Design and Implementation of Parallelized AWK

Master's Thesis

Ivans Kravcevs

Design and Implementation of Parallelized AWK

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ivans Kravcevs
born in Riga, Latvia

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

©2026 Ivans Kravcevs. All rights reserved.

Design and Implementation of Parallelized AWK

Author: Ivans Kravcevs
Student id: 5450535

Abstract

The project presents the design and implementation of a system for automatic parallelization of AWK programs. AWK remains a widely used language for text processing and data transformation. It is included as a standard utility tool on most Unix-like systems. The execution model of AWK is traditionally sequential, which limits scalability on multi-core hardware. The goal of this work is to investigate whether static program analysis can identify AWK scripts that can be executed in parallel and to integrate this capability into an AWK interpreter.

The proposed solution introduces a static analyzer that evaluates AWK programs based on variable dependencies, control flow, and other behaviors that impact data dependencies. The analyzer identifies reduction patterns for global variables and determines whether program semantics can be preserved under parallel execution. These results are then integrated into the interpreter, which enables deterministic multi-threaded execution.

The project adopts the MapReduce programming model to enable parallel execution of AWK. The main processing phase of a script is treated as the map stage, where independent partitions of the input are processed concurrently by multiple workers. Intermediate thread-local results are then combined in a reduce stage using aggregation strategies derived from static analysis. This model provides a structured way to preserve AWK's sequential semantics in the parallelized environment.

The implementation was evaluated on a dataset of real-world AWK scripts and through performance benchmarks on large text-processing workloads. The results show that a significant subset of AWK programs can be parallelized automatically, achieving execution speedups and state-of-the-art AWK performance.

The project provides a practical path for improving efficiency in text-processing workflows. This work also demonstrates that scripting languages can often benefit from modern parallel execution techniques, extending their practical relevance and performance in data-processing tasks.

University supervisor:	Prof. Dr. D. Spinellis, Faculty EEMCS, TU Delft

Thesis Committee:

Chair:	Prof. Dr. D. Spinellis, Faculty EEMCS, TU Delft
Committee Member:	Asst.Prof. Dr. B. K. Ozkan, Faculty EEMCS, TU Delft
Committee Member:	Asst.Prof. Dr. S. Chakraborty, Faculty EEMCS, TU Delft

Preface

I would like to thank my thesis supervisor, Prof. dr. Diomidis Spinellis, for the guidance during the research process and continuous support. His thoughtful feedback and willingness to discuss both high-level ideas and low-level implementation details have been essential in bringing this work to its current form.

I am also grateful to Asst. Prof. dr. B. K. Ozkan and Asst. Prof. dr. S. Chakraborty for kindly agreeing to be in my thesis defense committee and for taking the time to review my work.

I would furthermore like to thank Delft University of Technology and the Software Engineering Research Group for providing an inspiring and supportive environment in which to carry out this research. I am also thankful to my family and friends for their encouragement and understanding throughout this journey.

Ivans Kravcevs
Delft, the Netherlands
June 11, 2026

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
1.1 Background 1
1.2 Problem statement 1
1.3 Research Objective and Approach 2
1.4 Research Questions 2
1.5 Summary of Findings 3
1.6 Thesis structure 3
1.7 Use of Generative AI tools 4

2 Related work 5
2.1 Parallel computing foundations 5
2.2 Parallelization techniques in compilers 6
2.3 Data-processing with MapReduce 7
2.4 Challenges in interpreted languages 8
2.5 Classical AWK implementations 9
2.6 Modern AWK versions 10
2.7 Automatic parallelization of AWK 11

3 Parallelizability analyzer 13
3.1 Analysis Overview 13
3.2 Global variable detection 15
3.3 Parallelization rules 17
3.4 Associative array parallelization 21
3.5 Function parallelization 22

4 Integration with Interpreter **29**
4.1 System architecture overview 29
4.2 Input parsing 29
4.3 Output Ordering 31
4.4 Variable preparation and aggregation 32
4.5 Unassigned values 33
4.6 Summary on the implementation 35

5 Evaluation **37**
5.1 Applicability analysis 37
5.2 Performance analysis 40

6 Conclusions and Future Work **49**
6.1 Contributions 49
6.2 Research outcomes 50
6.3 Limitations 51
6.4 Discussion/Reflection 52
6.5 Future work 53

Bibliography **55**

List of Figures

3.1 Parallelization analyzer flow 14
3.2 Parallelizable use of *split* in AWK 24
3.3 Non-parallelizable use of *split* in AWK 25
3.4 Parallelizable use of *sub* in AWK 25
3.5 Non-parallelizable use of *sub* in AWK 25
3.6 Custom AWK function with a parameter shadowing a global variable 26

4.1 Parallel AWK execution flow model 30

5.1 Distribution of 970 programs by parallelizability 39
5.2 Benchmarking script (Counter): Count orders from Europe 42
5.3 Benchmarking script (ArrayCounter): Count orders from Europe using an array by country 43
5.4 Benchmarking script (DuplicateFinder): Count duplicate rows by the first field . 44
5.5 Benchmarking script (MemoryHeavy): Count duplicate rows using an array . . 44
5.6 Benchmarking script (Functions): Count the number of records during a period of time 45
5.7 Benchmarking script (Output): Output the first field on each input line 46
5.8 Comparison of best-performing sequential and parallel AWK implementations . 47

6.1 Dynamically parallelizable pattern 54

Chapter 1

Introduction

This chapter introduces the context, motivation, and objectives of the research. It begins by presenting the AWK programming language and its role in text processing. The limitations of AWK in the context of modern data-processing workloads are then discussed, leading to the formulation of the problem statement. This forms the research objective and research questions. Finally, the chapter presents a summary on the main findings and results.

1.1 Background

AWK is a scripting programming language for text processing. It was developed in the 1980s by Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan [1] and is still widely used for data-driven tasks such as semi-structured document analysis, field extraction, data aggregation, and cleanup. AWK is a built-in command-line tool for most Unix-like environments and is used in many legacy automation scripts. This makes AWK one of the most recognizable and classical tools for text processing.

AWK was designed around a simple execution model where input data is read line by line, each line is matched against defined pattern-action rules, and the associated actions are executed immediately for each matched record. The language is optimized for concise text manipulation, so the most popular operations in AWK include matching and processing fields, performing simple calculations, and updating variables. All variables in AWK have global scope (can be accessed from any part of the script) and are often used to aggregate information from multiple input lines. The typical AWK script also contains special *BEGIN* and *END* statements that are executed before and after the actual data-processing script.

The paper by A. V. Aho et al. [1] gives a comprehensive overview of language features and use cases.

1.2 Problem statement

The AWK design makes it effective for many practical scripting tasks, but it also imposes inherent performance limitations. As already noted by A. V. Aho et al. [1], the execution model of AWK restricts its ability to handle large datasets efficiently.

Sequential execution becomes one of the major performance bottlenecks for AWK. Each input record is processed one after another in AWK, and script execution for line N starts only after script execution for line $N-1$ is finished. While this model aligns well with streaming data processing, it limits the ability of AWK programs to take advantage of modern multi-core processors.

Parallelizing the AWK language can significantly boost the initial AWK performance. It would make the language scalable for use on growing modern data volumes. However, parallelizing AWK programs is non-trivial due to several native language characteristics such as global variables, order-dependent operations, and implicit state shared across records. These features introduce dependencies that may violate correctness if execution is parallelized naively.

Although modern AWK implementations [28, 5] have improved the initial performance of the AWK interpreter, they still process text sequentially. This exposes a fundamental limitation in AWK's design: its simplicity in text processing comes at the cost of scalability for modern data-processing workloads.

1.3 Research Objective and Approach

This work aims to improve the scalability of AWK by exploring and applying parallel approaches in text processing. In particular, the thesis investigates how AWK programs can be executed in parallel following the MapReduce paradigm [8]. The input data can be partitioned into multiple chunks, which are processed independently on separate CPU cores during a mapping stage, followed by a reduction stage that combines the intermediate results. The special `BEGIN` and `END` statements in the AWK scripts are not parallelized; instead, they are executed before the mapping stage and after the reduction stage, respectively.

While this approach is not universally applicable to all AWK programs due to complex inter-record dependencies in AWK scripts, this work explores the conditions under which it can be applied and evaluates its effectiveness in improving performance.

The proposed solution is implemented by implementing a static parallelizability analyzer for AWK and integrating it into a modern AWK interpreter. This work develops a custom static analyzer tool to detect and correctly execute parallelizable AWK scripts. The static analyzer implements full AWK variable dependency analysis, based on some classical compiler parallelization approaches [3, 25]. This enables automatic detection and execution of parallelizable AWK programs without requiring initial script modifications from the user. It also guarantees the determinism and correctness of parallel script execution.

Additionally, the research evaluates the proposed solution by performing an applicability analysis on real-world AWK scripts. The thesis also performs a benchmarking study and compares the performance of parallel AWK with other popular AWK versions. Finally, it discusses the strengths and limitations of the proposed solution.

1.4 Research Questions

This work addresses the following research questions:

- How can AWK programs be automatically parallelized while ensuring correctness with respect to their original sequential semantics?
- Is the proposed approach for parallelization applicable to real-world text processing tasks?
- What performance improvements can be achieved by executing AWK programs in a parallel environment compared to sequential execution?

1.5 Summary of Findings

The thesis demonstrates that parallel execution of AWK programs is feasible and effective for a subset of programs that do not rely on complex shared state. It is achieved by developing a static AWK script analyzer and integrating into a modern AWK interpreter. The proposed static analysis is effective in identifying such programs by detecting global variable usage and inter-record dependencies. The connection of a static analyzer with an interpreter makes it an effective tool for the parallel execution of AWK scripts.

The integration of the analysis into the interpreter enables automatic selection between sequential and parallel execution, requiring no modifications to existing AWK scripts. This preserves the usability of all current AWK scripts while extending the execution model to better utilize modern multi-core systems.

Experimental evaluation shows that the parallel execution strategy can be applied for up to 27% of existing real-world AWK scripts without adjustments to the actual scripts. This demonstrates a high applicability of the proposed solution for modernizing existing AWK scripts.

Additionally, the performed benchmarking shows that the parallelized AWK interpreter can achieve multiple times improved performance in comparison to all traditional sequential AWK interpreters, particularly for large input datasets. While using the increased amount of memory for the computations, the parallel version of AWK achieves a state-of-the-art performance for most of the tested AWK scripts.

Overall, the results indicate that combining static analysis with a data-parallel execution model provides a practical way to enhance the scalability of AWK without compromising the simplicity and correctness of the language.

Full implementation code and evaluation scripts of this work are available as a published artifact [22] and on GitHub [1].

1.6 Thesis structure

This section gives a brief overview of the thesis structure. Chapter 2 reviews related work on parallel computing, automatic parallelization in compiler design, the MapReduce technique, and its relevance for AWK parallelization. Additionally, it gives a brief description of the most popular AWK implementations and their difference. Chapter 3 presents the static

[1] Available at: `https://github.com/ikrvc/parallel_frawk` (Accessed: 2026-06-01)

parallelizability analyzer for AWK programs. Gives the main principles, global variable dependency analysis, and aggregation rules. Chapter 4 explains how the analyzer is integrated into the interpreter and how input partitioning and variable aggregation are implemented. Chapter 5 evaluates the usefulness of the proposed solution by performing the applicability assessment on real-world scripts and performance benchmarking on large data volumes. Lastly, Chapter 6 concludes the thesis, discusses limitations, and outlines future work.

1.7 Use of Generative AI tools

Generative AI tools (ChatGPT[2], Perplexity AI[3]) were used during the preparation of this thesis to improve the readability of some sentences and for grammar checks. All technical content, research design, implementation, experiments, and analysis were developed and verified by the author without AI tool usage.

