Undergraduate Thesis

x-search: A C++ Library for Fast External String Search

Leon Freist

Examiner: Prof. Dr. Hannah Bast
Advisers: Johannes Kalmbach

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures

April 20, 2023

Writing Period

15. 02. 2023 – 15. 05. 2023

Examiner

Prof. Dr. Hannah Bast

Advisers

Johannes Kalmbach

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

This thesis presents and describes *x-search*, a C++ library for string searching on external data. *x-search* provides a basic framework for implementing pipeline-based procedures. *x-search* provides a simple, ready to use API for searching patterns within file contents that can easily be included within C++ projects. The main goals of the *x-search* library include

- Performance: optimized multi-thread accelerated I/O and search routines.
- Usability: a simple yet powerful API for the most commonly used search result types (byte offsets, line indices, lines) and simple inclusion to other C++ projects.
- Variability: a robust backbone that enables developers to optimize the process for more specific use cases or to extend it by additional features.

The default components included in *x-search* work on text data encoded in UTF-8 and provide an interface for working on preprocessed data using *x-search*-specific preprocessing such as compression.
We further introduce a *GNU grep*-like executable called *xs grep* which uses *x-search* as its backbone. We will show that *xs grep* performs multiple times faster than *GNU grep* (a standard in exact string searching software on external data) on single files. We further show that *xs grep* also outperforms ripgrep (a grep-like command line search tool with better performances than *GNU grep*) in scenarios, in which the search process dominates reading the data. Therefore, we present and discuss runtime comparisons of *xs grep*, *GNU grep* and *ripgrep* on different hardware and with different search complexities (e.g. case-insensitive or regex searches).

Zusammenfassung

In dieser Arbeit wird *x-search*, eine C++-Bibliothek für die Stringsuche in externen Daten, vorgestellt und beschrieben. *x-search* bietet ein Grundgerüst für die implementierung von pipline basierten Prozessen. Außerdem beinhaltet *x-search* eine einfache API für die Suche von Suchwörtern in Dateien. Zu den Hauptzielen der *x-search*-Library gehören

- Leistung: optimierte I/O- und Suchroutinen
- Nutzbarkeit: eine einfache, aber leistungsfähige API für die am häufigsten verwendeten Suchen (Byte-Offsets, Zeilenindizes, Zeilen) und einfache Einbindung in andere C++-Projekte
- Variabilität: ein robustes Grundgerüst, das von Entwicklern erweitert und angepasst werden kann, um den Prozess für spezifische Anwendungsfälle zu optimieren oder zusätzliche Funktionen hinzuzufügen

Die in *x-search* enthaltenen Algorithmen arbeiten mit UTF-8 kodierten Textdaten und bieten eine Schnittstelle für die Arbeit mit vorverarbeiteten Daten unter Verwendung von *x-search*-spezifischer Vorverarbeitung wie zum Beispiel Kompression der Daten.
Darüber hinaus stellen wir ein *GNU grep*-ähnliches Programm namens *xs grep* vor, das *x-search* als Grundlage verwendet. Wir zeigen, dass *xs grep* bei der suche auf einzelnen Dateien um ein Vielfaches schneller ist als *GNU grep* (ein Standardprogramm für die exakte Suche nach Zeichenketten in externen Daten). Wir zeigen außerdem, dass *xs grep* in Szenarien, in denen die Suche das Lesen der Daten dominiert, auch schneller als ripgrep (ein grep-ähnliches Kommandozeilen-Suchwerkzeug mit besserer Leistung als *GNU grep*) arbeitet. Dafür präsentieren und diskutieren wir Laufzeitvergleiche von *xs grep*, *GNU grep* und *ripgrep* auf unterschiedlicher Hardware und mit unterschiedlichen Suchkomplexitäten (z.B. case-insensitive oder regex suchen).

Contents

List of Figures

List of Tables

1 Introduction

String search is a fundamental problem in computer sciences with applications in diverse fields such as information retrieval, natural language processing and bioinformatics. One can distinguish between two major string-searching approaches. First is the *full-text search*, which searches for matches of strings in individual database documents (Beall, 2008). This type of string search is often used in search engines that return a list of documents that contain patterns of a search query. In order to increase the performance of searching all documents of a database, the data are indexed (Ferragina and Grossi, 1999). The second type is the *exact string matching problem*. The *exact string matching problem* involves finding one or all occurrences of a pattern p of length m in a text d of size n (Charras and Lecroq, 2004).

This work focuses on the second type, the *exact string search* and its appliance in realistic computer hardware systems. We define *realistic computer hardware systems* as systems with limited computing power and primary memory. Thus, we consider string searching on *external* data (e.g. data are stored on secondary memory).

String searching on external data refers to the problem of searching for a specific string or pattern p within data d that are stored on secondary memory, such as a hard drive or a network storage device. Searching external data differs from in-memory string searching, where the entire data can be loaded into primary memory and searched using various algorithms and data structures. We provide a more detailed and formal definition of the problem in section 2.

One of the main challenges of string searching on external data is the limited size of primary memory, which may not be sufficient to store the entirety of the data simultaneously. As a result, we may only look at small chunks of the data at a time. The search results of each chunk can then be combined to produce the final search result.

Several approaches can be used to perform string searching on external data, depending on the specific requirements and constraints of the problem. We will discuss some of these approaches in section 3.

Preliminaries

This section formalizes terms that need a precise definition in the context of this work. When we use any of the terms in the remainder of this work, it is to be understood with the definition provided here.

Strings

An **alphabet** (Σ) is a finite set of characters. We define Σ to be the set of ASCII (American Standard Code for Information Interchange) characters.

A **string** (s) over an alphabet Σ is any finite sequence of characters from Σ:

$$s \in \bigcup_{n \in \mathbb{N}_0} \Sigma^n \iff s \in \Sigma^*$$

Thus, Σ^* denotes the set of all finite length strings over Σ.

Suffix: A string s is a suffix of another string t if and only if there exists a string p such as $ps = t$.

A **line** (l) is a string of size n whose last character is the only newline character ('`\n`', ASCII code F_{hex}) within l:

$$l \in \bigcup_{n \in \mathbb{N}_0} (\Sigma \setminus \{F_{hex}\})^{n-1} + F_{hex}$$

A **pattern** (p) is a string of size m over Σ that must not contain a newline character:

$$p \in \bigcup_{m \in \mathbb{N}_0} (\Sigma \setminus \{F_{hex}\})^m$$

The **size** (or length) of a string s is the number of characters in s denoted by $|s|$.

A **chunk** (c) is a set of i lines (l_0 to l_{i-1}).

Further Terminology

Primary memory refers to all kinds of computing memories directly accessible by the processor via the data bus. Here we use this term primarily to refer to modern computer's RAM (Random Access Memory).

Secondary memory refers to all kinds of data storage not directly accessible by the processor. When using the term *external* (e.g. 'external data'), we assume data to be stored on secondary memory (e.g. SSD, HDD).

A **task** is a unit of work. In our case, a task is a unit of work that can be computationally worked on. Examples are the task of reading or searching data.

I/O-bound describes the scenario where the runtime of a task is limited by reading or writing data to/from secondary memory. Thus, a task is IO-bound, if the execution time of the task is dominated by I/O processes such as reading from external memory. That is, the processor cannot perform as much computation, as physically possible, because it has to wait for new data to be loaded.

CPU-bound describes the scenario where the computing capabilities of the processor limit the runtime of a task.

The **bottleneck** of a procedure (series of either I/O- or CPU-bound tasks) is the task that requires the longest time to complete within the procedure.

A **pipeline** is a series of data processing tasks where one task's output becomes the subsequent task's input. Thus, the tasks rely on each other linearly, while the data can be processed independently.

Motivation

In the past, efficient string searching algorithms such as the well-known Rabin-Karp (Karp and Rabin, 1987), Knuth-Morris-Pratt (Knuth, James H. Morris, and Pratt, 1977) and Boyer-Moore (Boyer and Moore, 1977) algorithms have been developed and formalized. Those algorithms operate in-memory, meaning the pattern p and the string s remain in primary memory. Those algorithms are asymptotically optimal (Baeza-Yates, Choffrut, and Gonnet, 1994). However, naive implementations do not take into account physical limitations such as memory management. Thus, the research claim of this work resides in the challenge of implementing an algorithm that works fast in practice and on external data. Since search tools that operate fast on external data (e.g. GNU grep, which we will introduce later) already exist, the motivation for this work is to

1. Provide faster searches than existing tools.
2. Provide a C++ API for its usage in other C++ projects.

2 Problem Definition

This section will formalize the *External String Searching Problem* (ESSP). It further provides two example procedures that solve the ESSP as defined.

The External String Searching Problem

Let s be a static string. By *static*, we mean that s must not be changed after initialization. Let k be the number of newline characters (ASCII F_{hex}) within s. s represents a set of k lines $\{l_i \ : \ i \in [0, k) \wedge i \in \mathbb{N}_0\}$. The lines within s are indexed from 0 to $(k-1)$. Further, let p be the pattern searched within s.

The goal is to find all indices of lines within s that contain p while only using a limited amount of primary memory.

The requirements for software solving the *External String Searching* problem (ESSP) are summarized below:

1. The input consists of a pattern p and the path of a file containing a string compliant with the requirements stated for s.
2. The result is a list of indices of all lines that contain p.
3. The software only requires a limited amount of primary memory. This kind of assumes that s is *well separated* in the sense that new line characters are not too sparse.

It is important to note that the requirements do not exclude any preprocessing steps. In later sections, we will see how to use preprocessing to increase reading and searching performance in specific scenarios such as working with I/O-bound procedures.

Examples

Example 1 and Example 2 describe basic procedures fulfilling the requirements for solving the ESSP:

Algorithm 1 Example 1

```
1: procedure SEARCH(Pattern p, File f)
2:     Read f line by line while counting the number of read lines (k)
3:     for line l_k in read lines do
4:         search p in l_k
5:         if p was found then
6:             append k to the results list
7:         end if
8:     end for
9: end procedure
```

Algorithm 2 Example 2

```
1: procedure PREPROCESS(File f)
2:     Read f line by line while counting the number of read lines (k)
3:     for line l_k in read lines do
4:         Store the byte offset of the first character of l_k together with k
5:     end for
6:     Store the collected data in file m
7: end procedure
8: procedure SEARCH(Pattern p, File f, Metafile m)
9:     Read f in larger chunks c
10:    Read m into list l_m
11:    for c in read chunks do
12:        while p is found in c do
13:            Map the byte offset returned by the search to a line index using l_m
14:            add the line index to the result list
15:        end while
16:    end for
17: end procedure
```

3 Common Approaches

As mentioned, string search is frequently used in different contexts such as bioinformatics, information retrieval and natural language processing. Therefore, different approaches have been developed over time. This section presents different approaches and discusses their application for solving the ESSP described in the previous section 2.

