced to 66 following filter application
  Filters applied: Publication years: 2015-2025; Document type: Research Article; Publisher: ACM; Title exclusion: GPU; Language: English.
- **Web of Science:** String 1 yielded 30 articles, reduced to 6 after filter application
  String 2 yielded 66 articles, reduced to 20 after filter application.
  Filters applied: Publication years: 2015-2025; WoS Categories: Computer Science (excluding Artificial Intelligence); Title exclusion: GPU; Language: English.
- **Scopus:** String 1 yielded 34 articles, reduced to 6 following filter application
  String 2 yielded 145 articles, reduced to 49 following filter application
  Filters applied: Publication years: 2015-2025 Language: English Subject areas: Computer Science, Engineering; Document types: Article, Conference Paper; Title exclusion: GPU.
- **IEEE Xplore:** String 1 yielded 25 articles, reduced to 2 following filter application
  String 2 yielded 60 articles, reduced to 18 following filter application

Filters applied: Publication years: 2015-2025; Title exclusion: GPU; Language: English.

Following the initial database search and filtering stage, the resultant sets of candidate articles were aggregated.

The filtered outputs from each database and search query were subsequently merged, and duplicate entries were eliminated. Thereafter, a title and abstract screening was performed on the unique articles, and the established inclusion and exclusion criteria were applied. The articles that passed this screening phase (30) proceeded to a full-text review for a conclusive evaluation against the IC/EC. This procedure yielded a selection of Y primary studies designated for comprehensive analysis.

At the end of the selection process, Z studies were considered relevant and included in this systematic review.

## 3.3 ANALYSIS AND SYNTHESIS FROM SELECTED STUDIES

The 'Z' selected primary studies were critically analyzed in order to extract relevant information to answer the research questions.

### 3.3.1 Analysis related to RQ1: Techniques and Challenges in Parallelizing Aho-Corasick on Multicore CPUs

The reviewed literature indicates that the Aho-Corasick algorithm, despite its efficiency in sequential implementations for multiple pattern matching, especially in applications such as Network Intrusion Detection Systems (NIDS)...(to be continue)


4

# 4 METODOLOGIA

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

# 5

# 5 CONCLUSÃO

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

# REFERENCES

## References


[1] P. Pacheco, *An Introduction to Parallel Programming*, 2nd ed. Morgan Kaufmann, 2011.

[2] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley, 1997.

[3] M. J. Flynn, “Very high-speed computing systems,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[4] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley, 2003.

[5] E. Fredkin, “Trie memory,” *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Addison-Wesley, 1998.

[7] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley Professional, 2011.

[8] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Communications of the ACM*, vol. 18, no. 6, pp. 333–343, 1975.

[9] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[10] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt, “Fast pattern matching in strings,” *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[11] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

# APPENDICES

# Appendix A - Título do apêndice A

Os apêndices são materiais suplementares **elaborados pelo próprio autor**, que contribuem para a compreensão, fundamentação ou detalhamento do trabalho, mas que não são essenciais para o desenvolvimento principal do texto.

Eles podem incluir, por exemplo:

- Questionários e roteiros de entrevistas utilizados na pesquisa;
- Documentos, códigos-fonte, tabelas extensas ou informações complementares;
- Detalhes técnicos ou cálculos que, se inseridos no corpo principal, tornariam a leitura cansativa;
- Materiais que apoiam a metodologia ou os resultados, mas que não são obrigatórios para a compreensão geral.

Cada apêndice deve ser identificado por uma letra maiúscula (Apêndice A, Apêndice B, etc.) e ter um título claro e objetivo. Eles são inseridos após as referências bibliográficas e antes do índice remissivo, se houver.

A inclusão dos apêndices deve ser feita de maneira organizada e referenciada no texto principal, para que o leitor saiba quando e por que consultar esses materiais complementares.

# ANNEXES

# Annex A - Título do Anexo A

Os anexos são documentos, materiais ou informações que **não foram elaborados pelo autor do trabalho**, mas que servem para complementar, ilustrar ou fundamentar o conteúdo apresentado.

Exemplos comuns de anexos incluem:

- Legislações, normas técnicas ou regulamentos;
- Manuais, folhetos, publicações oficiais;
- Textos, tabelas ou documentos externos que deram suporte à pesquisa;
- Materiais gráficos ou audiovisuais cedidos por terceiros.

Assim como os apêndices, os anexos são organizados em sequência e identificados por letras maiúsculas (Anexo A, Anexo B, etc.), com títulos claros.

Eles são inseridos após os apêndices (quando houver) ou diretamente após as referências bibliográficas, e devem ser referenciados no corpo do trabalho para que o leitor saiba quando consultá-los.


idp

**Ensino** que
te **conecta**