[2]Available at: `https://chatgpt.com/` (Accessed: 2026-06-01)
[3]Available at: `https://www.perplexity.ai/` (Accessed: 2026-06-01)

Chapter 2

Related work

This chapter reviews the existing literature and research work relevant to the topic. It begins with an overview of the theoretical foundations of parallel computing, followed by classical techniques for automatic parallelization in compiler design. The chapter then discusses the MapReduce paradigm and its applicability to text-processing tasks. Furthermore, challenges related to parallelizing interpreted and dynamic scripting languages are examined. Finally, the chapter provides an overview of classical and modern AWK implementations, highlighting their limitations and motivating the need for automatic parallelization of AWK programs.

2.1 Parallel computing foundations

Parallelization is a fundamental technique to improve the scalability of computations by splitting the work between multiple processing units. Today, data volumes grow rapidly, making parallelization unavoidable for applications ranging from scientific computing to large-scale data analysis.

Theoretical foundations establish fundamental limits and opportunities for parallel speedup. Amdahl's Law [4] defines the parallelizable execution limit:

$$\text{Maximal Speedup} = \frac{1}{(1-P)+\frac{P}{N}}$$

where P is the fraction of the program that can be executed in parallel and N is the number of processors. G. Amdahl in 1967 [4] in the paper was skeptical about the potential for the speedup using parallel processing due to the large sequential component and processor synchronization overhead. Later, J. Gustafson [15] in 1988 reevaluated Amdahl's law for scalable problems. Paper mentions that the time for the serial part of the program, such as start-up, program loading, and result aggregation, typically stays constant with growing problem size. It means that the fraction of parallelizability (P) will grow on a scaled problem, and the number of processors (N) will play a crucial role in the execution time. Gustafson defines time for a parallelized task execution based on Amdahl's law:

$$\text{Time} = (1-P)+\frac{P}{N}$$

where the time for a serial program execution is equal to 1. The paper states that the massive parallelism can be highly efficient for a scaled problem size.

Parallelization of computation has been a developing topic since then. L. Valiant [33] in 1990 describes a theoretical model for general-purpose parallel computation machine. The BSP model (Bulk Synchronous Parallel model), introduced in the paper, describes parallel computation in terms of such components: processors with local memory that do computations independently, exchange data through message passing via a communication network, and wait for each other for synchronization after completing some steps. This theoretical model had a strong influence on most of the modern parallel and distributed models.

Parallel Virtual Machine (PVM) [13] is one of the first parallel programming systems used in practice, which pioneered SPMD parallelization (single program, multiple data). PVM executed identical program instances across processors, each processing a different part of the data and coordinating through message passing and synchronization. The SPMD data-parallelization approach, used in PVM, aligns closely with the AWK parallelization approach discussed later in this paper.

2.2 Parallelization techniques in compilers

Automatic program parallelization was explored extensively in compiler design. Foundational book by R. Allen and K. Kennedy [3] formalizes strategies, such as loop transformations and vectorization, that remain central to automatic parallelization today. Book by S.P.Midkiff [25] describes similar principles with a larger focus on a modern perspective to parallelization techniques, often used in popular programming languages and compilers to automatically detect parallelizable parts of the programs. The main part of the book goes deep into the compiler design and describes the parallelization strategies for loops, such as detecting and avoiding the inter-cycle dependencies in loops. The book also describes the general structure of modern programming languages from a parallelization perspective. It defines program intermediate representations, most importantly the Abstract Syntax Tree (AST) and Control Flow Graph (CFG), that are used for program parallelization analysis by the compiler. It also gives multiple techniques for program parallelization analysis, such as dependence analysis and constant propagation.

Importantly, data dependence analysis is the cornerstone for most of the parallelization techniques. The parallelization optimization can be applied after variable/array dependencies are found. The paper by J. Ferrante [12] defines the Program Dependence Graph (PDG). Here nodes represent program commands, and edges are their dependencies. For example,

$$A = B * C \quad S_1$$

$$D = A * E \quad S_2$$

in these two statements S_2 depends on S_1, since A is defined in S_1 and used in S_2. The paper describes the dependency graph creation principles. It shows that PDG can be directly used to detect parallelizable parts of the program. The paper of V. Sarkar [30] builds on top of

PDG to create parallel program execution schedules that can be used directly in the parallel execution software.

The mentioned techniques have matured into a widely deployed compiler infrastructure. Dependence graphs power popular production systems like Polly [14] (a code optimization tool for LLVM), exploiting loop parallelism on multicore CPUs for performance optimization. Comprehensive overviews [3, 20] document various techniques and decades of refinement, confirming automatic parallelization as a mature compiler capability.

2.3 Data-processing with MapReduce

Computation parallelization in the distributed data processing field is based on different principles. A major milestone in this field is the MapReduce principle introduced by J.Dean and G. Sanjay [9], which became a standard in big-data distributed data processing and can also be applied for parallel data-processing settings. The principle is also known as the map-fold computational pattern, mostly in functional programming settings. MapReduce principle consists of 2 stages. During the mapping stage, the dataset is divided into fixed-size splits, and each part is processed by a map worker using the associated function. The tasks in this phase are typically scheduled so that computation is performed near where the actual data is stored in the distributed file system. Afterwards, during the reduction phase the results are transferred to reduction nodes and aggregated using special reduction functions.

MapReduce allowed large datasets to be processed efficiently and inspired a wide ecosystem of parallel data-processing systems, including Spark [39] and Flink [7]. Such systems are mainly developed to execute a computation over a distributed file system. However, often the dataset resides on a single machine or within a single storage location, yet it would still be beneficial to use parallel processing based on the size of the data. In these scenarios, a MapReduce-style approach can still be applied by partitioning the dataset into chunks, processing them in parallel, and subsequently aggregating the partial results.

The paper by C. Ranger et al. [27] evaluates the suitability of MapReduce principles for a shared-memory system. The Phoenix API framework presented in the paper shows a practical implementation of MapReduce for multi-core and multi-processor systems with shared memory. It demonstrates that MapReduce abstractions are useful not only for clusters with distributed memory but also for exploiting parallelism on a single machine. It also shows that the framework can achieve significant speedups with respect to sequential code execution. This illustrates that the MapReduce pattern transcends distributed file systems and can serve as a high-level parallel programming model even in non-distributed, shared-memory environments, bridging the gap between cluster computing and multi-core parallelism.

The book by J. Lin and C.Dyer [23] also discusses several limitations of the MapReduce principle in text-processing, despite all the benefits discussed earlier. According to J. Lin and C. Dyer [23], the MapReduce programming model imposes multiple limitations that arise mostly from a lack of shared global state during the data-parallel execution. In MapReduce, each task operates independently in isolation and communicates only during the shuffle phase. While this design is good for scalability by distributing data processing tasks, it complicates execution of any algorithms that require global state or cross-record dependencies. For

example, tasks that rely on shared mutable variables and global counters. As a result, although MapReduce is well-suited for large-scale batch processing, it can be less efficient and less expressive for workloads that require persistent global state or iterative computation.

While MapReduce and related distributed frameworks provide an effective model for processing large datasets, many practical data-processing workflows are still implemented using text-processing scripting languages such as Python, Perl [36], as well as AWK. Scripting languages are widely used for processing text files (e.g., CSVs, log files) and other data-oriented tasks due to their simplicity and flexibility [26]. In such tasks, each input record is processed sequentially, and the global state is often implicitly maintained.

Unlike compiled languages or distributed dataflow frameworks, most of the scripting languages were originally designed with sequential semantics, which makes automatic parallelization more challenging.

2.4 Challenges in interpreted languages

Despite the large amount of work on parallelizing compiled languages, there is a relatively small amount of research done on interpreted language parallelization.

The paper by S. Wei et al. [38] develops a tool to automatically find the parallelisms in Python programs. The paper achieves this by performing a deep data dependency analysis. The paper manages to achieve program performance acceleration; however, the paper mentions a lot of limitations of the current design, such as the high overhead of creating and recycling processes. Additionally, the parallelizable part of the program is relatively small. The paper mentions parallelization difficulties due to the mutable global state and dynamic type system.

There are relatively few attempts at automatic parallelization of general-purpose interpreted scripting languages. Achieving significant performance improvements through parallelization is challenging for such languages because many runtime characteristics of interpreted languages limit static analysis and optimization opportunities. This makes it practically impossible to apply traditional approaches for automatic parallelization.

By contrast, the semantics of record-oriented scripting languages, designed to specifically process information record-by-record, offer simpler opportunities for parallel analysis and execution. The paper by N. Vasilakis [34] presents a solution for the automatically parallelized Shell script execution. The Shell core operations are mostly simple record transformations and filters, that can be translated into a dataflow graph. The paper enables parallelism for independent partitions of input data using the semantics-preserving transformations based on dataflow graph analysis. Interestingly, the paper explicitly mentions AWK as one of the tools that can not be directly parallelized as part of the presented solution.

AWK itself follows a record-oriented execution model that is conceptually similar to many data-parallel processing paradigms. In a typical AWK program, input is processed sequentially and pattern–action rules are applied independently to each record. This structure suggests that many AWK programs could potentially be parallelized by partitioning the input data and executing the same program logic on multiple data segments concurrently. However, safe AWK parallelization requires careful analysis of program semantics, particularly the

use of global variables, associative arrays, and other forms of state that may introduce dependencies between records. Despite the apparent suitability of AWK for data-parallel execution, there is currently no scientific literature that systematically explores automatic parallelization of AWK programs. As a result, this area remains largely unexplored and motivates further investigation.

2.5 Classical AWK implementations

The AWK programming language [1], originally developed by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, remains a fundamental tool for text processing in Unix-like systems. AWK is a simple tool to write small and efficient programs for text processing that are interpreted directly on Linux terminal [21]. Over the years, several major implementations have emerged, the most prominent being GNU AWK (gawk), mawk, and "The One True AWK"(nawk).

All three implementations are written in the C programming language and preserve compatibility with the original language specification. However, all three implementations differ in additional features and performance characteristics.

- gawk [28] is the most widely used AWK implementation. It extends the AWK language with a lot of additional built-in functions, improved regular expression handling, and support for extensions. It is the default AWK on many GNU/Linux systems. Its variability makes it a perfect choice for the relatively complicated text-processing tasks.

- mawk [5] is designed with performance as a primary goal. It is often faster than gawk for simple tasks due to its efficient execution model. However, mawk lacks some of the advanced features and extensions available in gawk, which can limit its applicability in more complex scripts.

- nawk [2] represents an evolution of the classical AWK interpreter and is the closest to the standard version of the language described in the book by A. Aho et al. [1]. It aims to balance portability and correctness. It does not include many modern extensions found in gawk and is often much less performance-efficient than other versions. However, it is often the most memory-efficient version of AWK.

The Table 2.1 gives a high-level overview of the benefits and disadvantages of classical AWK implementations.

The common limitation of all classical AWK implementations is their inherently sequential execution model. AWK processes input record by record while maintaining global state, which often makes parallelization difficult without fundamentally altering the language semantics. As a result, classical implementations do not natively exploit modern multi-core architectures, limiting the performance of text processing.

Feature	gawk	mawk	nawk
Main benefit	Feature-rich	High performance	Standard compliance
Performance	Medium	Best	Low
Language extensions	Extensive	Some	Limited
Memory usage	Moderate	Moderate	Low
Typical use case	Complex scripts	Fast processing tasks	Standard-conform scripts

Table 2.1: Comparison of major AWK implementations

2.6 Modern AWK versions

The original AWK implementations are written in *C* language. Despite the popularity of C and the common standard of Unix tools being written in C, it is an old and complex language to use. C language is not type safe or memory safe, and it quite often contains easy-to-exploit security flaws [29].

In response to these limitations, several modern reimplementations of AWK have been developed in safer and more expressive programming languages. One such example is GoAWK, developed by B. Hoyt [17], an AWK interpreter written in the Go programming language. GoAWK aims to provide a clean and portable implementation of AWK with improved safety and maintainability compared to traditional versions. Ben Hoyt also provides a basic benchmarking for AWK versions [17], where GoAWK achieves similar performance to C-based versions of AWK with up to 2 times speedup in some tests. GoAWK offers a more modern alternative while remaining compatible with standard AWK behavior.