Full-Text Indexing

Some search engines implement text-searching functionalities using full-text indexing. Full-text indexing involves an *Indexer* creating an index of all the words within the documents of a database along with additional information about the words, like frequency or the positions of the word. This index is used to efficiently search words or phrases within the database's documents. However, indexing a text and then searching for an infix (a substring that appears within a word) will not necessarily result in all occurrences of the infix. Typically, full-text-indexing-based search engines return the documents or sections containing the pattern rather than all occurrences of the pattern within the documents. Further, full-text indexing is typically only suitable for searching complete words (keywords). Since our problem includes searching for substrings of any kinds, full-text indexing is not suitable for solving the ESSP.

Suffix Trees

A *suffix tree* is a compressed trie representing all suffixes of a string. Suffix trees are a commonly used data structure in various string-related algorithms. Relevant to this work is its application for locating substrings within a string s. Ukkonen (1995) described

an efficient suffix tree construction from a string s. Its space and runtime complexity is linear in the size of s.

Substring search using a suffix tree can be implemented with a runtime linear in the size m of a pattern p ($|p| = m$, $\mathcal{O}(m)$) if each node holds an array of possible child nodes that can be looked up in constant time ($\mathcal{O}(1)$). Thus, each node in the suffix tree holds an array of size $|\Sigma|$. This is only practical if $|\Sigma|$ is small (e.g. for genetic sequences, Σ consists of only four elements). However, the strings that we consider in this work are built over the ASCII characters' alphabet; thus, the size of Σ ($|\Sigma|$) is 128. Further, according to McCreight (1976), the upper bound of the number of nodes ($|nodes|$) of a suffix tree of a string s is $|nodes| = 2 \cdot |s|$. Even if the average size of suffix trees might be smaller in practice ($< 1.38 * |s|$, Blumer, Ehrenfeucht, and Haussler, 1989), the overhead of the size of a suffix tree over the size of the original data is relatively large since each node does not only hold the actual character but also pointers to its child nodes and its byte position within the data. Thus, constructing a suffix tree requires external approaches for large data. Farach-Colton, Ferragina, and Muthukrishnan (2000) presented a theoretically optimal algorithm for constructing a suffix tree in external memory. However, no practical implementations are available. The most efficient algorithms, B^2ST and others, only operate on genomic data whose alphabet is restricted to only four characters and whose size is at most a few gigabytes (Barsky et al., 2008).

Plain Text Search

As mentioned in section 1, highly optimized string-searching algorithms are already widely used within many applications. The challenge focused on in this work is to search data stored on external memory. *GNU grep* solves the problem defined in section 2. *GNU grep* searches external data for a pattern and returns matching lines, line indices or byte positions of found matches. Because of its performance and low RAM usage, *GNU grep* is

considered a standard for string searching on external data. However, there are multiple disadvantages of *GNU grep*:

1. No library implementation of GNU grep is available but only an executable.
2. *GNU grep* only uses a single thread when searching on a single file.
3. *GNU grep* does not provide an interface for processing the original data before searching them.
4. *GNU grep*s performance is decreased when searching for line indices of matching lines.
5. *GNU grep* is licensed under the *GNU General Public License*[1], which makes it impossible to address the disadvantages (1 to 4) within *GNU grep* and use the software within other licensed products.

ripgrep already addresses some of the issues claimed for *GNU grep*: *ripgrep* provides a library implementation that can be included in other projects. *ripgrep* and its library are written in Rust but provide C bindings for the library. However, a few limitations remain that we aim to solve in our implementation:

1. Since the library is written in Rust and only C bindings exist for inclusion of *ripgrep* into C++ projects, it is impossible to adjust and extend the search procedure or efficiently embed custom processing tasks using C++.
2. *ripgrep* also only uses a single thread when processing a single file.
3. *ripgrep's* performance is still decreased when searching line indices of matching lines.

Boyer-Moore Algorithm

Since both, GNU grep and ripgrep implement versions of the Boyer-Moore algorithm, this section briefly introduces the algorithm.
The Boyer-Moore algorithm is structured into two steps:

[1] https://www.gnu.org/licenses/gpl-3.0.html

1. Preprocessing: The Boyer-Moore algorithm applies two different heuristics (*bad character* and *good suffix*). For both heuristics a lookup table is created within a preprocessing step. For the *bad character* heuristic, the lookup table contains all possible characters and the distance of the last occurrence of the character within the pattern from the end of the pattern. The lookup table created for the *good suffix* heuristic contains the the smallest shift distance that would align a prefix of the pattern with the largest suffix of the pattern that has been matched so far in the searched text.
2. Searching:
    a) The algorithm starts with aligning the first byte of the pattern with the first byte of the searched text.
    b) Now bytes of the patterns are compared to the bytes of the searched text starting at the last byte of the pattern. If a mismatch is found, the lookup tables are used to determine the possible shift of the pattern towards the end of the searched string. The algorithm chooses the maximum value between the determined shifts from the lookup tables.
    c) If both lookup tables cannot provide a possible shift, the pattern is shifted by its length.
    d) A match is found, if all bytes of the pattern match with the aligned bytes of the searched text.

The worst case runtime of the Boyer-Moore algorithm is $\mathcal{O}(n + m)$ with n being the size of the searched text and m being the size of the pattern. However, because of the applied heuristics, its runtime can be increased up to $\mathcal{O}(\frac{n}{m})$ if the pattern consists of distinct bytes. In this case, the pattern is shifted by its full length once a mismatch was found.

Our Approach

We use plain text search with an optional preprocessing step that collects metadata allowing for a further increase of the overall performance in specific scenarios. We follow a linear approach to process the main tasks necessary for the extern string search, which are listed below:

1. Reading data in chunks from external memory into RAM.
2. Processing the chunks so that they can be trivially searched.
3. Perform the actual search.
4. Managing the results collected for the different chunks.

In the subsequent sections, we will also stick to these four main task domains when explaining the structure of *x-search*.

4 Challenges

This section discusses the challenges within the main task domains that arise when implementing a solution to the ESSP. Solutions and explanations on how we have addressed these challenges are discussed in section 5.5.

Reading external Data

Reading large data from disk (and other secondary memory) is a commonly known bottleneck. While modern RAM modules provide data bandwidths of up to 25 GB per second and lane[1] (standard desktop baseboards operate on two lanes), modern SSDs are limited to about 7.5 GB/s[2].

Computing Line Indices

Computing line indices seems like a simple task: All one has to do is searching newline bytes and increase a counter when a newline byte is found. However, in practice, this can slow down a search process significantly. The already mentioned algorithms for string searching perform well because, in most cases, they need to perform fewer comparison operations than the data size (n). If we couple the search process to the simple line index computation described above, we must perform at least n comparisons to find newline characters and matches. On unprocessed data, it is impossible to perform better.

[1] Example: Kingston DDR5 modules
[2] Example: Samsung PCIe 4.0 NVMe M.2 SSD. Note that the stated bandwidth only holds for sequential read operations, which is the main scenario for us.

Substring Search

In section 3, we have mentioned different string search algorithms. These algorithms (or their derivatives) are used in many substring search implementations such as `std::strstr` or `std::string::find` (GCC and clang implementations). However, the runtime of those algorithms is not necessarily optimal in practice. Also, substring search becomes more complex and time-consuming when considering case-insensitive searches or searching for regular expressions (regex). Therefore, substring search remains a challenge within this work.

Thread Management

`x-search` runs different tasks that linearly depend on each other. However, the data passing the pipeline can be processed independently. Thus, we can use multiple threads (worker threads) for processing the data. Using multiple worker threads requires a thread management system with optimal task scheduling to utilize the available threads as efficiently as possible. Implementing such a thread management system is not trivial in most cases and thus remains a challenge we must address in this work.

5 The *x-search* Library

This section presents the practical result of this work: The *x-search* library provides a solution to the ESSP with features exceeding the requirements of software solving the ESSP.
We will present *x-search*'s features and limitations. After describing the core architecture of *x-search*, we will finally present the implementation details of the library and its components. Here we differentiate between the preprocessing (section 5.4) and the search (section 5.5) procedure.

5.1 Features

x-search provides two different kinds of APIs: First is a high-level API that aims to be easily included in other C++ projects and provide simple one-function calls to perform external substring searches. Second, a more complex API that enables developers to extend and add tasks run by the thread management system of *x-search*. The second API aims to enable developers to adjust the reading, processing and searching tasks specific to their requirements for optimal performance.

This work will not look into the APIs in more detail[1].

In addition to the requirements of software solving the ESSP, *x-search* implements the following features:

- Preprocessing the data to increase reading and search processes:
  - Collecting line index mapping data: line index mapping data are pairs of a line index with its corresponding byte offset. More details are provided in section 5.4.

[1]For API references, we refer to the Wiki pages of this project accessible via GitHub.

  - Compressing the original data for increased reading performance. More details are provided in section 5.4.

- Regex searches: *x-search* allows searching for regular expressions (regex).
- UTF-8 support: While the requirements only include ASCII data, *x-search* operates on UTF-8 encoded data. Please note the limitations regarding UTF-8 data stated in section 5.2.
- Different kinds of search results: The requirements include searching for line indices only, while *x-search* includes search routines for different results (e.g. byte offsets, lines, number of matches, ...).

5.2 Limitations

The following limitations must be considered when using *x-search*:

- Patterns must not include newline characters (included in the requirements). Since data are read in chunks by default, patterns including newline characters lead to false negative search results if a pattern starts at the end of one chunk and continues at the beginning of the next chunk.
- UTF-8 support does not include Unicode normalization. The default searchers operate on Unicode code points, and therefore, searches result in false negatives if the pattern and the data are not in the same Unicode normalization.

However, developers can address these limitations by implementing custom tasks. For example, to address the UTF-8 normalization issue, one could add a task that performs Unicode normalization and schedule it before the search task within the task pipeline or implement a searcher task that is aware of Unicode equivalences.

5.3 Overview and Architecture of the *x-search* Library

The core component of *x-search* is a pipeline called `Executor` that executes the provided tasks using multiple threads (c.f. section 5.3).

A Pipeline as Core Component

The `Executor` is a generalized pipeline wrapper holding tasks and executing them using multiple threads. As such, it can be applied in various scenarios that stick to the following requirements:

1. The overall process is a pipeline-like process (tasks linearly depend on each other, while data can be processed independently).
2. The tasks can be split into the following domains:
    a) **Data-Provider**: a singular task that represents the pipeline's entry point.
    b) **Inplace-Processors**: a set of tasks that process the provided data in place.
    c) **Return-Processor**: a singular task that performs final processing and returns the partial result (partial, because each chunk that passes the pipeline results in a separate result).
3. The partial results can be collected as a final result (this requirement can be bypassed by adding a final result type that effectively does nothing).

The `Executer` is constructed with a number of worker threads, a single data provider task, a list of inplace processors and a single return processor. The data provider task and the processors linearly depend on each other and are executed in the order they are provided to the `Executer`.

Apart from the primary purpose of the `Executor`, the external substring search process, it is also used for the *x-search*-specific preprocessing.

Thread Management

In the previous section, we have seen that *x-search* consists of the `Executor` executing a list in a linear dependency, starting with the data provider task. We can formalize this as follows:

We define $\mathcal{T}$ as the set of all tasks consisting of the reader task ($\mathcal{T}_{reader} = \{T_0\}$) and $e \in \mathbb{N}$ subsequent processing tasks (including Inplace and Return-Processors, $\mathcal{T}_{ps}\{T_i : 0 < i < e + 1\}$). While T_{reader} is independent of other tasks, all $T_i \in \mathcal{T}_{ps}$ depend on their precedent task T_{i-1}.

Since T_{reader} is independent, it can be started anytime, and thus multiple data chunks might be available simultaneously. Therefore, we can use multiple threads for processing the provided chunks to increase the overall runtime.

In section 4, we have stated the challenge of implementing an efficient thread management system. Following, we list the approaches that we have followed during this work:

1. **Fixed Thread-Task-Assignment:** In the first approach, we use fixed assignments of threads to tasks. For example, we assign one thread for working on T_0, one for working on T_1 and two for working on T_2 (considering $e = 2$ and the number of available threads $n_{threads} = 4$). The data are transferred between the tasks using thread-safe queues. The issue with this approach is that it is unclear what tasks require the most CPU capacities. E.g. it might be that T_1 is far more CPU demanding than T_2. Thus, processing T_1 is the bottleneck of the pipeline, and the threads assigned for T_2 are not at capacity because they are waiting for data that are processed within T_1. The system would perform better if the threads operating on T_2 temporarily work on T_1 whenever data for T_2 are missing.
2. **Priority Based Thread-Task-Assigment:** In the second approach, we use a priority-based assignment to address the issue faced in the first approach. We construct a thread pool whose threads receive tasks from a queue q_{task}. A task manager pushes the tasks of highest priority to q_{task}. While the issue claimed in the

first approach is resolved, another issue raises: This approach requires one thread for computing priorities instead of using this thread for operating on the actual tasks. Thus, assuming that $n_{threads} = 4$, only three threads work on the tasks, while one thread is responsible for computing priorities and creating corresponding tasks. Also, computing priorities for tasks is not trivial and must be perfectly balanced to achieve optimal thread usage.

3. **Semi-Fixed Thread-Task-Assignment:** The third approach includes a semi-fixed assignment of threads to tasks. By *semi-fixed*, we mean that by default, threads are assigned to tasks as in the first approach, but if a thread has no data to operate on, it starts working on the precedent task. However, this approach has another difficulty: Assuming the bottleneck of the whole procedure is T_{reader} (e.g. because of the disk I/O), no more threads can increase reading throughput, and all threads of subsequent tasks do not have data to operate on. Thus, the threads assigned to subsequent tasks try operating on their precedent tasks but fail due to missing data, which results in a busy-wait[2] state of these threads.
4. **Rotating Threads:** In the fourth approach, we design the procedure so that each thread sequentially runs all tasks without passing data between threads. This means each thread first reads data, then runs all processing and searching tasks on the read data and finally starts reading again, continuing until no data are left to read.
5. **Dedicated Read - Rotating Threads:** The last approach is a mix of the previous ones. It aims to consider the actual conditions for reading data from secondary memory that must be taken into account: Reading data using multiple threads might result in an overhead of the context switch of different threads accessing the same file stream. Therefore, having a dedicated reader thread that is only responsible for reading data can increase reading performance. The read data are provided to the other worker threads (that perform the processing and searching) via a thread-safe queue. However, reading data using multiple threads concurrently for fast secondary memory with fast random access is still possible by falling back

[2]Busy-wait: The threads repeatedly check if any task has data to operate on. Thus, the threads require CPU resources without operating on any task.

to the *Rotating Threads* approach.

Additional implementations could eliminate most of the issues that we claimed for the first three approaches. However, this would have made the code more complex, less readable and less maintainable.

Further, we give a proof that the *Rotating Threads* approach is an optimal solution for the given problem of thread task assignments.

Proof of Optimal Thread Usage in Approach 4 (Rotating Threads)

We define *optimal thread usage* as follows: The thread usage is optimal if all threads work at maximal capacity or data are missing due to other hardware restrictions (e.g. waiting for I/O operations to complete) limiting the threads. Thus no task can be processed using the remaining CPU capacities.

The following statements directly follow from the definition of a pipeline given in section 1:

S0 T_0 can be processed independent of other tasks

S1 $\forall\ T_i \in \mathcal{T}_{ps} : T_{i-1} \mapsto T_i$ (T_i depends on T_{i-1})

We further claim the following assumptions that can be considered to be trivial:

A0 A thread that is working on a CPU-bound task is at maximal capacity

A1 Processing and searching text data that are directly accessible by the CPU is CPU-bound and thus $\forall\ T_i \in \mathcal{T}_{ps}$: T_i is CPU-bound

A2 The bottleneck of the reader task (T_0) depends on the secondary memory and CPU performance and thus T_0 can be

   a) CPU-bound (if the secondary memory provides data faster than the CPU can process them)

   b) I/O-bound

From here, we can conclude conclusion C0:

C0 From assumptions A0 and A1 and the definition of optimal thread usage, we conclude that whenever a thread is working on $T_i \in \mathcal{T}_{ps}$, its usage is optimal.

It remains to show that all threads not currently working on any task within $\mathcal{T}_{ps}$ are also used optimally.

Using statement S0, we argue that all threads not working on tasks from $\mathcal{T}_{ps}$ start working on T_0 (T_0 can be processed independently). We consider the two cases for the bottleneck of T_0 stated in assumption A2.

C1 If T_0 is CPU-bound, we use assumption A0 to argue, that the thread usage is optimal. Together with conclusion C0, we argue, that in this case, the overall thread usage is optimal.

C2 We consider T_0 to be I/O bound. Since every thread sequentially runs all tasks from $\mathcal{T}$, and all tasks from $\mathcal{T}_{ps}$ depend on their precedent task (statement S1), no data are available for processing. Thus, we can conclude, that the thread usage is optimal in this case too, since the threads cannot work on another task with higher capacity. Together with conclusion C0, we conclude that the overall thread usage is optimal in this case.

Since conclusion C1 and conclusion C2 cover all possible scenarios, the thread management described above is optimal.

5.4 Preprocessing Procedure

The preprocessing supported by *x-search* includes compression and collecting metadata that will then be used to increase the reading and searching performance within the search procedure. The preprocessing is constructed using the pipeline wrapper `Executor`

described in the previous section (c.f. 5.3). The pipeline consists of the following tasks executed in the given order:

1. Reading data
2. Collecting Line index mapping data (optional)
3. Compression (optional)
4. Writing compressed data and metadata to files

A schematic overview of the preprocessing procedure is provided in figure 1.

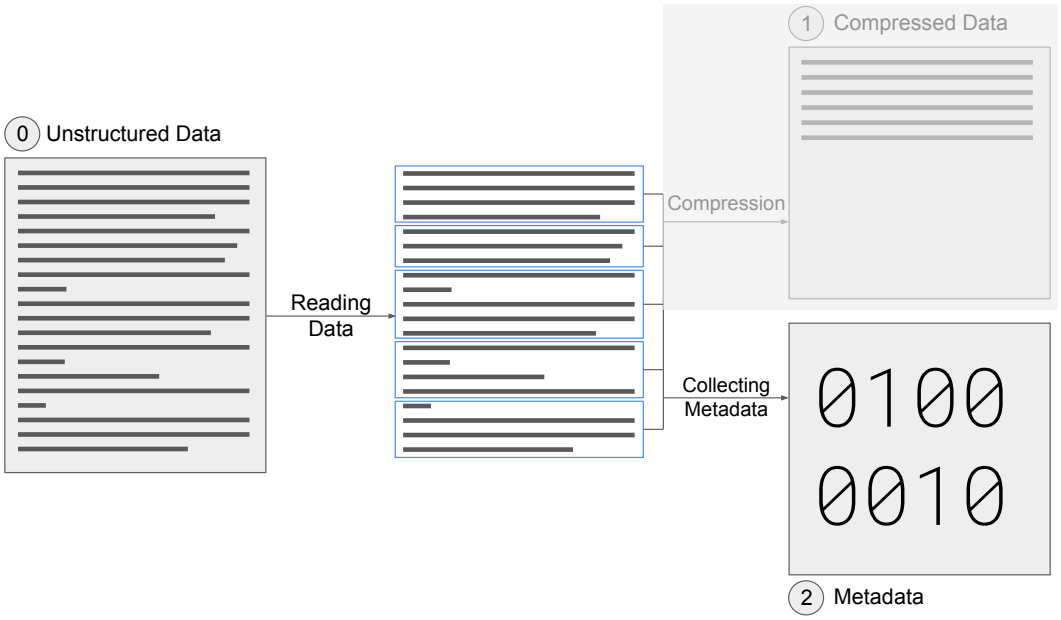

Figure 1: Preprocessing Procedure Scheme: The original unstructured text data (⓪) are read in chunks of similar sizes, all ending with a newline character. Metadata written to a separate metafile is collected for every chunk (②). Each chunk is compressed if the compression option is set, and the compressed data are written to a separate file (①).

Implementation Details

This section focuses on the implementation details of the tasks involved in the preprocessing procedure.

Reading Data

The original data are read in chunks of fixed sizes (by default 16 MiB) plus additional bytes until a newline character or EOF (end-of-file character) is read. Therefore, each chunk begins right after a newline character and ends with a newline character. Reading to the end of a line is necessary because it ensures that the chunks can be processed and searched independently of each other (in section 1, we have defined the pattern as a string that must not contain a newline character). The position of the first byte (absolute to the beginning of the original data) and the size (number of bytes) of each chunk are stored as metadata for later usage (c.f. section 5.4).

Collecting Line Index Mapping Data

Line index mapping data consist of an absolute byte position of a character within a chunk and the absolute line index of the corresponding line containing this byte. The task consists of searching newline characters and storing information about their position within the data. In more detail, the tasks sequentially search for newline characters, and once a given number of bytes are searched, the current byte offset and the line index of the line containing the current byte are stored as a pair. The *line index mapping data* are stored in the metafile, too (c.f. section 5.4). We will describe how these data can be used for searching line indices faster in section 5.5.

Compressing Data (optional)

x-search supports compression as preprocessing using either the *LZ4*[3] or the *ZStandard*[4] algorithms. In addition to the metadata collected while reading the original data, the absolute byte position of the first byte of the *compressed* chunk and its size are collected

[3] LZ4 is a compression algorithm that focuses on fast decompression times.
[4] ZStandard is a compression algorithm developed by Facebook that focuses on a good ratio between compressed size and compression/decompression time.

as metadata, too (c.f. section 5.4). Additionally, a new file is written containing the compressed chunks.