Another modern implementation is *frawk*[11] project. It can interpret and execute most AWK programs and often has better performance than the classic AWK versions. Additionally, it has support for some modern features that are not implemented in classic AWK (e.g CSV support out of the box).

frawk is implemented in the Rust programming language. Rust provides a modern, memory-safe alternative to C languages. Based on the paper by W. Bugden and A. Alahmar [6], Rust provides comparable performance to C in most programs. Additionally, Rust provides type safety and memory safety by introducing a compilation-time ownership checker. This drastically reduces the safety risks of the language. Additionally, Rust has a more modern syntax and packaging syntax, making it more convenient in modern scenarios.

frawk project also has an attempt to implement simple parallelization.*frawk* splits the data into chunks and copies all global variables declared in *BEGIN* statement between threads. The variables are aggregated into one using sum and can be used in *END* statement afterwards. *frawk* also introduces a special *PREPARE* statement on top of classical AWK, which is executed after the main statements on each thread to manually control where to save the global variable states for each thread.

2.7 Automatic parallelization of AWK

Despite the good initial attempt to implement parallelization in *frawk*, it has some major issues that are not solved. The most important one is that the parallelizability of AWK script should be manually controlled, as otherwise the result of the program executed in parallel can be inconsistent with the result of a sequentially launched program. Additionally, the variables are aggregated as a sum only, which is not always the aggregation operation that the program requires. The current *frawk* implementation does not guarantee the correctness of parallel execution, due to not preserving the original input ordering, very simple aggregation rules and no automatic parallelizability detection.

This project aims to solve these issues by defining and implementing the static AWK program analyzer, which will detect whether the program can be parallelized. Additionally, the static analyzer will detect the aggregation operation for each of the global variables based on the variable assignments in the main part of the AWK program. The static analyzer will be integrated into the *frawk* project, and the AWK interpreter will be adjusted to run programs in parallel based on the analysis results.

Chapter 3

Parallelizability analyzer

This chapter presents the design of a static parallelizability analyzer for AWK programs. The goal of the analyzer is to determine whether a given AWK program can be safely executed in parallel without changing its observable behavior compared to sequential execution. The analysis operates on the abstract syntax tree (AST) produced by the AWK parser and focuses on detecting dependencies that violate per-record independence. The chapter first introduces the detection of global variables, followed by a set of rules that define which language constructs can be parallelized. Finally, special cases such as associative arrays and function calls are discussed, highlighting their impact on parallel execution.

3.1 Analysis Overview

The static parallelizability analysis is based on the principle that each input record (text line) in an AWK program must be processed independently of the previous state to enable safe parallel execution. In a sequential execution model, records are processed one after another, and global program state is updated incrementally. In contrast, a parallel execution model evaluates multiple records simultaneously, requiring that intermediate computations do not depend on the processing order or shared mutable state. The program is classified as safe for parallel execution if executing it in parallel produces the same results as sequential execution.

To determine whether the AWK program is safe for parallel execution, the analyzer processes the script's syntax and performs a structured traversal of all relevant constructs. During this traversal, it collects information about variable assignments, variable usage, dependencies, and side effects. The analysis is currently conservative: if any construct cannot be proven to be safe under parallel execution, the entire program is classified as non-parallelizable.

The core idea is to identify all dependencies that span across input records. These dependencies primarily arise from global variables. The analyzer tracks how values are defined and used throughout the program, distinguishing between variables that are local to a single record and those that may carry state across records. Based on this information, it evaluates whether updates to shared variables can be expressed as order-independent reductions to collect the global state after parallel execution.

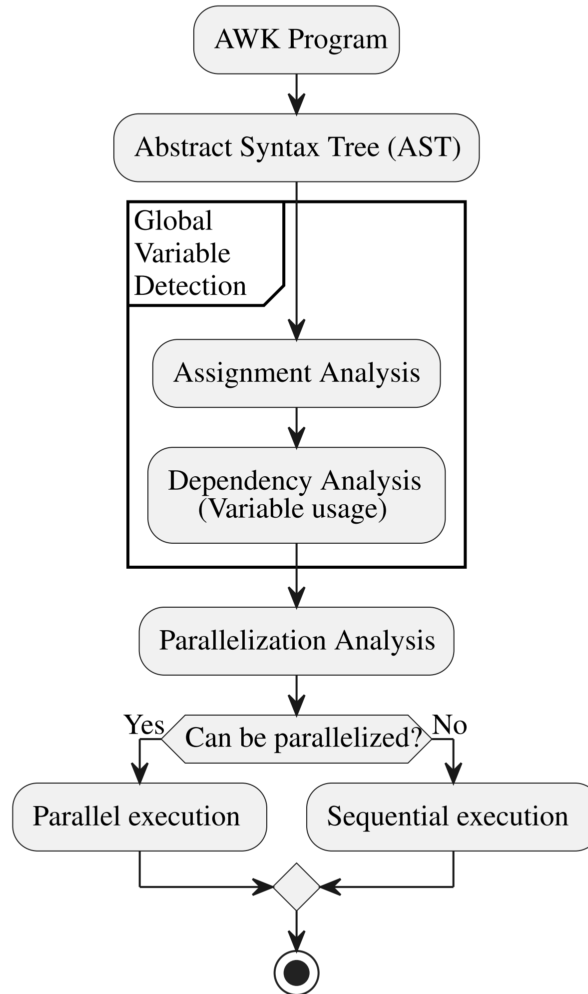

Figure 3.1: Parallelization analyzer flow

The analysis proceeds in multiple stages. Figure 3.1 shows the full flow of the analyzer. First, variables that may act as global state are identified by analyzing the AST. Initially, assignments are analyzed to detect potential global variables. Next, the dependencies between potential global variables and their usage is explored to detect truly global variables. After the global variables are found, the program is checked against a set of rules that define safe patterns for parallel execution, including restrictions on assignments, conditionals, and output operations. It detects whether the program can be executed in parallel or the interpreter should fall back to sequential execution.

The special AWK statements *BEGIN* and *END*, which are executed only once per program, are not included in the parallelizability analysis. These statements are used for initial setup before text processing and final calculations after processing. Since they do not operate on a per-record basis, they are not suitable for data-parallel execution and are therefore always executed sequentially. In this work, all AWK program statements, excluding

these special blocks, are named as the main program part and are explored for potential parallelism.

By combining these steps, the analyzer determines whether the AWK program satisfies the requirements for parallel execution or should be executed in a conservative sequential way.

3.2 Global variable detection

The global variables keep their value throughout the program and, therefore, are important in parallelization analysis. Global variables in a parallel execution setting hold thread-local intermediate values during the main program execution stage and are only combined during the reduction stage. As a result, any statement that accesses a global variable during the main program would observe a partial and thread-specific value rather than the value that would be produced by a sequential execution. This leads to output that can be inconsistent with the sequential semantics of AWK. Due to that, the AWK program should be analyzed for global variables and their usage before it can be safely executed in parallel.

Any variable used within the program in AWK is global by default. It means that the variable defined at one point can be accessed from all parts of the script. Additionally, the variable does not need to be declared and keeps the default value if used before declaration.

This project defines global variables in a different way for parallelization analysis. The variables that are important for parallelizability are only the variables that are non-constant during the main part of the program (have at least one assignment). Additionally, the variable is global only if the value assigned during the script execution for one record can be reused during the script execution for another line of text (e.g the variable that is assigned a new value for each record independently of the previous program state should not be considered global).

Global variable detection is done in two separate stages. The first stage traverses the main program AST to detect all assignments and find potentially global variables. The second stage detects truly global variables by analysing dependencies between potentially global variables.

3.2.1 Assignment analysis

The first step of the static parallelizability analysis detects all potentially global variables. This is done by analysing the assignments in the AWK script.

There are three ways to assign a new value to the variable in AWK:

- Classical assignment (e.g $a = b + c$): here left-hand side of the expression is assigned a new value. The variable on the left-hand side is extracted as a potentially global.
- Operator assignment (e.g $a += b + c$): this assignment is a special case of the classical assignment. Here the variable on the left-hand side of the expression is re-assigned new value based on it's previous value and the expression on the right side, this variable is also potentially global

- Increment operation (e.g $a++$): this expression also contains variable value re-assignment and should be considered here.

Special case is when the left-hand side variable is a field variable (e.g $\$1 = b + c$). The field variables contain record information and are reset after each line; therefore, new assignments to such variables will contain local, record-specific information. These assignments are skipped by the global variable detection algorithm.

This set of rules helps to find all the variables potentially impacting parallelization. The deeper dependency analysis is performed on those variables to detect the true global variables that capture information across the records.

3.2.2 Variable scope analysis

The second stage of the global variable detection algorithm checks the scope of usage of global variables. The variables containing local values are removed from further analysis at this stage. These are the variables that are assigned a constant value or one of the field variables' values. Most importantly, these are the variables for which any usage in the expressions occurs strictly after their assignment in the script. This means that the variable does not aggregate any information between records and can be treated as a record-local variable.

The second stage is implemented by maintaining an environment that is passed along during the traversal of the abstract syntax tree. The environment keeps track of any assigned variable in the current scope. A variable is classified as truly global if it is used in a statement, but no corresponding variable assignment is found in the current environment at that point in the traversal. Additionally, the variable is classified as truly global if it depends on the value of another truly global variable.

The environment is implemented as a stack of sets. Each set contains the variables defined within a particular scope, while the stack structure captures nested scopes (such as branches and blocks in code). Upon entering a new scope, a new set is pushed onto the stack, and upon exiting the scope, it is removed.

As an example, in the case of if statement, a new set is added on top of the stack, and it will contain all the variables defined inside the if statement. The upper stack element is removed after the if statement is processed. An additional scenario is if the variable is declared in both the if and the *else* statements. In such a case, the variable will always be assigned after the branch. The program moves such a variable to the previous level of the stack and continues program processing with the combined environment.

Additionally, the usage of variables is maintained in another data structure. It helps to find the variables that depend on truly global variables and should be considered global as well. For example, in $a = b + c$, if c is global, a depends on c and is global as well; but b does not depend on c or a, and can be both local and global.

Equation 3.1 shows an example of variable assignment processing inside the $if - else$ branch and all data structures. Let's assume a, b, c are potentially global variables. b is used in the expression before assignment, so it is truly global. b is used in a definition, so a is global as well. On the other hand, c is assigned a local value and is not global. Additionally,

a is defined in both branches of if statement, and it will stay in the environment after if processing.

$$
\begin{aligned}
&\textbf{Statement: } \texttt{if (true) \{ a = b \} else \{ c=1; a = c \}}\\
&\textbf{Before processing: } \texttt{Environment: } [\emptyset],\ \texttt{Truly global: } \emptyset,\ \texttt{Usages: } \{\}\\
&\textbf{After processing: } \texttt{Environment: } [\{a\}],\ \texttt{Truly global: } \{b\},\\
&\texttt{Usages: } \{b \to \{a\},\ c \to \{a\}\}\\
&\textbf{Final result: } \texttt{Global variables: } \{a, b\}
\end{aligned}
\tag{3.1}
$$

In summary, the global variable detection procedure combines assignment analysis with scope-aware dependency tracking to distinguish between record-local and truly global variables. As a result, this two-stage approach provides a basis for subsequent parallelizability checks.

3.3 Parallelization rules

This section defines the parallelizability rules for the static analyser. The analyzer traverses AWK program based on the defined rules and detects whether the program can be executed in parallel and how the global variables should be treated.

3.3.1 Patterns

The AWK program consists of multiple pattern-action blocks. The general semantics of each AWK program are shown here:

$$\text{Program} \to \text{Command}^*$$

$$\text{Command} \to \text{Pattern } \{ \text{ Action } \}$$

$$\text{Pattern} \to \texttt{/regex/} \mid \texttt{expr} \mid pattern_1, pattern_2$$

Patterns define the conditions under which corresponding actions are executed for each input record. For every line of input, each pattern is evaluated. The corresponding action is executed only if the pattern evaluates to true. Patterns can take several forms, such as regular expressions that match the current record, boolean expressions involving variables and fields, or range patterns that span multiple records

Patterns are defined to be parallelizable only if they are not based on global variables. Regex-based patterns are parallelizable by default as they do not use any global info. Patterns as expressions cannot be correctly executed in parallel if they rely on a global variable value, because the global variable value is correct only after aggregation and is separate for each thread during main program execution in parallel. The program is marked as non-parallelizable if the global variable appears in any pattern or condition.