Runtime Complexity

The preprocessing procedure is a series of tasks with runtime complexities linear in the data size (n, $\mathcal{O}(n)$). This trivially holds for reading the data and searching line indices, while compression depends on the used algorithms. The two algorithms supported by *x-search* (*ZStandard* and *LZ4*) also have linear runtime with respect to the data size (n, $\mathcal{O}(n)$). Since we only use the libraries provided for the compression algorithms respectively, we will not look at their runtime in more detail. The overall runtime of the preprocessing procedure depends on the number of tasks (n_{tasks}) the chunks are passed through, and the data size (n) in the form of the number of chunks (n_{chunks}) the data are split into and the size of a single chunk (s_{chunk}). Since the size of the chunks and the number of tasks is constant at runtime, the overall runtime complexity is $\mathcal{O}(n_{tasks} \cdot n_{chunks} \cdot s_{chunk}) \in \mathcal{O}(n_{chunks}) = \mathcal{O}(n)$

Space Complexity

The memory used during preprocessing is dependent on the number of threads ($n_{threads}$) used, and the size of the chunks (s_{chunk}) read: Each assigned thread reads data and then operates on them. Thus, memory usage (m) can be computed by $m = n_{threads} \cdot s_{chunk}$. Since the number of threads is constant at runtime, and the size of a chunk does not depend on the size of the data (n), the overall space complexity is constant: $\mathcal{O}(n_{threads} \cdot s_{chunk}) \in \mathcal{O}(s_{chunk}) = \mathcal{O}(1)$

Metafile

As mentioned, metadata collected during preprocessing are stored in a separate metafile in binary format. Figure 2 provides an overview of the structure of the binary format. The metafile consists of an integer indicating the compression type of the data provided (⓪), followed by a series of metadata collected for each chunk (① and ②). Thus, the size of the metafile ($s_{metafile}$) depends on the chunk sizes (s_{chunk}) and the number of bytes between collecting the next *line index mapping data* ($md\_distance$) for each chunk. $s_{metafile}$ can be calculated using equation 1:

$$s_{metafile} = (4 + \sum_{chunk \in C} 40 + \lceil \frac{s_{chunk}}{md\_distance} \rceil \cdot 16)\ bytes \tag{1}$$

where C indicates the set of chunks that the data are composed of.

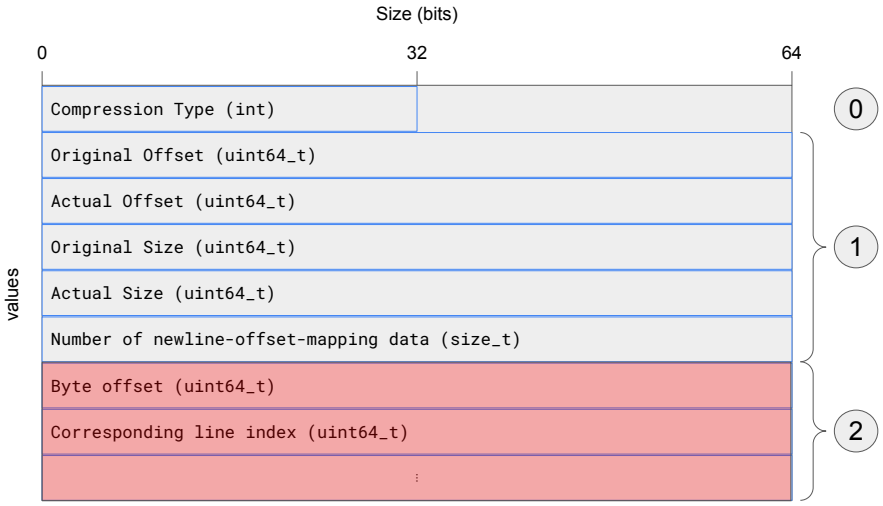

Figure 2: Metafile structure: The metadata are stored in a separate metafile in binary format. The metafile begins with an integer indicating the compression type (unknown: 0, None: 1, LZ4: 2, ZSTD: 3) of the data (⓪). Then, for every chunk, size and position information and the number of collected line index mapping data are stored (①). The line index mapping data are stored for each chunk separately (②).

5.5 Search Procedure

As the preprocessing procedure, the searching procedure utilizes the `Executor` described earlier and thus consists of different tasks. This section will describe the tasks used in the search procedure in more detail. We will orient ourselves at the four stated main task domains, and we will include how we addressed the challenges stated in section 4 corresponding to the task domains:

1. Reading data from external memory into RAM
2. Processing data so that they can be trivially searched
3. Perform the actual search
4. Managing results

Implementation Details

This section provides implementation details and runtime information about the tasks involved in the search procedure.

Reading Data

Unpreprocessed data are read using the same reader used for reading data within the preprocessing procedure. For preprocessed data, we have seen that metadata are collected that can be used to increase reading performance. More specifically, if preprocessed data are read, the start position and the size of each chunk are already known, and the reader does not need to read until it reaches the first newline character. However, reading data from secondary to primary memory is a known bottleneck. In section 4, we have pointed out the performance difference between disks and RAM storage. *x-search* implements different concepts to approach this challenge:

Reading Data in Chunks

Reading data from disk performs best when reading large chunks of bytes instead of single bytes. The reader tasks provided within *x-search* read the data in chunks of predefined sizes.

Reading Data using Memory Mapping

Memory mapping can further improve I/O by directly mapping the data from the disk into the program's virtual memory space.

Data Compression

By compressing the data, we can trade computing time (data must be decompressed before searching them later) for reading time (the data are smaller). It is important to note that this trade-off is not always beneficial. In most cases, we must run benchmarks on the CPU and disk to tell if compression will increase the overall runtime.

Further Approaches (not implemented)

While we explicitly consider reading external data, it is possible to implement reader tasks that add additional features like

1. Buffering data that are most frequently used to provide these data faster to subsequent tasks
2. Providing data that are owned by another process

Processing Data

Data that are read by reader tasks are provided for processing tasks. The purpose of the processing tasks is to process the data so that the subsequent searcher task (c.f. section 5.5) operates on normalized text data that can be searched without further limitations.

Working with Compressed Data

In section 5.5, we mention compression as a possible solution to increase reading performance. In this case, decompression is a necessary processing task (c.f. section 5.4 for more details).

Further Approaches (not implemented)

One possible additional processing task that is not included within *x-search* is the internationalization of read data (e.g. support of different encodings or Unicode normalization)

Searching Data

By default, *x-search* supports the following searches for literal and regex patterns:

- Count the number of matches of the search pattern within the data.
- Count the number of lines that contain the given search pattern.
- Search byte offsets of all matches of the pattern within the data.
- Search byte offsets of the start of all lines containing the search pattern.
- Search line indices of all lines containing the search pattern.

While the first four search operations can be implemented using standard search algorithms (for a literal substring or regex search, respectively), the last one (searching line indices) is more complicated to implement in a performant way. We have pointed out the difficulty of

searching line indices in section 4. Our solution to solve this difficulty is using additional *line index mapping data*

Line Index Mapping

We source each byte's necessary newline character comparison out into a processing step done within the preprocessing or right before the actual search is performed (if no metadata are available) as described in section 5.4. We can now assume that the *line index mapping data* are available for the actual search. In order to get the line index of a match, a search is run that returns the byte offset of a match, and then, the *line index mapping data* are used to map this byte offset to a line index: Therefore, we perform a binary search on the collected *line index mapping data* to get the line index of the byte closest to the match's byte offset. We only need to count newline characters starting at this byte until we reach the byte offset returned by the search to get the line index.

The runtime of mapping a byte offsets to a line index consists of two factors: First is the binary search, and the second is the remaining data size that needs to be scanned for newline characters. The binary search is implemented using `std::upper_bound` included in the C++ standard algorithm library. The specifications claim a runtime complexity that is logarithmic in the size of the mapping data (n_{md}, $\mathcal{O}(log(n_{md}))$, ISO/IEC, 2023). Scanning the remaining data for newline characters is performed by using a trivial character search function which has a complexity linear in the size of the remaining data (n_r, $\mathcal{O}(n_r)$), resulting in a complexity of $\mathcal{O}(log(n_{md}) + n_r)$. Since n_r is a constant set while preprocessing, the runtime of the mapping results in $\mathcal{O}(log(n_{md}))$.

Substring Search

As mentioned, *x-search* supports literal and regex pattern searches. The regex search is supported using the Google/RE2 regex library[5]. The Google/RE2 library uses Deter-

[5] Google/RE2 is a regex engine written in C++, developed and maintained by Google™

ministic Finite State Automata (DFA) to perform the regex search. The runtime of this approach is linear in the size of the input data (Cox, 2007). The literal substring search is done using a SIMD-based substring search algorithm developed in the scope of this work.

SIMD Search Algorithm

To increase the performance of the actual substring search, we have implemented versions of `std::strstr` and `std::strchr` (originally implemented within the cstring library of the C++ standard, ISO/IEC, 2023) using SIMD (Single Instruction, Multiple Data) instructions. More specifically, we have implemented `std::strstr` and `std::strchr` using AVX2[6] instructions. However, the algorithms described are also suitable for other SIMD instruction sets like *SSE* (Streaming SIMD Extensions) or *Neon-ARM*.

Limitations

For a C-style null-terminating byte string (on which we operate), we likely produce segmentation faults if we load data longer than the string provided (in this case, we may try to access data from a restricted memory area). The benefit of SIMD instructions is that they load multiple data at once. We most likely produce a segmentation fault if we do not check the data for the null-terminating byte before loading them into a SIMD register. However, checking each byte for being a null byte would negate the benefit of SIMD instructions.

We address this observation by changing the signatures of `std::strstr` and `std::strchr` by adding the haystacks and patterns size:

```
std::strstr(char* haystack, char* needle)
std::strchr(char* str, int ch)
```

[6] *AVX2* is an expansion of *AVX* (Advanced Vector Extensions), an expansion of the instruction set of the x86 architecture.

become

```
simd::strstr(char* haystack, size_t haystack_size,
             char* needle, size_t needle_size)
simd::strchr(char* str, size_t str_size, int ch)
```

Algorithms

`simd::strstr`: We use predicate equality of the first ($first = p_0$) and the last ($last = p_{n-1}$) byte of the pattern p of size m. These two bytes are loaded into two SIMD vector registers (R_{first} and R_{last}), respectively. We now iteratively load two chunks c (size n_c, which depends on the vectors size supported by the SIMD instruction used) of the data (haystack) d of size n into two additional SIMD vector registers A and B while counting the offset i. i is increased by n_c at the end of each iteration. A is read starting from i and B is read starting from $i + n_p - 1$. Thus, A and B are shifted by the pattern size m. We now compute the following vector expression:

$$(R_{first} == A) \wedge (R_{last} == B)$$

The result is a byte vector (or mask) R_{match}, where `true` values indicate that $first$ and $last$ matches within A and B with a distance of m. For each `true` value within this mask, we perform a string comparison of the pattern and the data beginning at the offsets indicated by the `true` values from R_{match}. If this comparison evaluates positively, we return the pointer $first$. To avoid segmentation faults, the iteration ends if $i + n_c > n$ evaluates to true, and the remaining data are searched using `std::strstr`.

`simd::strchr`: In the first step, we load the searched character p into a SIMD vector register R_p. We now iteratively load chunks c (size n_c, which depends on the vectors size supported by the SIMD instruction used) of the data (str) d (size n) into a SIMD vector register A while counting the offset i. A is read from offset i which is increased by n_c in the end of each iteration. Now, we compute the vector expression $R_p == A$ and

check for `true` values in the resulting mask. We return the pointer to this character if we find a `true` value. Else, we skip to the next iteration. To avoid segmentation faults, the iteration ends if $i + n_c > n$ evaluates to true, and the remaining data are searched using `std::strchr`.

Runtime Analysis

`simd::strstr`: The computing complexity of `simd::strstr` is $\mathcal{O}(n)$ where n is the size of the data d since all bytes are loaded into vector registers and compared to the first and last byte of the pattern.

The space complexity of `simd::strstr` is $\mathcal{O}(n+m)$ where n is the size of the data d and m is the size of the pattern p. In our use case, this term is dominated by n resulting in a space complexity of $\mathcal{O}(n)$.

`simd::strchr`: The computing complexity of `simd::strchr` also is $\mathcal{O}(n)$ where n is the size of the data d since all bytes are loaded into vector registers and compared to the searched character.

The space complexity of `simd::strchr` is $\mathcal{O}(n)$ where n is the size of the data d. This is because the pattern p is a single byte with a constant size of $m = 1$ and thus $\mathcal{O}(n+m) \in \mathcal{O}(n)$.

Runtime Complexity

The overall runtime complexity of the search procedure depends on the tasks involved in the search procedure. However, since the tasks linearly depend on each other, the overall runtime is the sum of the runtime of the single tasks. In the previous sections, we have shown that the tasks involved in the default search procedure (reading, decompression, searching, line index mapping) are either linear in the size of the input data (n, reading,

decompression, searching) or logarithmic in the size of the *line index mapping data* (n_{md}, which is linearly dependent on n). Thus, the overall complexity is $\mathcal{O}(n+n+n+log(n_{md})) = \mathcal{O}(3n + log(n_{md})) \in \mathcal{O}(n)$.

Space Complexity

The space complexity of the search procedure is constant. However, the arguments for this depend on the used thread management:

1. **Rotating Threads:** As for the preprocessing, the used memory depends on the number of threads that concurrently run the tasks of the search procedure. Each thread works on one chunk of size s_{chunk} at a time. s_{chunk} does not depend on the size of the input data n but is a constant at runtime. Since the number of threads ($n_{threads}$) is constant at runtime, too, the overall space complexity is constant: $\mathcal{O}(n_{threads} \cdot s_{chunk}) \in \mathcal{O}(s_{chunk}) = \mathcal{O}(1)$.
2. **Dedicated Read - Rotating Threads:** The used memory depends on the size of the queue (q_s) on which the reader task pushes the read chunks. Since the size of the queue is constant at runtime, the overall space complexity results in $\mathcal{O}(q_s \cdot s_{chunk}) \in \mathcal{O}(s_{chunk}) = \mathcal{O}(1)$.

Providing Search Results

The last step of the search procedure is providing the collected results. By default, *x-search* supports the search results listed in section 5.5.

A result consists of a collection of partial results. A partial result is the search result of one single chunk.

By default, *x-search* provides two base result types. First is a count result type (for match counts), and the second is a container result type for collections of results (e.g. byte offsets or line indices).

x-search also provides iterators for the default result types that enable access and iterate over the already collected partial results, while data chunks are still being processed.

6 xs grep: a Command-Line Text Search Utility

This section introduces *xs grep*, a *GNU grep*-like executable built using *x-search*. Extended information are provided in the scope of the Bachelorproject of the author of this work in form of a blog post available at *ad-blog.uni-freiburg.de*.

6.1 Limitations

The following limitations must be considered when using *xs grep*.

- **Single File Input:** At the state of this work, *xs grep* supports searching on single file inputs only instead of on directories or multiple files.
- **Linux Only:** *xs grep* is not yet ported to MaxOS and Windows and is only tested on Linux (Ubuntu 22.04).
- **Limited Command-Line Options:** *xs grep* does not provide all command-line options provided by alternatives like *GNU grep* and *ripgrep* (e.g. Context control arguments).

6.2 Advantages

The main advantage of *xs grep* over *GNU grep*, and *ripgrep* is the ability to use multiple threads for a single input file. While *GNU grep* and *ripgrep* can be used for multi-threaded search on multiple input files, *xs grep* can utilize multiple threads for searching on a single input file. In section 7, we will see that this feature increases the overall runtime for systems with fast secondary memory access. Further, *xs grep* provides additional

command-line options for internal configuration (c.f. section 6.3). Therefore, users can configure *xs grep* specifically for their system specifications and use cases.

6.3 Usage

*xs grep*s basic usage is similar to that of *GNU grep* or *ripgrep*: The program takes a pattern, an input file and a set of (optional) command-line arguments and then searches the content of the input file for the given pattern (literal or regex).

Command-Line Options

The following command-line options are equivalent in use and function to those of *GNU grep* and *ripgrep*:

Option		Description
`--count`	`(-c)`	Count and Output the number of matching lines
`--byte-offset`	`(-b)`	Output matching lines with their absolute byte offset within the file
`--line-number`	`(-n)`	Output matching lines with their line number
`--only-matching`	`(-o)`	Only print nonempty parts of matching lines
`--ignore-case`	`(-i)`	Search case-insensitive
`--fixed-string`	`(-F)`	Consider pattern as fixed string (literal search)

Additionally, *xs grep* provides the following command-line options for internal configuration rather than for specifying search details:

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td><code>--metafile [PATH]</code> <code>(-m)</code></td><td>Read the additional metafile (c.f. section 5.4)</td></tr>
<tr><td><code>--threads [INT]</code> <code>(-j)</code></td><td>Use the given number of worker threads:
<table><tr><th>Value</th><th>Actual number of used threads</th></tr><tr><td><code>value < 0</code></td><td>number of systems physical cores</td></tr><tr><td><code>value == 0</code></td><td>$\frac{1}{2}$ of systems physical cores</td></tr><tr><td><code>value</code></td><td><code>value</code></td></tr></table></td></tr>
<tr><td><code>--max-readers</code></td><td>Maximal number of threads concurrently reading from file</td></tr>
<tr><td><code>--no-mmap</code></td><td>Suppress using memory mapping</td></tr>
</table>

7 Evaluation

This section first describes the hardware and software used for the experiments. It then provides information about the input data and patterns used for the benchmarks. Last, this section presents and discusses the results of the benchmark experiments.

Benchmarking the *x-search* library is difficult. Since *x-search* aims to provide a backbone for pipeline-driven systems where single components can be individually set, benchmarking the default components is not expedient. Hence, we want to benchmark the whole system rather than single components. Therefor, we perform benchmarks using *xs grep*, a *GNU grep*-like executable that we have introduced in the previous section (c.f. section 6). The components used within *xs grep* can be considered neither trivial nor highly optimized. Therefore, *xs grep* becomes a solid use case of *x-search* for benchmarks.

In the first evaluation step, we evaluated different internal configurations of *xs grep* on different hardware (c.f. section 7.2.1). The results of this evaluation were used to define the default configuration of *xs grep* that has been used for comparison with other command-line search utilities.

We consider the following command-line search tools for comparison:

- **ripgrep (baseline)**: In section 3, we have introduced *ripgrep* as an alternative to *GNU grep*. It aims to provide faster text search than *GNU grep* and provides a Rust library with C bindings. We use *ripgrep* as the baseline for the following benchmarks because it can be considered a modern, high-performance text search utility which is widely used.
- **GNU grep**: We have introduced *GNU grep* in section 3 as a standard external string search tool used on many systems. Therefore, when performing search tool comparison benchmarks, *GNU grep* should be considered.
- **xs grep**: *xs grep* is our implementation of a grep-like command line search tool. Its implementation uses *x-search* as its backbone and implements custom search

and result types. *xs grep* was introduced in section 6. It was written in the scope of this work and is used to benchmark the performance of *x-search*.

7.1 Experimental Setup

Hardware

In the previous sections, we point out that the external string search performance depends on two main factors: First is the I/O bandwidth of the external memory, and the second is the algorithms used for the searching procedure and, thus, the CPU performance. Therefore, we run the benchmark experiments on two different computers (HW0 and HW1) that only differ in secondary memory:

- CPU: AMD Ryzen 7 3700X (8 Cores + SMT)
- RAM: 128 GB, DDR-4
- Secondary Memory:

SSD: 2×2 TB SSD (NVMe, RAID 0)
HDD: 3×32 TB HDD (RAID 5)

Benchmark Input Characteristics

This section describes the characteristics of the data and the patterns that are used within the benchmarks. Last, we claim an introductory remark about the fairness and extensiveness of the benchmarks.

Text Data

We run all benchmarks on the English, Spanish and Greek samples of the *OpenSubtitles2016* dataset (a collection of subtitles of movies from opensubtitles.org) published by Pierre

Lison, 2016. We have decided to run benchmarks on these three data sets because they differ in size and character characteristics: While the English version primarily consists of single byte characters, the Greek version consists of mainly multi-byte characters. The Spanish version represents a mix of single- and multi-byte characters. Since the core statements made within the discussion apply to all three input files, we only present and discuss the results of the benchmarks run on the English version. Below, some statistics about the English file are listed:

	Metric	Value
	Number of Lines	337,845,355
bytes per line	Mean	29.5
	Min	1
	25 % Quartile	14
	50 % Quartile	24
	75 % Quartile	38
	Max	26,586

The actual performance of substring matching depends on the searched data and the pattern. We consider the sample data a representative data file for text search because of its natural language content (subtitles of movies and TV). Therefore, we focus on comparing benchmark results on different search pattern complexities rather than on different input files. However, the benchmark results are also reproducible for files of other sizes and characteristics.

Search Patterns

We run searches on the previously described data using different patterns described below.

Pattern	Description	Matching lines	% of lines
`'Sherlock'`	substring search pattern	13645	0.0040
`'She[r ]lock'`	simple regex pattern matching `'Sherlock'` and `'She lock'`	14211	0.0042
`' [sS][A-Za-z]*[kK] '`	complex regex pattern matching words starting with upper or lower case `'s'`, followed by any alphabetic character and ending with lower or upper case `'k'`	1079731	0.3196

The patterns differ in the number of matches and search complexity. We consider `'Sherlock'` as a pattern of low search complexity since the search tools will run a standard substring search on the data. `'She[r ]lock'` is a regex pattern of medium complexity. The last pattern, `' [sS][A-Za-z][kK] '`, is of high search complexity since it does not contain any literal characters and it matches a relatively large number of lines (about 0.3 %).

Remark

It is important to remark that a comprehensive benchmark must consider more patterns that differ in size, number of matches and search complexities.

Further, it is essential to remember that the author of this work also designed the benchmark procedure. Even if the benchmarks were designed with the intention of creating a fair benchmark, the benchmarks might be biased towards *xs grep*.

To put this remark into perspective, it must be said that the components for reading and searching data used by *xs grep* have not been optimized with respect to the input data and

patterns used within this benchmark. The searching and reading algorithms are advanced but general and thus perform similarly on different patterns and data. Further, this benchmark aims to show that the *x-search* library provides a backbone for pipeline-based procedures that can be used to create competitive software.

Benchmark Tools

To automate and simplify benchmark runs, we have written a python program called *cmdbench* that reads a JSON file containing information about a benchmark and sets up and runs benchmarks based on the provided information. *cmdbench* uses an external benchmarking tool for collecting metrics. By default, *cmdbench* can use either *GNU Time* or *InlineBench* as an external benchmark tool.