The special case is comma-separated range patterns; such patterns are defined as not parallelizable by default. These patterns are true if the first pattern is already true and the

second pattern is not yet false. Such a scenario implies that we know about the state of this condition from the previous line, which makes it an additional global knowledge. This makes comma-separated patterns hardly parallelizable, and static analyzer marks them as non-parallelizable.

3.3.2 Actions

In AWK, an action is a sequence of statements that is executed for each input record whose associated pattern evaluates to true. Actions define the main behavior of the program and are responsible for updating variables, producing output, and controlling the execution flow. Their structure is central to determining whether the program can be safely parallelized. The statement grammar (only the most popular forms mentioned) is defined here:

```
Statement ::= expression
            | print expression_list
            | printf ( format, expression_list )
            | if ( expression ) statement
            | if ( expression ) statement else statement
            | while ( expression ) statement
            | for ( expression ; expression ; expression ) statement
            | for ( variable in array ) statement
            | do statement while ( expression )
            | break
            | continue
            | next
            | exit
            | exit expression
```

The simplest statement is an expression, and this is also the most important statement for parallelizability analysis. Among expressions, those that modify program state are of particular importance, since they can introduce dependencies between input records. In AWK, this primarily occurs through assignments to global variables, whose values persist across records.

Not all such assignments prevent safe parallel execution. A key subset consists of updates where the new value of a global variable depends only on its previous value and on record-local information. These updates can often be interpreted as reductions, where each record independently contributes to an accumulated result.

As an example, parallelizable are assignments in the form of:

$$var = var \oplus const$$

Here $\oplus$ is one of the operators that will also define the final aggregator. *const* indicates that the other part of the expression is the expression on constants and local variables. The assignment operations (e.g $+=$) can be treated in a similar way, as well as the increment operations ($++$ and $--$).

Operation	**Canonical form**	**Reduction Operation**	**Commutativity**
$+$	$v = v + e$	$+$	✓
$-$	$v = v + (-e)$	$+$	×
$*$	$v = v * e$	$*$	✓
$/$	$v = v * (1/e)$	$*$	×
&&	$v = v \&\& e$	logical AND	✓
||	$v = v \|\| e$	logical OR	✓
string concatenation	v = v e	concat	×

Table 3.1: Candidate reduction patterns for aggregation
v denotes the affected global variable, and e is any expression over local values.

Table 3.1 shows the operations that can be parallelized using one of the reduction operations. The reduction operation is the operation that will be applied to the thread-local results of the same variable from different parallel execution workers after the parallel computation is finished. The reduction should produce the variable values that are the same as after the sequential execution, such that global values can be used afterwards in *END* statement of the script. The candidate reduction operation should remain valid for the full parallelized part of the program. (e.g, if there are two assignments for the same variable with different candidate reduction operations, the pattern becomes not parallelizable).

Additionally, commutativity of the operation is an important factor. An operation is commutative if changing the order of its operands does not affect the result, such as in addition ($a + b = b + a$). In contrast, non-commutative operations produce different results when the operand order changes, as with subtraction ($a - b \neq b - a$). The result of the non-commutative operation will differ if the parameter order is changed. Due to that, the non-commutative operation is parallelizable only if the global variable under assignment is mentioned only on the left side of the operation, as it can be reduced to one of the canonical forms in Table 3.1.

For example, $v = v - 1$ is parallelizable, as it can be rewritten as $v = v + (-1)$ and reduced using the addition operation. On the other hand, $v = 1 - v$ can only be rewritten into $v = (v * (-1)) + 1$, which is a mix of two different reduction operations and can not be parallelized.

Another simple form of potentially parallelizable assignment is:

$$var = const$$

Here, a global variable is assigned a constant or local value, and it does not depend on the previous value of the variable. The reduction operation for such an assignment would take

the last value for the variable for the partition where it was actually assigned any value. It will be a default value if it is not assigned by any of the workers. For this to work, we should keep the order of the partitions and assign the chunks of data according to that order.

3.3.3 Conditionals

Conditional statements such as `if`, `while`, `do-while`, and `for` loops introduce additional challenges for parallelization, especially when their condition expressions depend on global variables. In a sequential AWK program, global variables are changed as each line is processed, and decisions about which branch to take are based on a continuously evolving program state.

In a parallel execution model, each worker thread processes a subset of input lines with its own local copy of global variable values. As a consequence, conditionals evaluated during the mapping stage may take different branches than they would in a sequential execution, since the global state is incomplete and thread-local. This divergence can affect both control flow and side effects such as variable updates and output generation.

For example, consider a condition such as $if(count > 10)$, where *count* is a global variable updated as records are processed. In sequential execution, it reflects the cumulative result of all previously processed records. In parallel execution, each thread only observes its own partial count. This means one thread may enter the branch, while it would not happen in the sequential mode.

Because of these risks, conditionals that reference global variables cannot generally be guaranteed to preserve correctness under parallel execution. For this reason, the static analyzer conservatively disallows parallelization if any conditional expression references a global variable. Conditionals that depend exclusively on constants, field variables, or non-global variables are considered safe. This ensures that the semantics of the original AWK program are preserved.

3.3.4 Non-parallelizable side-effects

There are several AWK constructs whose semantics inherently depend on sequential execution. This project defines such constructs as non-parallelizable and treats their presence as a hard constraint preventing parallel execution of the program.

The `print` and `printf` statements in AWK produce observable side effects in the form of console output. While printing expressions that contain only constants, field variables, or local (per-line) variables will produce the correct output regardless of execution order. The output that contains global variable values cannot be parallelized. Due to this, the analyzer marks the program as non-parallelizable if any `print` or `printf` statement references a global variable.

Another aspect that could potentially impact parallelizability is built-in AWK variables. Built-in variables can change the AWK default function behavior, as well as contain some global information that is kept throughout the program.

The built-in variables `NR` and `FNR` represent the total number of records processed so far and the number of records processed in the current file, respectively. Both variables are inherently sequential, as their values depend on the complete ordering of input records.

In a parallel execution model, each worker processes only a subset of input lines and therefore cannot observe the correct global values of `NR` and `FNR` during the mapping stage. The final values could be reconstructed during reduction stage by applying addition on the partial results, but any use of `NR` or `FNR` in the main part of the program would observe incorrect intermediate values and lead to behavior that diverges from sequential execution. The static parallelizability analyzer maps any AWK program as non-parallelizable if it references these built-in variables in the main part of the program.

Other AWK built-in variables (such as `NF`, `FS`, `RS`, `OFS`, `ORS`, and similar) are by default constant. Therefore, they can be safely accessed in the main part of the program. However, reassignments to built-in variables in a parallel setting may occur independently in multiple threads. This can lead to an inconsistent state and non-deterministic behavior, since built-in variables can affect record parsing, field splitting, output formatting or produce other execution side-effects. Due to this, the analyzer conservatively disallows parallelization if any built-in variable is assigned a new value in the main part of the program.

3.4 Associative array parallelization

AWK supports associative arrays, where array indices may be strings, numbers, or expressions evaluated at runtime. The associative array in AWK is a one-dimensional key-value array, where the value can be accessed by providing the associated key. While associative arrays are a powerful language feature, they introduce additional challenges for safe automatic parallelization. This project distinguishes two cases for array usage based on the structure and types of array indices to distinguish and safely parallelize AWK programs with arrays.

Each array element can be treated as an independent variable for parallelization analysis if all indices of an array, used throughout the program, are simple constants, and all index constants are of the same type (e.g., all string literals or all numeric literals). In this case, access to a specific array element is treated equivalently to a scalar global variable access, and the same aggregation rules described for scalar variables apply independently to each index.

On the other hand, if the used index values of the same array are of different types (e.g, string and integer), they can not be safely treated as different index values. The AWK dynamically casts all non-string values to strings during program execution; due to this, the indexes of the two types can represent the same index in the end.

If at least one array index is not a simple constant (e.g., a variable or expression), or if indices of different types are used for the same array, the analyzer conservatively treats the entire associative array as a single global variable.

This situation arises because different threads may access or update overlapping or dynamically determined indices, making it impossible to statically guarantee independence between different array elements. Treating the full array as one variable preserves correctness at the cost of reduced parallelization opportunities.

When an associative array is treated as a single global variable, assignments to array elements are parallelizable only under strict conditions. Specifically, an assignment of the form

$$arr[i] = arr[i] \oplus e \quad or \quad arr[i] = e \oplus arr[i]$$

is allowed only if the index expression `i` is syntactically identical on both the left-hand side and within the right-hand side expression. This restriction ensures that each parallel worker updates the same logical element of the array and that the operation can be safely reduced using the detected aggregation operator. Additionally, the variables that are part of index `i` should not be changed as part of expression `e`. That is because it can potentially represent a different index for assignment in case of change.

Assignments where the index expression differs, or where the array is accessed with multiple distinct index expressions within the same assignment, are marked as non-parallelizable. Additionally, access to the index, where the index is represented by the other global variable, is not parallelizable.

3.5 Function parallelization

The AWK language provides a possibility to call functions as part of the script. The function can be custom - defined in the script, or one of the rich set of built-in AWK functions. Functions in AWK allow programmers to reuse the common logic and structure complex data transformations. Although programs written in AWK language are typically much less function-oriented than in some general-purpose programming languages, functions are still an important structural component of AWK programs.

Function calls complicate the reasoning about side effects and dependencies in the context of parallelization. Function bodies can contain assignments to global variables, further function calls, and other statements with potential side effects impacting parallelization. As a result, the parallelizability of a program with function calls requires the analysis that accounts for the behavior of invoked functions.

The side-effect-free functions or pure functions can be relatively easily parallelized. Such functions do not change anything outside the scope of the function, meaning that they can not modify global state from inside in any way. Pure functional programming languages, such as Haskell, with hard constraints on side effects, can be heavily optimized by introducing parallel execution of the functions. Paper by S. Marlow et al. [24] describes the implementation of a parallel execution environment for Haskell that helped to speed up the performance by up to 6 times. There were also attempts to define pure functions in other programming languages for parallelization. For example, the paper by T. Süß et al. [32] introduces special pure functions in C language and optimizes the compiler for parallel execution when such functions are used in loops.

In AWK, the situation is complicated due to the language's specifics, such as the default global scope of the variables. A function can implicitly read or modify global variables, which can break the independence between records required for safe parallelization.

Furthermore, function arguments introduce the local scope, allowing some variables to exist independently of the global program state. In AWK, scalar arguments are passed

by value, meaning that the value of the variable is copied when the function is invoked. Consequently, the operations within the function on the provided arguments do not impact the global state.

On the other hand, the associative arrays are passed by reference to the function. This means that the function operates directly on the data structure, instead of copying the values. Consequently, any change to the elements of the array, passed as an argument, impacts the array state in the caller's scope. From a parallelization perspective, this is important: functions that access or update associative arrays cannot be safely executed in parallel without additional dependence analysis.

The functions in AWK can be split into built-in functions and custom user functions. Built-in function purity can be pre-defined and does not need to be analyzed during script execution; in comparison, the custom function definitions should be statically analyzed before the parallel execution.

3.5.1 Built-in functions

The AWK language defines a lot of built-in functions that are used to process text input, do mathematical operations, or manipulate program behavior. This study splits the built-in functions into three groups: pure functions (parallelizable), functions with non-parallelizable side effects, and functions with parallelizable side effects.

Pure functions

Pure functions are functions that do not produce any side effects. These functions take the input in the form of a variable or an expression, compute some value based on the argument, and return it as the result. They do not modify global variables, do not update associative arrays, and do not depend on external program state. This group consists of all numerical computations (e.g square and power functions), some string manipulation functions (e.g length and substring functions), and so on. The absence of side effects guarantees the deterministic behavior of pure functions, making them safe to be parallelized.