1. **GNU Time:** *GNU time* is a command line tool that allows collecting different metrics like CPU time (CT) and Wall time (Real Time, RT) of other programs. We use *GNU Time* whenever we only need a program's total CPU or Wall time (e.g. comparison benchmarks).
2. **InlineBench:** *InlineBench* is a C++ benchmarking tool the author has written in the scope of this work. Users can start and stop timers dynamically within C++ code. *InlineBench* supports CPU and Wall timers that collect thread-specific timings (e.g. if multiple threads pass an *InlineBench* timer, times are collected for each thread separately). The *InlineBench* timers must be activated at compilation using the `-BENCHMARK` flag. We use *InlineBench* to measure runtimes of specific components within *x-search*, and *xs grep* (e.g. only the reading task).

Evaluation Metrics

This section describes the three evaluation metrics used for the benchmarks. Since we are running an exact substring search, there is no metric for the quality of the results.

Wall Time

Wall time is the intuitive time that we would measure using a classic stopwatch measuring from a time point *start* to a time point *stop*.

CPU Time

CPU time refers to the total time that CPU cores have been involved in the computations. In more detail, CPU time is computed using the number of CPU cycles per core and the CPU frequency. Therefore, CPU time from *start* to *stop* can be greater than the corresponding Wall time if multiple CPU cores are concurrently computing for the same program. Thus, CPU Time can also be considered as a metric for power efficiency: Lower CPU time corresponds to less power usage. Since the compared search tools solve the same problem, less CPU Time corresponds to better power efficiency.

Relative Wall Time (Wall %)

The relative wall time is used for comparing the results of different benchmark setups. A baseline run is set to a base value of 100 %, and the Wall % of compared runs is computed by $Wall_{baseline}/Wall_{compared\_run} \cdot 100$. Thus Wall % refers to the performance of a run (Wall time only) relative to that of a baseline (e.g. A Wall % of 200 indicates a speedup of factor 2 compared to the baseline).

7.2 Results

This section presents and discusses a selection of the benchmarks run[1]. We divide the benchmarks into two different domains. Table 1 lists a selection of the benchmarks for different *xs grep* configurations (section 7.2.1). Table 2, 3 and 4 list a selection of the

[1]The entirety of the raw data were submitted together with the source code

comparison benchmarks of the command-line search tools *GNU grep*, *ripgrep* and *xs grep* (section 7.2.2).

7.2.1 `xs grep` Configuration

Secondary Memory Performance vs Processing Complexities: Using Multiple Worker Threads

The benefit of using multiple worker threads depends on the limiting factor within the procedure. Two limiting factors can be observed within our procedure that can dominate the runtime: First, the performance of the secondary memory (I/O time, depending on the secondary memory type: RAM cache, SSD, HDD). Second, the computing complexity of processing the data (processing complexity, depending on the search complexity: literal, regex, case-insensitive, ...). If the processing complexity dominates the I/O time (table 1, Cached Reads, and SSD Reads (C0)), the wall time increases with additional worker threads. For the less complex literal search, the dominating factor swaps between using four and eight worker threads. From here on, additional worker threads cannot increase performance because no data are available to operate on. For the complex regex search, we can see that the wall time increases even when using eight worker threads. Here, we can observe speedups of over 700 %. When reading data from HDD, I/O is the limiting factor. Thus, the overall process is I/O bound and using additional worker threads does not increase performance.

SMT Cannot Increase Performance

We have mentioned that the CPU used for the benchmarks use Simultaneous Multithreading (SMT). We observe that using more worker threads than physical cores available (8) does not result in performance increases (even for CPU-bound processes). This is because each physical core corresponds to two virtual processors. However, the virtual processors

			`'Sherlock'`			`' [sS][A-Za-z]*[kK] '`		
			Wall (s)	CPU (s)	Wall (%)	Wall (s)	CPU (s)	Wall (%)
Cached Read	Threads	1	0.938	0.938	**100.0**	27.270	27.276	**100.0**
		2	0.548	1.046	**171.2**	14.082	27.944	**193.7**
		4	0.408	1.438	**229.9**	7.134	28.492	**382.3**
		8	0.426	2.786	**220.2**	3.654	29.054	**746.3**
		16	0.434	2.790	**216.1**	3.696	29.376	**737.8**
	mmap	yes	0.424	1.450	**100.0**			
		no	1.680	2.786	**25.2**			
SSD Read	Threads	1	2.734	3.122	**100.0**	28.876	29.168	**100.0**
		2	1.888	3.542	**144.8**	15.028	30.254	**192.1**
		4	1.612	4.624	**169.6**	7.662	30.816	**376.9**
		8	1.624	7.376	**168.3**	4.002	31.902	**721.5**
		16	1.632	7.164	**167.5**	4.020	32.048	**718.3**
	mmap	yes	1.604	4.498	**100.0**			
		no	3.512	4.574	**45.7**			
HDD Read	Threads	1	34.526	4.188	**100.0**	41.642	28.792	**100.0**
		2	37.534	4.198	**92.0**	45.736	43.178	**91.0**
		4	37.842	3.944	**91.2**	37.044	53.766	**112.4**
		8	37.868	4.196	**91.2**	36.216	57.380	**115.0**
		16	37.834	4.194	**91.3**	36.054	57.736	**115.5**
	mmap	yes	37.472	4.134	**100.0**			
		no	25.674	4.940	**146.0**			

Table 1: *xs grep* Configuration (number of worker threads ("Threads") and whether to use memory mapping ("mmap", "yes") or not ("no")): The listed benchmarks consider two differently complex search patterns (`Sherlock` and `' [sS][A-Za-z]*[kK] '`) and three different secondary memory types (RAM Cache, SSD and HDD).

share the same processing unit while utilizing different registers. Therefore, each physical core can run two threads in parallel but can only perform arithmetic operations for one of them simultaneously. This can increase performance in specific scenarios: When one thread cannot compute (e.g. because of cache misses, I/O time, dependencies of sequential instructions, ...), the other thread can start computing without a precedent context switch because the second thread runs on a virtual processor that has its own registers. However, in our scenario, utilizing SMT decreases performance for most measurements which is most likely due to the shared resources of the virtual processors: The threads work on relatively large data (within the range of multiple megabytes) stored in RAM. The RAM bandwidth the physical core can utilize might be at capacity for a single virtual processor. Further, virtual processors share the same cache and thus changing the context may result in cache misses since the virtual processors work on different data. Thus, utilizing a second virtual processor cannot further increase performance.

Trading Real Time for Computing Time

Earlier, we observe that adding worker threads increases performance given that the process is CPU-bound. We can trade wall time for CPU time until the process swaps from CPU-bound to I/O-bound. This implies decreasing wall time while increasing CPU time. However, we can also see that for I/O-bound processes, additional threads cannot further increase wall time but may increase CPU time. Thus, tuning the number of threads to a value that achieves reasonable wall time increases while not increasing CPU time too heavily is important. We observe an extreme case of this scenario when looking at the cached read, literal search setup (table 1): When using at least eight threads, the wall and CPU time increase (compared to using four threads). Thus, using more than four worker threads increases computing time while not decreasing wall time.

Memory Mapping is Slow on HDD

From the memory mapping benchmarks (mmap rows in table 1), we can see that data can be provided significantly faster when using memory mapping for reads from RAM cache (factor 4) and SSD (factor 2). The opposite can be seen for reads from HDD: Not using memory mapping results in faster wall time (factor 1.5). However, it must be remarked that this cannot be generalized for HDD reads. The reader used by *xs grep* that uses memory mapping (mmap-reader) is not optimized for reads from secondary memory with slow random access (such as HDDs): Memory mapping requires reading chunks starting at a multiple of the systems page size. Also, the chunk size must be a multiple of the system's page size. However, the mmap-reader provides data starting right after and ending with a newline character. Thus, the reader aligns the file pointer to positions that fulfill the memory mapping requirements while the actual needed data are within the range of the mapped data. Therefore, the mmap-reader maps overlapping chunks of the file. This does not result in significant overheads when the secondary memory from which data are mapped has fast random access (e.g. cache and SSD) but can result in high overheads for reads from HDD (resulting from physically moving the magnetic reader head of the HDD).

x-search's Thread Management Performs Well in Practice

We only consider the heavily CPU-bound scenario to allow for a valid statement on the thread management's performance (cached read and search for `' [sS][A-Za-z]*[kK] '`, table 1). Here, we can see that doubling the number of worker threads ($1 \rightarrow 2 \rightarrow 4 \rightarrow 8$) results in a speedup of more than 1.9 for each. Theoretically optimal would be a speedup of 2. However, a small overhead of utilizing multiple threads must always be considered. We cannot observe a performance increase when doubling the number of worker threads from 8 to 16. The reason for this is described in the previous section (SMT cannot Increase Performance).

Conclusion: Defining Default Values for xs grep

From the results described above, we have set the following default settings for *xs grep*:

Parameter	Default Value	Explanation
Worker Threads	$\frac{1}{4}$ of available processors	Using only a single worker thread is best for low CPU times. However, for CPU-bound processes, multiple worker threads can increase performance. In most cases, using $\frac{1}{4}$ of the available processors is considered a good compromise.
Memory Mapping	on	Most modern computers operate using SSDs as secondary memory. Further, a common use case of a command-line search utility is to search different patterns within the same data. Therefore, data are cached by the operating system after the first search and subsequent searches perform better when using memory mapping. However, it is recommended not to use memory mapping (by specifying the `--no-mmap` flag) when searches are performed on data stored on HDD.

If not stated otherwise, these default values are used for *xs grep* within the comparison benchmarks of section 7.2.2.

7.2.2 Search Tool Comparison

		Cached Read								
		`'Sherlock'`			`'She[r ]lock'`			`' [sS][A-Za-z]*[kK] '`		
		Wall	CPU	Wall %	Wall	CPU	Wall %	Wall	CPU	Wall %
Line	ripgrep	0.836	0.830	**100.0**	1.262	1.256	**100.0**	21.166	21.162	**100.0**
	GNU grep	3.206	3.204	**26.1**	3.930	3.926	**32.1**	33.032	33.026	**64.1**
	xs grep -j 1	0.956	0.964	**87.4**	1.108	1.120	**113.9**	11.314	11.324	**187.1**
	xs grep	0.424	1.458	**197.2**	0.436	1.504	**289.4**	3.022	12.068	**700.4**
	xs grep -j	0.450	2.808	**185.8**	0.442	2.816	**285.5**	1.554	12.280	**1362.0**
Line Number	ripgrep	1.102	1.092	**100.0**	1.526	1.522	**100.0**	21.292	21.286	**100.0**
	GNU grep	5.832	5.828	**18.9**	6.542	6.538	**23.3**	35.924	35.916	**59.3**
	xs grep -j 1	3.886	3.900	**28.4**	4.080	4.092	**37.4**	14.944	14.956	**142.5**
	xs grep	1.286	4.726	**85.7**	1.296	4.758	**117.7**	3.882	15.484	**548.5**
	xs grep -j	0.894	6.144	**123.3**	0.894	6.158	**170.7**	2.010	15.822	**1059.3**
Ignore Case	ripgrep	1.472	1.466	**100.0**	2.468	2.462	**100.0**	21.150	21.146	**100.0**
	GNU grep	13.042	13.040	**11.3**	13.236	13.234	**18.6**	35.374	35.368	**59.8**
	xs grep -j 1	2.216	2.224	**66.4**	11.796	11.806	**20.9**	27.384	27.392	**77.2**
	xs grep	1.714	6.436	**85.9**	3.078	12.184	**80.2**	7.106	28.358	**297.6**
	xs grep -j	1.770	13.146	**83.2**	1.598	12.432	**154.4**	3.632	28.794	**582.3**

Table 2: Search tool comparison benchmarks (cached reads): Wall, CPU and relative Wall (Wall %) measurements for patterns of different complexities (`'Sherlock'`, `'She[r ]lock'` and `' [sS][A-Za-z]*[kK] '`). Three different searches are listed: searching for lines, line numbers and case-insensitive searches (Ignore Case). Each row represents a single experiment. The best wall time results are highlighted in green.

		SSD Read								
		`'Sherlock'`			`'She[r ]lock'`			`' [sS][A-Za-z]*[kK] '`		
		Wall	CPU	Wall %	Wall	CPU	Wall %	Wall	CPU	Wall %
Line	ripgrep	2.840	2.760	**100.0**	3.272	3.202	**100.0**	23.384	23.324	**100.0**
	GNU grep	5.054	5.038	**56.2**	5.798	5.784	**56.4**	35.182	35.164	**66.5**
	xs grep -j 1	2.730	3.118	**104.0**	2.894	3.284	**113.1**	13.224	13.422	**176.8**
	xs grep	1.608	4.538	**176.6**	1.614	4.550	**202.7**	3.598	14.630	**649.9**
	xs grep -j	1.626	7.112	**174.7**	1.630	7.406	**200.7**	1.978	15.736	**1182.2**
Line Number	ripgrep	3.174	3.106	**100.0**	3.576	3.504	**100.0**	23.628	23.572	**100.0**
	GNU grep	7.708	7.694	**41.2**	8.424	8.410	**42.5**	37.786	37.770	**62.5**
	xs grep -j 1	5.788	6.174	**54.8**	5.960	6.330	**60.0**	16.940	17.216	**139.5**
	xs grep	2.102	7.946	**151.0**	2.164	8.072	**165.2**	4.556	18.336	**518.6**
	xs grep -j	1.790	12.190	**177.3**	1.766	11.834	**202.5**	2.494	19.760	**947.4**
Ignore Case	ripgrep	3.496	3.428	**100.0**	4.484	4.422	**100.0**	23.402	23.340	**100.0**
	GNU grep	14.928	14.912	**23.4**	15.002	14.988	**29.9**	37.438	37.422	**62.5**
	xs grep -j 1	4.288	4.664	**81.5**	13.610	13.904	**32.9**	28.866	29.160	**81.1**
	xs grep	2.704	10.094	**129.3**	3.618	14.692	**123.9**	7.654	30.806	**305.7**
	xs grep -j	2.720	21.18	**128.5**	1.990	15.790	**225.3**	3.994	31.814	**585.9**

Table 3: Search tool comparison benchmarks (SSD reads): Wall, CPU and relative Wall (Wall %) measurements for patterns of different complexities (`'Sherlock'`, `'She[r ]lock'` and `' [sS][A-Za-z]*[kK] '`). Three different searches are listed: searching for lines, line numbers and case-insensitive searches (Ignore Case). Each row represents a single experiment. The best wall time results are highlighted in green.

			HDD Read		
			'Sherlock'		
	File Size (GB)		Wall	CPU	Wall %
Plain	9.3	ripgrep	25.810	3.462	**100.0**
		GNU grep	25.632	7.998	**100.7**
		xs grep	25.662	4.826	**100.6**
ZStandard	1.5	zstcat → ripgrep	9.518	8.068	**269.4**
		zstcat → GNU grep	9.922	8.058	**260.1**
		zstcat → xs grep	9.484	8.294	**272.1**
		xs grep (xspp)	3.976	8.116	**649.1**
LZ4	3.7	lz4cat → ripgrep	8.184	6.332	**315.4**
		lz4cat → GNU grep	13.096	7.896	**197.1**
		lz4cat → xs grep	8.228	7.586	**313.7**
		xs grep (xspp)	8.862	6.438	**291.2**
LZ4 HC	2.1	lz4cat → ripgrep	6.446	5.750	**400.4**
		lz4cat → GNU grep	11.614	6.680	**222.2**
		lz4cat → xs grep	7.102	7.040	**363.4**
		xs grep (xspp)	5.358	5.744	**481.7**

Table 4: Comparison benchmark results for HDD reads: Comparing *GNU grep*, *ripgrep* and *xs grep* using plain text and compressed (ZStandard, LZ4, LZ4 HC) input. The 'File Size' Column indicates the size of the input data (in GB). The rows indicated with 'Plain' are the results for reading plain text data. Rows indicated with 'ZStandard', 'LZ4' and 'LZ4 HC' contain the results for input data that were compressed using the corresponding algorithm, respectively. The arrow notation (x → y) indicates that the input data were passed to x and x's output piped as input to y. *xs grep* is run using the `--no-mmap` flag.

Similar Results for HDD Reads

When comparing the results for HDD reads, we see that *GNU grep*, *ripgrep*, and *xs grep* perform similarly on plain text input (c.f. table 4). This is because the process is

heavily I/O-bound. Therefore, the underlying algorithms' runtime fades into the reading's runtime, which is hardware and OS-dependent.

While decompression predominates the processes' runtime when reading data from fast secondary memory, it can increase the overall runtime when used on slow secondary memory. Therefore, we consider the benchmark results for compressed data input when looking at the HDD benchmark results (table 4) while predominantly highlighting differences in the search performance when looking at the benchmark results of cached and SSD reads (c.f. tables 2 and 3).

Fast Secondary Memory: The Search Algorithm Becomes the Limiting Factor

Table 2 and 3 show that the search algorithms become the limiting factors even for the literal substring search (pattern `'Sherlock'`) when reading from fast secondary memory such as SSD or RAM cache. Here, the CPU time is strictly greater or equal (within the range of minor deviations) to the corresponding wall time. Thus, searching the data dominates reading the data, and the underlying search algorithm limits the overall runtime. We will now look closer at the search algorithm implementations used by the different search tools. The Boyer-Moore implementation used by *GNU grep* searches for the last byte of the pattern using `memchr` whenever it can skip bytes. *ripgrep* also implements Boyer-Moore but with a remarkable difference. In contrast to the *GNU grep* implementation of Boyer-Moore, *ripgrep* uses heuristics to *guess* the *rarest* byte within the pattern and then searches for this byte instead of searching for the last byte whenever it can skip bytes. Both use `memchr` to search for the corresponding byte. Since `memchr` is auto-vectorized[2] by modern compilers, this search can be done relatively fast compared to the overall substring search process. In our case (pattern `'Sherlock'`), *GNU grep* runs `memchr` on `'k'`, while *ripgrep* runs it on *'S'*. In our data set, *'k'* occurs about 2.8 times more frequently than `'S'` does (81,258,453 vs 29,018,097). Therefore, *ripgrep* can

[2] *Automatic Vectorization* describes the process of the compiler transforming scalar implementations into vector implementations by utilizing, e.g. SIMD if possible.

skip far more bytes using the vectorized `memchr` and thus becomes much faster. *xs grep* uses an entirely different search algorithm described in section 5.5. Even though this algorithm does not use any heuristics, it performs well. This is because of the additional condition of searching the pattern's first and last byte in the expected distance using SIMD instructions. The computation becomes more complex because two bytes are searched, and additional operations must be performed to check their distance. However, the number of false negative expensive inner loop runs (comparing the bytes between the first and the last byte of the pattern and the data) is much smaller in most cases: e.g. `'S.....k'` (`'.'` as a wildcard for any byte) only occurs 211,305 times which is over 130 times less frequent than `'S'`. Thus, the overall runtime of the search procedure decreases in most cases.

CPU-Bound Searches: xs grep Outperforms Alternatives

Our benchmarks consider different scenarios in which the procedure is CPU-bound: For cached and SSD reads, all runs are CPU-bound. For HDD reads, the process becomes CPU-bound if the input data are compressed using *ZStandard* or *LZ4 (HC)*. In those cases, *xs grep* can outperform its alternatives by up to 1300 % (c.f. table 2). Table 2 and table 3 show that *xs grep* outperforms *GNU grep* and *ripgrep* in almost all benchmarks (except for case-insensitive literal search for `'Sherlock'` when reading from cache, table 2). This is because *xs grep* is the only tool considered within this benchmark that utilizes multiple worker threads when searching on a single file. In the *xs grep Configuration Benchmarks* (c.f. section 7.2.1), we have already seen that using multiple worker threads can speed up performance. However, it must be said that this speedup trades with power efficiency: Utilizing multiple worker threads increases CPU time, and thus the power usage of the search increases as well.

ripgrep is the most Power Efficient

When looking at the CPU time spent within the different scenarios, *ripgrep* comes out as the absolute baseline. In most scenarios, *ripgreps* CPU time undercuts those of *GNU grep* and *xs grep*. This is primarily due to the optimized search algorithm implemented within *ripgrep* described earlier.

Case-Insensitive Literal Searches

When looking at the case-insensitive searches of the benchmarks listed in table 2 and table 3 (row: 'Ignore Case'), we see that *ripgrep* outperforms *xs grep* in some cases. It is also notable that *GNU grep* performs badly for case-insensitive literal searches. The reason for the named observations lies within the used search algorithms. All three tools approach the challenge of case-insensitive searches in another way. For case-insensitive searches, Boyer-Moore cannot be applied anymore the way it is for case-sensitive searches. *GNU grep* falls back to its regex engine for case-insensitive searches, which results in high wall and CPU times. *xs grep* transforms the pattern and input data into lowercase and then runs a case-sensitive search on the transformed data. *xs greps* approach is costly when using a single worker thread. *ripgreps* high-performance results from a unique search algorithm called Teddy. *Teddy* was introduced in 2019 by Andrew Gallant, the author of *ripgrep*. *Teddy* is a multiple substrings matching algorithm implemented using SIMD instructions. *ripgrep* constructs multiple literals out of a prefix of the provided pattern covering all possible case distributions (e.g. for *'Sherlock'*, the following literals can be built: `she|She|sHe|shE|SHe|ShE|sHE|SHE`) and runs *Teddy* on them. For reported matches, the remaining pattern must be compared to subsequent bytes of the data using a naive approach of case-insensitive matching. However, the filtering done by *Teddy* reduces the overall number of case-insensitive comparisons that must occur.

Google/RE2 fails in Regex Pattern Optimization

When looking at the benchmark results for the pattern `' [sS][A-Za-z][kK] '` from table 2 and table 3, we see that *ripgrep* and *GNU grep* perform similarly for the case-sensitive and case-insensitive search respectively. *xs greps* runtime for the case-insensitive search is much higher than the case-sensitive search. The interesting part here is that for this specific pattern, case-sensitive and case-insensitive searches do not affect the actual search because the pattern already is case-insensitive: Every character within the pattern can already either be upper- or lowercase from the regexes definition. However, the regex engine used by *xs grep* (Google/RE2) does not recognize this optimization possibility. It starts a more complex search, which increases wall and CPU time. However, *xs grep* is still faster than *ripgrep* and *GNU grep* because of its multithreading capabilities.