Non-parallelizable functions

Another group of AWK built-in functions is the functions that produce non-parallelizable side effects. These functions interact with the external environment, modify the execution flow, or affect the global behavior of the program. For example, *srand* function, which sets the seed for the random functions, can not be safely parallelized as it changes the global seed value. After *srand* function execution, the seed value is used by the *rand* functions during the sequential execution, which can lead to unexpected behavior in the parallel mode.

Another important representative of such a non-parallelizable function is the *system* function. While in the classical AWK description this function is described as a separate statement [1], the syntax of it follows exactly the other built-in function pattern. Additionally, in some later versions of AWK, it is treated as a function [28], including the internal *frawk* interpreter logic. This function allows executing arbitrary operating system commands within the AWK script and then returning to AWK code. While this function does not modify the

global state of the AWK program, it can produce the side-effects non determinable by static analysis, so it cannot be safely parallelized.

Functions with parallelizable side-effects

The third group of built-in functions defined in this study is functions with potentially parallelizable side effects. This group mostly consists of the functions that modify one of the provided arguments, producing side effects as a result. Most importantly, the potential side-effect impact on the global program state can be statically analyzed.

The *split* is a good representative of such a function and is commonly used in AWK scripts. This function splits the string using the provided separator and stores the produced values in the array. This function modifies the internal state of the array provided as one of the arguments. Interestingly enough, the function is cleaning up the previous array state, which helps in the function parallelization.

The static parallelization analyzer treats a call to *split* function similarly to the simple array assignment with an unknown index. However, in this case, it is known that the value of the array is fully redefined, and the previous value was lost. Therefore, the static analyzer treats such arrays as a special scenario, where the array can be treated as a local variable, in case it is fully reassigned every time using *split* function before the next use for the same line.

As example, Figure 3.2 shows the parallelizable way of *split* function. Here, the argument array *fields* is fully reassigned before any use and does not retain any values from the execution of previous text lines. The static analyzer can treat *fields* variable here as a local variable and safely parallelize this code snippet. In comparison, Figure 3.3 shows the non-parallelizable version of a similar program. Here, *fields* can contain the value from previous execution cycles if the condition is false for the current text line. Such a variable potentially contains a global value from previous lines and can not be safely parallelized, since it is used in the next *for* loop.

```
1 {
2     split($0, fields, ",")
3     sum = 0
4     for (i in fields)
5         sum += fields[i]
6     print sum
7 }
```

Figure 3.2: Parallelizable use of *split* in AWK

Another similar to *split* functions in terms of parallelization is *sub* and *gsub* functions. These functions help to match occurrences of a regular expression in the target string and replace them with the provided value. The *gsub* function replaces all occurrences of the regular expression, while *sub* function only replaces the first one. These string-manipulation functions are very common in the main part of AWK scripts, as they enable powerful operations on the text input. Most occurrences of these functions operate solely within

```
{
    if ($1 == 1) {
        split($0, fields, ",")
    }
    sum = 0
    for (i in fields)
        sum += fields[i]
    print sum
}
```

Figure 3.3: Non-parallelizable use of *split* in AWK

local scope and avoid introducing global state. Therefore, including these functions in static analysis is important for effective parallelization.

Functions *sub* and *gsub* should be treated separately due to their unique argument handling. As mentioned earlier, AWK functions use pass-by-value argument passing for general variables. In contrast, these two functions treat the target argument (textual input) as a reference to the global variable and change the value of the variable in place. Consequently, the operation introduces a state change of the variable whose aggregation behavior cannot be reliably predicted under parallel execution. For this reason, safe parallelization is only possible when the provided target argument refers exclusively to a local value rather than impacting global state.

The Figure 3.4 shows an example of parallelizable *sub* function use. In this case, *cleaned* variable contains a local value, and therefore operations on it, such as *sub* function, are safe from a parallelization perspective. On the other hand, the variable *prefix* in Figure 3.5 contains a global value. It can not be safely parallelized and afterwards aggregated predictively before the END statement execution.

```
{
    cleaned = $0
    sub(/a/, "b", cleaned)
    print cleaned
}
```

Figure 3.4: Parallelizable use of *sub* in AWK

```
{
    cleaned = cleaned $0
    sub(/a/, "b", cleaned)
    print cleaned
}
```

Figure 3.5: Non-parallelizable use of *sub* in AWK

By analyzing these built-in functions, the parallelization framework can accurately identify operations that are local to a record versus those that modify shared or global state. This distinction is crucial because functions like *split* or *sub* can either safely operate in parallel on per-record data or, if applied to shared variables, introduce order-dependent results and fail parallelization.

The built-in functions are extensively used in AWK scripts. Built-in functions are a powerful tool to manipulate input, do numerical computations, and control the general program execution flow. The inclusion of the most popular built-in functions in the static parallelization analyzer helped to significantly boost the initial power of the parallelized AWK version developed in this study.

3.5.2 User custom functions

AWK allows users to define their own custom functions, which extend the language capabilities. These functions can include commonly used computations, text transformations, or control logic, making programs more modular and readable. Unlike some built-in functions, user-defined functions strictly follow pass-by-value semantics for ordinary variables.

The custom functions introduce the scope of variables for function parameters. In general, all the variables in AWK can be accessed globally; however, the parameters within the function create a function scope, which isolates changes to those variables from the global context. This means that modifications to function parameters do not affect the rest of the program, unless explicitly written to a global variable. Consequently, understanding the scope behavior of custom functions is important for functions' parallelization analysis, as functions that operate solely on their local parameters can be executed concurrently across records without introducing any behavior changes. For example, Figure 3.6 shows an example where a change to the parameter does not impact the global state of the program. The global function *count* is shadowed by the parameter inside the function scope, and the value of the global function stays constant, while the parameter value is changed.

```
1  BEGIN { count = 10 }
2
3  function custom_increment(count) {
4      count += 1      # modifies parameter value
5      return count
6  }
7
8  {
9      local_count = custom_increment(count)
10     print local_count        # will print 11
11     print count              # will print 10
12 }
```

Figure 3.6: Custom AWK function with a parameter shadowing a global variable

The program in Figure 3.6 can be safely parallelized, since both variable values *count* and *local_count* are not impacted by the information from previous record execution. *count*

value stays constant throughout the program, and local_count value is reassigned a new value during each record execution.

During the analysis of real AWK programs (dataset description and extraction methods are described in Chapter 5), it was found that custom function usage inside the main part of the program in a parallelizable way is rare. Only 32 programs out of 970 were following the criteria for potential parallelization:

- The program calls a custom function in the main part of the program
- No supplied to function arguments are global variable values
- The program is not failing in parallelization during the previous analysis

The implementation of a static analyzer for custom functions could potentially parallelize only up to 3% of the real-world programs, in case all the matching programs were successfully parallelizable. Instead, the custom functions were often used in the sequentially executed *BEGIN* and *END* statements.

Due to the small potential for parallelizability results improvement, it was decided not to include function analysis in the current version of the static parallelizability analyzer.

The analyzer can be modernized in the future to include custom function analysis as well. This will require analyzing function bodies in a similar way to the main program syntax. Additionally, the introduced function parameters can be treated as local variables during the function body analysis.

Chapter 4

Integration with Interpreter

This section dives deeper into the implementation details to integrate the parallelization static analyzer with AWK interpreter. The interpreter chosen for integration is *frawk*, due to its modern syntax and initial support for some basic parallel execution. This section discusses the parallel execution flow, including input parsing, output production, variable preparation, and variable aggregation.

4.1 System architecture overview

The overall parallel execution pipeline of AWK script is illustrated in Figure 4.1. Execution begins with the sequential execution of the *BEGIN* block, which initializes the program state. Additionally, the input data is prepared for processing by partitioning it into independent chunks. Each worker will be assigned a separate chunk of data during the parallel execution. Additionally, global variables are prepared for the parallel execution. Thread-local copies of the variables are prepared for independent use by workers during parallel execution.

The main part of the AWK program is then executed across multiple worker threads, each processing a separate partition of the input. During parallel execution, each worker operates on thread-local copies of global variables. After processing is complete, these intermediate results are combined in a reduction phase using aggregation strategies. Finally, the aggregated global state is passed to the *END* block, which is executed sequentially to produce the final output.

The static analyzer determines whether a given AWK program satisfies the conditions required for safe parallel execution. If the program is not parallelizable, the interpreter reverts to the standard sequential execution model. This design cleanly separates parallelization analysis from actual script execution, enabling transparent parallelization with minimal adjustments to the actual AWK interpreter logic.

4.2 Input parsing

Initially, *frawk* interpreter had two parallel input processing modes. The first mode (per record) was parallelizing based on data chunks. The interpreter created multiple file readers

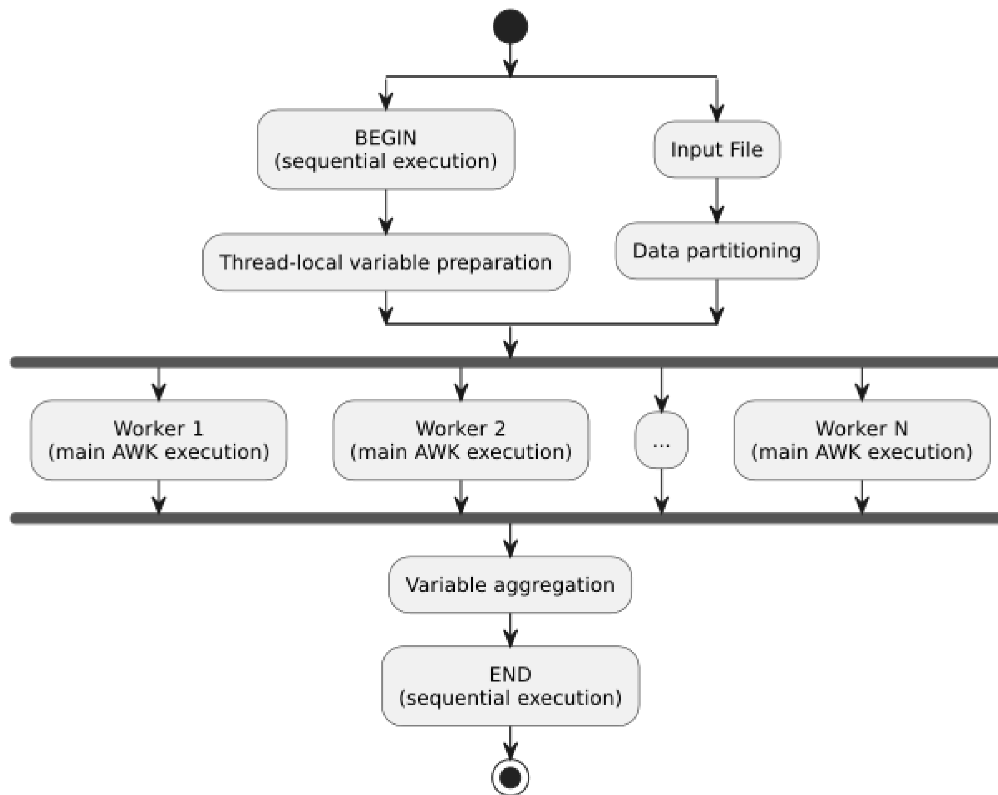

Figure 4.1: Parallel AWK execution flow model

that were reading small chunks of data sequentially. The readers were providing chunks of data to one of the available workers. The workers were processing consumed input in parallel afterwards. The second mode (per file) was parallelization based on files. A separate reader was created per file, and each worker was assigned a file to read.

Both parallel input processing modes were not fully suitable for integration with the analyzer for parallelized execution. It was not possible to keep the order of data processing with the first mode of operation, since the read data chunks were randomly assigned to available workers. The order is essential here for a correct variable aggregation with non-commutative operations. On the other hand, the second mode of operation was only usable for multiple data files. The program was executed sequentially in case of a single large file.

To support parallel execution within the *frawk* interpreter, the input processing system needed to be extended so that different worker threads could independently process disjoint portions of the input file. This project introduces a third input reading mode that preserves input order, as well as executes in parallel with a single-file input. This mode of operation splits the file into approximately equal parts and assigns each worker to an individual part of the file.