Searching Line Numbers

This section considers the benchmarks for searching line numbers (row 'Line Number') listed in table 2 and table 3. We observe a decreasing performance with regard to all three search tools when searching line numbers instead of lines. We have explained the reason for this in section 4. The benchmark runs of *xs grep* record the highest performance decrease: For *xs grep*, the wall time quadruples when searching for line numbers instead of searching for lines. The reason for this is the following. *GNU grep*, and *ripgrep* are searching for newline characters and the provided pattern simultaneously. *xs grep* in contrast, first searches newline characters and stores mapping data (described in section 5.5), then performs the substring search and finally maps the substring search results to line indices using the collected mapping data. This is more work because *xs grep* traverses the data twice; thus, searching line numbers is much slower for *xs grep* when using a single worker thread. However, *xs greps* approach can be easily parallelized: The newline character searching and the substring searching can run concurrently on the same data. Only mapping the byte offsets to the line indices requires the results of both searches. Since the mapping is relatively fast in most cases (because, in most cases, the number of matches is

small compared to the input data size), the overall runtime results in the runtime of the slower search (either newline character or pattern search). Therefore, *xs grep* achieves better results for line number searching concerning the wall time than *GNU grep* and *ripgrep* when using multiple worker threads.

Compressed Input: Trading I/O for Computing Time

We can see benchmark results for compressed data input in table 4. We have seen that reading data from HDD is the bottleneck within the overall procedure in table 1. Therefore, compression can increase the overall performance: Compression trades I/O for computing time by lowering the size of the data that must be read but increasing the computational effort because the read data must be decompressed before being searched. We can see the benefit of compressing data when I/O time is the bottleneck in table 4. All three compression algorithms (Zstandard, LZ4 and LZ4 HC) perform at least twice better than reading uncompressed data. The fastest option in our setup is compressing the input using the *x-search*-specific preprocessing using the ZStandard algorithm (xs grep (xspp), ZStandard). In this case, a speedup of over 600 % compared to reading uncompressed text data can be reached. For the LZ4 HC compression, the *x-search*-specific preprocessing also achieves the best benchmarking results. The reason for this is that *xs grep* can decompress the read data using multiple threads. Therefore, the best results are achieved when reading the data as fast as possible, which is the case for the ZStandard compressed data (size 1.5G). For the LZ4 compressed input, it is faster to use `lz4cat` for decompression and piping its output to one of the search tools than using the *x-search*-specific processing. This is because the LZ4 compressed file is still relatively large (3.7G), and the decompression of LZ4 compressed data is fast (c.f. decompression benchmark comparisons listed on LZ4's GitHub repository). Thus, the overall process is still I/O-bound and using multiple worker threads cannot increase performance.

Secondary Memory Affects Search Tool Requirements

Another important observation is that secondary memory affects the search tool requirements: We have seen that the search algorithm becomes the limiting factor for fast secondary memory (SSD and RAM cache). Thus, it is essential to implement fast and efficient search algorithms within the tools. Moreover, if the process is heavily CPU-bound, it is worth using multiple worker threads for searching. However, if the secondary memory is slow (HDD), the process becomes I/O-bound, and the underlying search algorithm does not affect wall times. However, using efficient and fast search algorithms affect the process's power efficiency (CPU time). Thus, when reading from slow secondary memory, it is recommended *not* to use multiple worker threads since their overhead may result in higher CPU time without affecting wall time.

8 Conclusion

This thesis has introduced an extensible and efficient backbone for implementing pipeline-based procedures named *x-search*. *x-search* is optimized for external searches and therefore provides many predefined components that can be used for external searches. The external string search implementation provided by *x-search* meets the requirements for solving the *External String Searching Problem* (ESSP). The thread management used within *x-search* has been proven to be optimal, and the benchmark results show that the thread management also performs well in practice. We have also introduced a *GNU grep*-like executable (namely *xs grep*) that outperforms *GNU grep* and *ripgrep* especially for CPU-bound searches. Thus, *xs grep* is a solid alternative to *GNU grep* and *ripgrep* for searching on single files. We have seen how compression can increase performance for slow secondary memory up to over 600 %. Further, from the SSD benchmarks, we can conclude that external substring search is not necessarily I/O-bound; thus, the implemented underlying algorithms affect the runtime of the overall process.

Further Work

This final section lists possible approaches to further extend and optimize *x-search* and its components.

Optimize Search Algorithm

The search algorithm used for literal searches works well for standard substring searches. However, searching for line indices or performing case-insensitive searches remains a subject for optimization. Extending the search algorithm used by *x-search* could improve performance for case-insensitive, line number and standard literal searches:

- **Case-Insensitive Search:** As described, case-insensitive searching is done by transforming the pattern and the data to lowercase and then performing a case-sensitive search. This works well when using multiple worker threads. When using a single worker thread, it can be more efficient to implement a case-aware substring search algorithm rather than transforming the data first. We have mentioned the multiple substrings matching algorithm *Teddy* that is used by *ripgrep*. A possible improvement would be to implement a similar algorithm for *x-search*. However, an implementation of *Teddy* requires a lot of low level optimizations and a deeper understanding of the available SIMD instructions for serious performance increases. Therefore, the task is considered relatively time intensive.
- **Line Number Search:** So far, searching for line numbers has also required a pre-search step: Before searching for matches, the positions of newline bytes are mapped to their line index. This means that the data are traversed twice when searching for line numbers of matches. A more efficient implementation could be achieved by extending the current SIMD-based string-matching algorithm: When searching for matches of the first and last byte of the pattern, newline bytes are also searched, and a corresponding counter is incremented if a new line is found. The search function then returns the relative line number of the provided data along with the byte offset if a match is found. This would reduce the byte offset search time compared to the original algorithm but increase the single-core performance when searching for line numbers. Extending the existing search algorithm to search for new line characters and the pattern simultaneously is considered trivial.
- **Literal Search:** We could also extend the existing SIMD-based string matching algorithm by applying heuristics similar to *ripgrep* that specify the bytes within the pattern that are searched using SIMD instructions. If the bytes used for this first search are as rare as possible within the data, the search algorithm will enter the expensive inner loop less frequently for comparing all bytes of the pattern and thus the overall performance increases. Extending the existing algorithm to support search of other characters than the first and the last one can be done in a trivial way and thus should not be too time intensive.

Optimize Readers

We have noted the performance decrease when using memory mapping on secondary memory with slow random access. This may not be a general issue of such secondary memory but relies on the current implementation of the reader. Therefore, the memory mapping reader could be optimized for using less file pointer movement while reading. This could be done by buffering larger data ranges and providing parts of the buffer when chunks are requested. Thus, larger regions of the input file would be mapped at once, and the file pointer would be moved less. Another possibility would be to detect slow random access while reading and switching to another reader.

Adding Support For Multiple Input Files

In contrast to *ripgrep* and *GNU grep*, *xs grep* only supports single file inputs. This could be approached by

- Implementing additional readers, that traverse directories and accept multiple input paths, and then provide data of all the input files.
- Sourcing the directory traversing out and constructing an *Executor* object for each file.

Adding Readers for Reading Shared Data

In order to be able to use *x-search* for searching data owned by another process, an additional reader must be implemented that provides `const` data from the foreign process. By default, the data object used by *x-search* already supports such approaches since memory-mapped data are considered foreign and `const`. Therefore, implementing such a reader is a trivial task.

Optimizing the Metafile

Following, we list possible improvements of the metafile format and the values that should be stored in the metafile:

- Drop `Original Size` and `Original Offset` values if no compression was used since these data correspond to `Actual Size` and `Actual Offset`, respectively.
- Use smaller integer types: If possible, store values as `uint32_t`.
- Add character heuristics (e.g. frequency distributions) for faster substring searches.

9 Acknowledgements

First of all, I would like to thank Prof. Dr. Hannah Bast for admission into the group and for giving me the opportunity to work on such an interesting topic.
I would also like to thank my supervisor Johannes Kalmbach for all time and effort you put into helping, debugging and explaining everything to me.
Last but not least, I would like to thank my family, especially Svenja, for all the support I have received during this long work period.

Bibliography

Baeza-Yates, R. A., C. Choffrut, and G. H. Gonnet (1994). "On Boyer-Moore automata". In: *Algorithmica* 12.4-5, pp. 268–292. DOI: 10.1007/bf01185428.

Barsky, Marina et al. (2008). "A new method for indexing genomes using on-disk suffix trees". In: *Proceeding of the 17th ACM conference on Information and knowledge mining - CIKM '08*. ACM Press. DOI: 10.1145/1458082.1458170.

Beall, Jeffrey (Sept. 2008). "The Weaknesses of Full-Text Searching". In: *The Journal of Academic Librarianship* 34.5, pp. 438–444. DOI: 10.1016/j.acalib.2008.06.007.

Blumer, Anselm, Andrzej Ehrenfeucht, and David Haussler (1989). "Average sizes of suffix trees and DAWGs". In: *Discrete Applied Mathematics* 24.1-3, pp. 37–45. DOI: 10.1016/0166-218x(92)90270-k.

Boyer, Robert S. and J. Strother Moore (1977). "A fast string searching algorithm". In: *Communications of the ACM* 20.10, pp. 762–772. DOI: 10.1145/359842.359859.

Charras, Christian and Thierry Lecroq (2004). *Handbook of Exact String Matching Algorithms*. Texts in algorithms. King's College. ISBN: 9780954300647.

Cox, Russ (2007). "Regular Expression Matching can Be Simple And Fast". In: *swtch*.

Farach-Colton, Martin, Paolo Ferragina, and S. Muthukrishnan (2000). "On the sorting-complexity of suffix tree construction". In: *Journal of the ACM* 47.6, pp. 987–1011. DOI: 10.1145/355541.355547.

Ferragina, Paolo and Roberto Grossi (1999). "The string B-tree". In: *Journal of the ACM* 46.2, pp. 236–280. DOI: 10.1145/301970.301973.

ISO/IEC (2023). *ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++*. Geneva, Switzerland: International Organization for Standardization (ISO).

Karp, Richard M. and Michael O. Rabin (1987). "Efficient randomized pattern-matching algorithms". In: *IBM Journal of Research and Development* 31.2, pp. 249–260. DOI: 10.1147/rd.312.0249.

Knuth, Donald E., Jr. James H. Morris, and Vaughan R. Pratt (1977). "Fast Pattern Matching in Strings". In: *SIAM Journal on Computing* 6.2, pp. 323–350. DOI: 10.1137/0206024.

McCreight, Edward M. (1976). "A Space-Economical Suffix Tree Construction Algorithm". In: *Journal of the ACM* 23.2, pp. 262–272. DOI: 10.1145/321941.321946.

Pierre Lison, Jörg Tiedemann (2016). "OpenSubtitles2016: Extracting Large Parallel Corpora from Movie and TV Subtitles". In: *In Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC 2016)*.

Ukkonen, E. (1995). "On-line construction of suffix trees". In: *Algorithmica* 14.3, pp. 249–260. DOI: 10.1007/bf01206331.