The core idea of such an input processor is to treat an input file as a collection of

independently readable "slices." Each slice corresponds to a contiguous byte range of the file, adjusted to start and end at record boundaries to preserve AWK's record-oriented semantics. The interpreter constructs a dedicated chunk producer for each slice of data, which is used internally by *frawk* to read input and feed records to the AWK program. Each worker is assigned to a specific chunk producer and processes the data from it. This helps to keep the deterministic order of workers and aggregate the results in the correct order after script execution.

This mode of input processing required a small file pre-processing before the actual reading. The file is opened once before actual reading, and the approximate byte ranges are adjusted to stop on the actual line separator. This pre-processing helps to avoid lines on the border of slices being skipped, read partly, or read multiple times by different workers. This is done by moving the file pointer to the mentioned border position in the file and reading a small buffer from the file until it contains a line separator symbol. The data slices are adjusted based on the found line separator position accordingly afterwards.

Additionally, the lazy-initialization pattern ensures that each worker only pays the input processor initialization cost when it actually begins reading. It also makes the multiple-processor file reading initialization parallelized. Each processor then behaves identically from the interpreter's perspective; it exposes the same interface to the worker and can supply input records independently of the others.

This design cleanly integrates with the existing interpreter architecture while enabling scalable, data-parallel execution of AWK programs. It encapsulates input slicing behind a uniform API, allowing the rest of the pipeline to remain unchanged.

4.3 Output Ordering

Output operations in AWK, such as *print* and *printf*, introduce side effects, as they write data to the standard output stream or to an output file. In a sequential execution model, records are processed one after another, and output is produced in a deterministic order that corresponds to the input processing order. However, in the parallel execution mode, allowing each worker to emit output immediately would break the initial sequential order. The resulting output would become interleaved and would not follow the sequential order if produced directly during processing, since workers execute concurrently and process data independently.

To preserve AWK's sequential semantics, this project introduces an additional output buffering mechanism within the interpreter. Instead of writing output directly during parallel execution, each worker accumulates its generated output in a dedicated in-memory buffer. All output operations performed while processing a worker's assigned input slice are redirected to this buffer rather than to the actual output destination.

Once parallel execution has finished, the interpreter enters an output production stage. The buffers are processed in the same deterministic order as the corresponding input partitions, starting from the worker responsible for the first slice of the file and continuing sequentially to the last worker. The contents of each buffer are flushed to the standard output stream or to

the target output file. As a result, the final output is identical to the output that would have been produced by a purely sequential execution of the program.

This approach provides deterministic output generation without requiring synchronization between workers during record processing. By postponing output operations until the completion of the parallel phase, the design avoids contention on shared output streams and preserves the exact ordering guarantees expected by AWK programs.

The buffering mechanism requires minimal modifications to the existing output infrastructure. The *frawk* interpreter already relies on a dynamic buffering system for handling large strings and output generation. In the original sequential execution model, this buffer is typically flushed immediately after each output operation. While this behavior is suitable for correctness, it can be inefficient for programs that perform a large number of small-sized output operations, as it may result in frequent write calls to the underlying output stream. In the parallel execution model introduced by this project, each worker retains its output in the existing dynamic buffer throughout the entire parallel processing phase. The buffer is only flushed during the final aggregation stage, when worker outputs are flushed in the correct order.

A consequence of this approach is that the total amount of output produced during the parallel phase must fit into available memory, since all worker outputs are retained in memory until execution completes. This introduces a limitation for workloads that generate extremely large amounts of output. One possible solution to overcome it would be to extend the buffering system with support for temporary files for buffer storage, allowing worker buffers to be moved to disk once they exceed a predefined memory threshold. Such a mechanism would remove the memory limitation and enable processing of arbitrarily large outputs. However, disk-based buffering introduces potential I/O overhead and may significantly affect performance for output-intensive workloads. Designing an efficient hybrid memory-and-disk buffering strategy, therefore, requires further investigation and can be done as part of future work.

4.4 Variable preparation and aggregation

Parallel execution in *frawk* relies on a slot-based mechanism for transferring variable values between worker threads. Each AWK variable is assigned a dedicated slot, and values are written into these slots when a worker finishes processing one stage of execution, then restored when another worker begins its next stage. This provides a uniform interface for synchronizing interpreter state across workers.

To preserve AWK's sequential semantics during parallel execution, variables must be handled carefully at two specific points in the execution flow: after the *BEGIN* block is executed and before the *END* block. These two points correspond to the mapping phase, when information is sent to multiple workers, and the reduction phase, when the information from multiple workers should be correctly aggregated.

This project adjusts the value of slots pre-determined for specific global variables based on the found aggregator operation. After the begin block, the slot values should be reset to their defaults for certain operations to avoid duplication, and the slot states should be

saved for future value aggregation. The value of variables is changed to 0 or an empty string (equivalent in AWK's dynamically typed environment) if the operation is addition, to 1 in case of multiplication, empty string in case of string concatenation. In the case of the last assignment aggregator, the value is changed to a special undefined value, which will be described in more detail in the next section. The variable value remains unchanged for the *and* and *or* aggregators.

The aggregation of multiple worker variables is then fairly straightforward. The aggregation is done after data processing is finished. The aggregation is done starting from the worker that was processing the last part of the file and going upward to the worker with the first part of the file. This order is crucial to preserve the correctness of the result for the non-commutative operations. The aggregators for addition, multiplication, and string concatenation are just the operations themselves. The aggregator for the last assignment operator is based on the previously mentioned undefined values; the value is changed to the first value from the worker where it is defined.

The aggregation for the operations *and* and *or* is special due to the behavior of AWK for these operations. There is no typical boolean type in AWK. Instead, any value except the default ones is treated as 1. Zero and an empty string are treated as 0. In this case, any boolean operation on any values will return 0 or 1. Due to this, aggregation for the boolean operation checks for the actual values of workers.

The logic for *and* aggregation is: the value will be 0 if any of the values is 0 or an empty string, the value will be 1 if that is not the case and any of the values is 1. If these two conditions are not true, it means that no workers have actually performed the operation on this variable inside the main loop. It also means all workers' values for this variable should be the same (the value assigned during the BEGIN stage), and we can assign the value of any worker as the final value.

The assignment logic for *or* aggregator is similar. The final value will be 1 if any of the values is 1. The value will be 0 if any of the values is 0 or an empty string. If neither condition is true, then all values will be equal to an initial value, and the final value should also be equal to it.

4.5 Unassigned values

A key challenge in aggregating variables after parallel execution is correctly handling assignments that occur only in some workers. In sequential AWK, a variable that is assigned within the main block simply takes its final value from the last executed statement. In the parallel setting, however, workers operate on disjoint slices of input, and not all workers necessarily assign to every variable. To preserve sequential semantics, the aggregator must be able to determine which worker performed the last meaningful assignment to a given variable.

This project defines a special "unassigned" value for this purpose. These values are used instead of relying on default AWK values (0 and an empty string) to avoid the collisions of unassigned values and the variable that was specifically assigned a default value.

The special "unassigned" value is defined for every variable type. The AWK variables can take the form of a number or a string. The type is dynamically adjusted based on the operation it is used in. The interpreter internally stores values of the variables as a custom string type, 64-bit integer, or 64-bit float. The values are afterwards dynamically converted to the required type based on the operation they are used in.

Introducing a special wrapper type to mark unassigned values was not feasible, since *frawk* internally uses compact data structures with tight memory usage control for performance efficiency. Many internal interpreter functions assume the values have a fixed size and specific memory layout. Adding even a single bit of overhead would break these assumptions and require widespread refactoring across performance-critical paths, potentially affecting the memory usage and performance of the interpreter.

For this reason, the project explored the actual type values stored in memory to find any unused bit pattern that can be used to describe a not-assigned value. This approach keeps the memory footprint identical to the original implementation, preserving compatibility with the rest of the interpreter, while enabling the correct parallel aggregator functionality.

The string value in *frawk* has multiple representations. The simplest representation of a string is a 128-bit (16 bytes) memory to store a constant string up to 15 one-byte characters. Here, 15 bytes are used for characters, and the last byte is a metadata byte. The metadata byte is split initially into 5 bytes for the length of the string, and 3 bytes are used to encode the string storage type. However, string length encoding can be decreased to 4 bits in this case, since the maximum character length of such a string representation is 15 and can be fit into 4 bits. The last bit in this case is left unused, and this project reuses it to mark unassigned string values. This simple trick avoids the large refactoring of interpreter operations with string variables; additionally, it avoids any possible collision of unassigned value with real data.

The float in *frawk* is represented as a 64-bit float without any wrappers on top of it. The 64-bit float type consists of 1 sign bit, 11 exponent bits, and 52 significand bits, based on the IEEE standards of floating points [19]. Floating-point type also has special values like inf, $-inf$, and NaN. The NaN value is defined as all 1s in exponent bits and at least one 1 in significand bits. This implementation provides some free bits in the significand to encode some information in case of all exponent bits are equal to 1. This project defines an unassigned float value as a value with all exponent bits set to 1 and significand bits set to 0 except for the least significant bit in the significand. This provides a value that will never collide with any real floating-point data. Even NaN values, possibly created during execution, will have a different internal bit structure.

The integer in *frawk* is represented as a 64-bit integer type. In comparison with other types, there are no special cases for the standard integer implementation when one bit will not impact the actual stored value. An integer value consists of 1 sign bit and 63 value bits. It is not possible to encode an additional unassigned value that will not collide with the other values without changing the format of the type.

Because changing the integer format was not an option, since many parts of the interpreter rely on the exact size, alignment, and binary representation of the integer type, the only feasible solution was to reserve a specific integer value to act as the unassigned. The project selects the lowest possible 64-bit value (INT64 MIN) to correspond to an unassigned value.

It is unlikely that such a value will appear during the result aggregation phase in real-world AWK programs, which typically operate on bounded numeric domains such as counters, IDs, or arithmetic results derived from input text.

While this approach cannot fully guarantee the absence of collisions, it gives a practical balance: it keeps the memory constraints of the interpreter, avoids the large structural changes to value handling, and provides a reliable way to detect when a variable was never assigned within a given worker during the aggregation phase. In practice, the probability of a program legitimately producing the reserved integer is negligible, making this technique robust for the intended use case and lightweight at the same time.

4.6 Summary on the implementation

The integration of the static analyzer into the *frawk* interpreter required modifications at several levels of the execution pipeline. A new ordered input-processing mode was introduced to enable deterministic partitioning of single-file workloads, ensuring that parallel workers could process disjoint slices of data while preserving record boundaries and execution order. The output in-memory buffering and ordered production were introduced to maintain a sequential-like output behavior.

In addition, the interpreter's slot-based variable transfer mechanism was extended to support analyzer-driven preparation and aggregation of global variables. This enabled parallel execution to preserve sequential AWK semantics across a range of aggregation operations. To support correct reduction in cases of partial assignments, specialized unassigned-value encodings were introduced for all supported internal types while maintaining the interpreter's original memory layout.

Together, these changes provide the runtime foundation required for semantics-preserving parallel execution of AWK programs identified as parallelizable by the static analyzer.

Chapter 5

Evaluation

The goal of the evaluation is to determine the practical usefulness of the proposed automatic parallelization approach. The implementation of the parallelization analyzer will be tested on two aspects.

Firstly, the project performs an applicability test to understand if real-world AWK programs can be parallelized using the defined above rules. The large dataset of publicly available AWK scripts was collected to do this. Each program was processed by the static analyzer, and the number of safely parallelizable programs was detected. This experiment helped to estimate the fraction of existing programs in AWK that can benefit from parallel execution without any modification to the program required.

Secondly, the benchmarking tests were performed. For this, we selected sample parallelizable programs and executed them on large datasets with both sequential and parallel execution modes to test the performance in both cases. The performance was also compared to other widely used and classical AWK versions. The objective of this part is to quantify the speedup achievable by the parallelization and to show possible future improvements on the current implementation.

5.1 Applicability analysis

5.1.1 Data extraction

The data extraction principles described in this section were inspired by the paper of D. Spinellis et al. [31]. Authors extract the repositories that follow the pre-defined set of requirements using GitHub APIs [1] and build the testing dataset based on it. One of the most important criteria defined in the paper for repository extraction is more than 10 stars or forks, which helps to filter out all personal projects and student exercises. This filter leaves only the programs important to the community. Unfortunately, the mentioned previously GitHub APIs does not support filters on stars or forks when specific files are searched; it is supported only on a repository search level, so this project moved to a different source of information.

[1] Available at: `https://docs.github.com/` (Accessed: 2026-06-01)

This project uses an open-source GitHub dataset published on Google Cloud [16] in 2016. This is officially extracted GitHub data that contains information about approximately 2 billion files. This project uses the prepared repository table that contains the top 400.000 repositories with at least 2 stars received in the first five months of 2016, as well as the table with all file information.

These tables helped to extract the list of existing publicly available AWK scripts that will be used to evaluate the parallelizer's applicability. The research uses the following filters on the tables:

- The file should end with *.awk*. This helped to search only for the files written in AWK language and do not extract any other files.
- The repository should have at least 10 star events during the mentioned period of time. These criteria repeat the idea discussed in the paper by D. Spinellis et al. [31] to only use the popular enough repositories for evaluation.

The query was constructed to satisfy the following criteria and applied to the dataset. The results were 1061 distinct files from 303 repositories. After the GitHub API query to extract the full code of the files it resulted in 970 files from 278 repositories. The other 91 files could not be extracted, possibly due to them not being located on the main branch of the repository or due to their deletion or archiving. All correctly extracted files were used in the following applicability analysis.

5.1.2 Parallelizability evaluation

The extracted AWK file dataset was used to evaluate the applicability of the static parallelization analyzer. For this reason, a special mode of operation was introduced in *frawk* code, such that the supplied program was parsed and statically analyzed, but not executed. This helps to quickly analyze the large dataset with AWK programs.

The programs were split into 4 subgroups during the analysis. Except for parallelizable and non-parallelizable, the project also separates error programs and programs without the main statement. Error programs are those that could not be parsed correctly or contained syntax errors. The programs that did not contain any statements in the main part of the program were also separated, as these are the programs created not for text processing and can be considered outside the scope of the study. There is no code outside of *BEGIN* and *END* statements in these programs. Such programs theoretically can be counted towards a parallelizable group, since they will not violate any of the constraints defined for parallel execution; however, there will be no code that will be executed in parallel in this case. Due to these specifics, such programs were separated into a separate group and were not taken into account in the evaluation process.

The evaluation results are presented in Figure 5.1. The 50.5% of the programs were non-parallelizable, 25.6% were not parsed correctly, 19.4% of the programs could be safely parallelized, and 4.5% did not contain any statements in the main part of the program.

The results show that almost half of all the examined programs can not be fully automatically parallelized. This confirms that many real-world AWK scripts rely on sequential

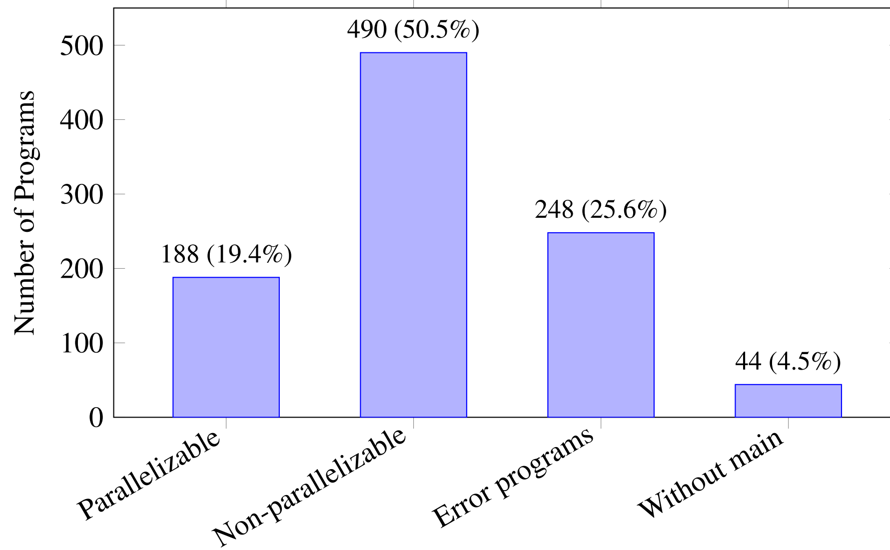

Figure 5.1: Distribution of 970 programs by parallelizability

semantics and wide use of shared global variables. These constructs prevent independent record processing and therefore should still be executed using a classical sequential approach.

Interestingly, another major part of the dataset (25.6%) consists of syntactically invalid programs. This shows the typical nature of public repository code, where scripts may be incomplete or depend on external content. However, the examination of the concrete examples of the code from this category showed that many programs rely on the features of non-classical AWK implementations. Modern versions of AWK have diverse additions and modernizations to the initial classical syntax. GNU AWK (gawk), pre-installed on Linux systems, is the most actively maintained version with the largest amount of features available [18]. For example, the official description of gawk [28] describes a switch case syntax for pattern matching, which is also often used in the extracted AWK programs. However, such syntax is not standard for other versions of AWK, as well as for the *frawk* parser, on which this study relies.

Most importantly, 19.4% of all programs in the dataset can be safely parallelized using the methods described in this study. It represents a substantial number of real-world scripts and demonstrates that the proposed methods are capable of identifying practical opportunities for script parallelization, rather than only working on artificially written examples. The relative part of the parallelizable programs grows to 27.7% if all non-executable inputs (error programs and programs without a main processing block) are excluded from the dataset, meaning that roughly every third to fourth program written in AWK can be parallelized.

These results strengthen the practical relevance of the approach, since a substantial portion of AWK programs can be automatically parallelized and executed with multiple workers without modifying the program itself.

5.2 Performance analysis

The study has also performed the benchmarking of the parallel AWK version to quantify the practical benefits of the technique. This section focuses on testing the performance of the proposed static parallelization approach in comparison with other AWK versions. The major goal of this study is to measure the performance improvements, evaluate scalability with respect to input size, and evaluate any potential overheads introduced by the parallel execution.

The performance analysis and benchmarking are based on the previously conducted study [10], which evaluates the execution time and peak memory usage of major AWK implementations, including gawk, mawk, and nawk, using a set of idiomatic benchmark programs. It uses the *cgmemtime* [2] tool to measure the performance and memory usage of each run and calculate the common statistics. The study uses an open-source test dataset of 1.5 million artificially created sales data to test the performance of executed scripts. Additionally, the study performs one warm-up run followed by multiple consecutive measurement runs to reduce initialization bias and obtain a reliable average execution time.

The benchmark suite of this study consists primarily of adapted versions of the original benchmark programs modified for parallel execution. In addition, some new benchmark scripts have been introduced to evaluate the performance characteristics of the implemented parallelization logic, including the overhead associated with parallel task management and result aggregation. This provides a more comprehensive assessment of both the benefits and costs of the proposed approach.

5.2.1 Experimental setup

This study compares the new parallel interpreter with the other popular AWK versions. The study selects 3 most widely used AWK versions, identified in the paper by B. Hoyt [18]: GNU Awk (gawk) [28], the GNU implementation that extends the original AWK language with a lot of modern features and is commonly pre-installed on Linux systems; mawk (mawk) [5], a performance optimized implementation that is also default for some Linux versions; and "One True AWK" (nawk) [2] [1], which represents a close to classical implementation of AWK language and is pre-installed on some macOS systems. Additionally, GoAWK [17] is used as an example of a more modern AWK interpreter, implemented in Go language. As well as *frawk* is used as the sequential baseline for the parallel interpreter. Table 5.1 shows all the AWK versions used for performance analysis, along with their release dates.

The parallel implementation was tested in two configurations, with 5 and 10 parallel workers, respectively. This setup allows for an analysis of how increasing the degree of parallelism affects overall performance. The study assesses the scalability of the parallel approach and detects whether a higher number of workers leads to proportional performance improvements by comparing execution times and resource usage across these two modes.

All performance tests were conducted on a native Linux machine with 2x 4-core Intel Xeon E5-1410 CPUs. Table 5.2 provides the machine's main specifications.

[2] Available at: `https://github.com/gsauthof/cgmemtime` (Accessed: 2026-06-01)

Name	Short	First Release	Version	Version Release
GNU Awk	gawk	1989	5.2.1	22-11-2022
mawk	mawk	1991	1.3.4	23-01-2024
One True AWK	nawk	1985	20251225	25-12-2025
GoAWK	goawk	2018	1.31	23-12-2025
frawk	frawk	2021	0.4.7	02-01-2023

Table 5.1: AWK implementations used in performance analysis

Metric	Value / Description
CPU Model	Intel(R) Xeon(R) CPU E5-1410
CPU Frequency	2.80 GHz
Amount of CPUs	2
Cores per CPU	4
RAM Size	72 GiB
Operating System	Debian GNU/Linux 12

Table 5.2: Machine and environment specifications for benchmarking

The datasets used in this evaluation are identical to those employed in the referenced study [10]. Specifically, the original dataset consists of 1.5 million pre-generated records representing synthetic sales records, designed to resemble realistic retail or business sales logs. Each record contains typical attributes found in sales data, such as order identifier, product information, quantities, pricing, order dates, region, country, and others. In addition to this baseline dataset, a larger version containing 5 million rows was used to further assess scalability and performance under increased data volume. The datasets were transformed from comma-separated to textual format by replacing the space symbol with an underscore and comma symbols with a space. This made sure all AWK versions support file formats. Both datasets are artificially generated but structured to closely mimic real-world sales records, which ensures consistent benchmarking while providing realistic data characteristics for performance analysis. Both datasets are publicly accessible open-source resources that can be freely used for testing and benchmarking purposes [35].

In addition, this study performs one initial warm-up run followed by five consecutive measurement runs in order to obtain reliable benchmarking results. The warm-up run helps minimize initialization effects and transient system overhead. The repeated measurement runs reduce the influence of external factors on individual executions; additionally, it enables the calculation of average values and corresponding measurement variability across runs, providing a more robust statistical representation of performance.

This study relies on the built-in Unix tool *time* [3] to measure the performance parameters, instead of relying on the external *cgmemtime* tool as described in the referenced benchmark [10]. Built-in tool gives a precise program execution time, peak memory usage, and some other useful benchmarking metrics. Additionally, it is supported and available by default on

[3]Available at: `https://man7.org/linux/man-pages/man1/time.1.html` (Accessed: 2026-06-01)

any Linux machine.

5.2.2 Performance results

This section presents the results of the performance evaluation for the selected AWK implementations. The analysis focuses on execution time and memory usage across different datasets and configurations. The results are structured to enable a direct comparison between implementations for each benchmark script and to assess scalability when processing larger data volumes.

This study aims to benchmark programs comparable to those used in the referenced study [10] in order to maintain methodological consistency. However, many of the original benchmark scripts were written in a non-parallelizable manner, limiting their suitability for evaluating parallel execution models. To address this, the programs were carefully adapted to enable parallel processing while preserving their original functionality, logic, and overall structure as closely as possible. This approach ensures that performance comparisons remain meaningful while allowing the impact of parallelization to be systematically assessed.

The first benchmark script, shown in Figure 5.2, counts the number of orders in Europe. Each input line contains the continent in the first field. The script increments a counter for each occurrence of a European order. It prints the aggregated result as the output.

```
1 $1 == "Europe" { i++ }
2 END {
3     print i
4 }
```

Figure 5.2: Benchmarking script (Counter): Count orders from Europe

The performance of different AWK implementations on the first script is summarized in Table 5.3. A clear trade-off between execution time and memory usage across the tested implementations can be observed in the results. Among sequential interpreters, *gawk* is the fastest (0.68s execution time on the small dataset), with *frawk* and *mawk* close behind, while *goawk* and *nawk* are substantially slower on both dataset sizes. All traditional interpreters maintain very low peak memory footprints in the range of roughly 2–4 MB, *goawk* and *frawk* use more memory at peak.

In contrast, parallel AWK achieves markedly lower execution times. Parallel AWK with 10 workers provides the best performance on both datasets (0.22s execution time on the small dataset and 0.65s on the large dataset). This corresponds to approximately 4-times speedup over sequential *frawk* implementation and more than 3-times speedup over the most efficient sequential AWK version.

The second benchmark script, shown in Figure 5.3, performs the same operation, but it counts the number of orders by country in Europe. Each input line contains the continent in the first field and the country in the second. The script increments a counter for each occurrence of a European country and, in the END block, iterates over the collected entries to calculate the total number of European orders. This script is similar to the first one, but it helps to benchmark operations with the array.

AWK Version	Execution Time (s)		Peak Memory (MB)	
	Small Dataset	Large Dataset	Small Dataset	Large Dataset
gawk	0.68 ± 0.00	2.36 ± 0.01	3.98 ± 0.02	3.96 ± 0.07
nawk	3.38 ± 0.07	11.24 ± 0.32	2.21 ± 0.03	2.18 ± 0.02
mawk	0.91 ± 0.01	2.96 ± 0.02	2.30 ± 0.07	2.31 ± 0.05
goawk	1.36 ± 0.00	4.50 ± 0.01	9.20 ± 0.09	9.40 ± 0.14
frawk	0.82 ± 0.02	2.64 ± 0.14	12.26 ± 0.02	12.29 ± 0.09
Parallel AWK (5 workers)	0.27 ± 0.02	0.86 ± 0.04	13.27 ± 0.08	13.28 ± 0.09
Parallel AWK (10 workers)	$\mathbf{0.22 \pm 0.01}$	$\mathbf{0.65 \pm 0.04}$	14.17 ± 0.20	14.12 ± 0.07

Table 5.3: Performance comparison of AWK implementations on the script in Figure 5.2

Table 5.4 shows the benchmarking results for that script. The results are very similar to the first script and show that the array operations are not significantly impacting the performance of the scripts.

```
$1 == "Europe" { eu[$2]++ }
END {
    for (country in eu) { n++; }
    print n
}
```

Figure 5.3: Benchmarking script (ArrayCounter): Count orders from Europe using an array by country

AWK Version	Execution Time (s)		Peak Memory (MB)	
	Small Dataset	Large Dataset	Small Dataset	Large Dataset
gawk	0.72 ± 0.00	2.49 ± 0.01	3.97 ± 0.07	3.99 ± 0.02
nawk	3.47 ± 0.08	11.32 ± 0.19	2.16 ± 0.05	2.19 ± 0.03
mawk	0.91 ± 0.01	3.06 ± 0.04	2.32 ± 0.03	2.33 ± 0.05
goawk	1.42 ± 0.01	4.70 ± 0.02	9.41 ± 0.12	9.42 ± 0.09
frawk	0.89 ± 0.05	2.94 ± 0.17	12.33 ± 0.10	12.34 ± 0.05
Parallel AWK (5 workers)	0.29 ± 0.02	0.91 ± 0.06	13.91 ± 0.18	13.87 ± 0.14
Parallel AWK (10 workers)	$\mathbf{0.23 \pm 0.01}$	$\mathbf{0.73 \pm 0.05}$	15.40 ± 0.23	15.21 ± 0.21

Table 5.4: Performance comparison of AWK implementations on the script in Figure 5.3

The next benchmarking script is present in Figure 5.4. This script counts the amount of duplicate rows by the first field in the text. The benchmarking results, shown in Table 5.5 is quite similar to the previous two scripts. The parallelized version of AWK achieves a multiple times better performance in comparison with all sequential versions of AWK.

```
{ x[$1]++ }
END {
    res = 0;
    for (val in x) { i += x[val]-1 }
    print i
}
```

Figure 5.4: Benchmarking script (DuplicateFinder): Count duplicate rows by the first field

AWK Version	Execution Time (s)		Peak Memory (MB)	
	Small Dataset	Large Dataset	Small Dataset	Large Dataset
gawk	0.74 ± 0.00	2.55 ± 0.00	3.97 ± 0.03	3.93 ± 0.03
nawk	3.54 ± 0.06	11.73 ± 0.05	2.22 ± 0.02	2.18 ± 0.02
mawk	0.93 ± 0.02	3.09 ± 0.01	2.31 ± 0.06	2.34 ± 0.04
goawk	1.23 ± 0.00	4.08 ± 0.01	9.27 ± 0.12	9.39 ± 0.10
frawk	0.97 ± 0.03	3.27 ± 0.12	12.28 ± 0.06	12.26 ± 0.09
Parallel AWK (5 workers)	0.34 ± 0.02	1.04 ± 0.08	13.30 ± 0.10	13.45 ± 0.11
Parallel AWK (10 workers)	$\mathbf{0.28 \pm 0.01}$	$\mathbf{0.85 \pm 0.04}$	14.36 ± 0.06	14.41 ± 0.06

Table 5.5: Performance comparison of AWK implementations on the script in Figure 5.4

This study includes an example of memory-heavy computation as one of the benchmark programs, since the parallelized versions of *frawk* normally use an increased amount of memory for the computations. Figure 5.5 shows an example of such program. This program is very similar to the previous one, but it stores full row as the array index instead of only the first field. Since there are a lot of different rows in the document, the array size grows rapidly and requires a lot of memory for processing.

```
{x[$0]++}
END {
    res = 0;
    for (val in x) {i += x[val]-1}
    print i
}
```

Figure 5.5: Benchmarking script (MemoryHeavy): Count duplicate rows using an array

The Table 5.6 shows the results for this benchmark. *goawk* performs the best for this script, outperforming all other versions by 2-3 times. Noticeably, parallel AWK versions are slower than sequential implementations and using significantly more memory, growing up to approximately 2 GB of peak memory usage for the large dataset. In the parallel version, each worker is forced to store and operate on a separate copy of the global variable. The large data structures are stored in multiple versions and afterwards aggregated together into one before *END* statement. Duplication of the large array explains both the substantial increase

in peak memory and the slower runtime compared to the sequential configuration.

AWK Version	Execution Time (s)		Peak Memory (MB)	
	Small Dataset	Large Dataset	Small Dataset	Large Dataset
gawk	4.20 ± 0.01	9.15 ± 0.02	565.11 ± 0.02	851.52 ± 0.06
nawk	3.67 ± 0.02	9.38 ± 0.07	278.43 ± 0.05	434.20 ± 0.05
mawk	3.14 ± 0.01	8.16 ± 0.02	290.10 ± 0.03	437.35 ± 0.03
goawk	$\mathbf{1.78 \pm 0.01}$	$\mathbf{4.86 \pm 0.02}$	338.18 ± 6.08	845.64 ± 9.95
frawk	2.82 ± 0.03	7.09 ± 0.12	317.80 ± 0.83	466.89 ± 1.67
Parallel AWK (5 workers)	5.14 ± 0.05	13.04 ± 0.03	664.85 ± 0.12	2114.68 ± 1.23
Parallel AWK (10 workers)	7.77 ± 0.03	19.79 ± 0.06	702.63 ± 2.55	2140.82 ± 1.36

Table 5.6: Performance comparison of AWK implementations on the script in Figure 5.5

Interestingly, the performance of the classical "One True AWK" (nawk) for the more memory-heavy script is better than for the previous lighter script, while for all other versions, it is significantly worse. This shows that field splitting logic is slow for *nawk*, since this script uses the full string ($0) as the array key, instead of the extracted first field ($1). In *nawk* code [2] $0 is directly assigned a value before each line processing; on the other hand, the fields ($1, $2, ...) are parsed only in case one of the field variables is accessed. For all other versions of AWK, field parsing is implemented in a more efficient way, and the script in Figure 5.4 is executed much faster than the script in Figure 5.5.

Another benchmarking script is shown in Figure 5.6. This script counts the number of sales records in the specified range of time. This script uses multiple built-in functions and helps to evaluate the performance of the AWK versions on heavy built-in function usage scripts. The benchmarking results in Table 5.7 show that parallel AWK significantly outperforms standard AWK implementations. The version with 10 workers is 10 times faster than *gawk*, and 3 times faster than *mawk* and the original sequential *frawk* implementation.

```
1 {
2     split($6, a, "/");
3     d = sprintf("%d%02d%02d", a[3], a[1], a[2]);
4     if (d >= "20140301" && d <= "20150331") n++
5 }
6 END {
7     print n
8 }
```

Figure 5.6: Benchmarking script (Functions): Count the number of records during a period of time

Additionally, the output-heavy script was evaluated since the output logic was adjusted in parallel mode. Figure 5.7 shows a simple script to output the first field of each line. The benchmarking results for this script are shown in Table 5.8. The parallel AWK implementations deliver the best execution times in both dataset sizes, with the 10-worker variant being

AWK Version	Execution Time (s)		Peak Memory (MB)	
	Small Dataset	Large Dataset	Small Dataset	Large Dataset
gawk	7.26 ± 0.06	24.56 ± 0.52	3.96 ± 0.02	3.97 ± 0.04
nawk	6.67 ± 0.09	22.22 ± 0.25	2.20 ± 0.05	2.21 ± 0.06
mawk	2.28 ± 0.02	7.64 ± 0.20	2.31 ± 0.06	2.33 ± 0.01
goawk	3.56 ± 0.02	11.91 ± 0.02	9.29 ± 0.06	10.12 ± 1.32
frawk	2.63 ± 0.05	8.70 ± 0.14	12.49 ± 0.17	12.33 ± 0.06
Parallel AWK (5 workers)	0.87 ± 0.08	2.87 ± 0.27	13.36 ± 0.17	13.39 ± 0.08
Parallel AWK (10 workers)	$\mathbf{0.73 \pm 0.05}$	$\mathbf{2.40 \pm 0.13}$	14.46 ± 0.17	14.54 ± 0.13

Table 5.7: Performance comparison of AWK implementations on the script in Figure 5.6

the fastest overall. This speedup comes at the cost of noticeably higher peak memory usage than the sequential interpreters. The higher memory peak in this case occur due to the output buffering used in parallel AWK.

```
{
    print $1
}
```

Figure 5.7: Benchmarking script (Output): Output the first field on each input line

AWK Version	Execution Time (s)		Peak Memory (MB)	
	Small Dataset	Large Dataset	Small Dataset	Large Dataset
gawk	0.69 ± 0.00	2.39 ± 0.03	3.96 ± 0.03	3.98 ± 0.04
nawk	3.39 ± 0.01	11.38 ± 0.19	2.19 ± 0.05	2.19 ± 0.06
mawk	0.94 ± 0.01	3.09 ± 0.03	2.16 ± 0.05	2.22 ± 0.07
goawk	11.33 ± 0.03	37.78 ± 0.14	9.30 ± 0.20	9.25 ± 0.14
frawk	0.87 ± 0.01	2.94 ± 0.08	12.39 ± 0.16	12.43 ± 0.08
Parallel AWK (5 workers)	0.33 ± 0.01	1.17 ± 0.09	49.56 ± 1.67	102.95 ± 4.10
Parallel AWK (10 workers)	$\mathbf{0.27 \pm 0.01}$	$\mathbf{0.98 \pm 0.02}$	49.13 ± 1.57	140.19 ± 7.61

Table 5.8: Performance comparison of AWK implementations on the script in Figure 5.7

Overall, the benchmark suite demonstrates that parallelization in *frawk* consistently yields the lowest execution times. Figure 5.8 shows the achieved performance speedup of the parallel AWK execution over the traditional sequential execution across a variety of workloads. Here the best parallel execution time is shown proportionally to the best sequential time for each script. The speedup is achieved at the cost of a substantially higher memory footprint compared to classical AWK interpreters. Parallelization should be avoided in the case of memory-heavy global data-structure usage in the script, as this will not give the desired performance optimization. The traditional implementations, such as *mawk* and *gawk*, remain

attractive for memory-constrained environments, whereas parallel *frawk* is most beneficial for compute-heavy or throughput-oriented scenarios where additional memory consumption is acceptable. In this case, the parallelization provides large performance improvements on different AWK scripts. Taken together, these results highlight an explicit trade-off between time and space that should guide the choice of AWK engine and configuration depending on dataset size, hardware constraints, and performance objectives.

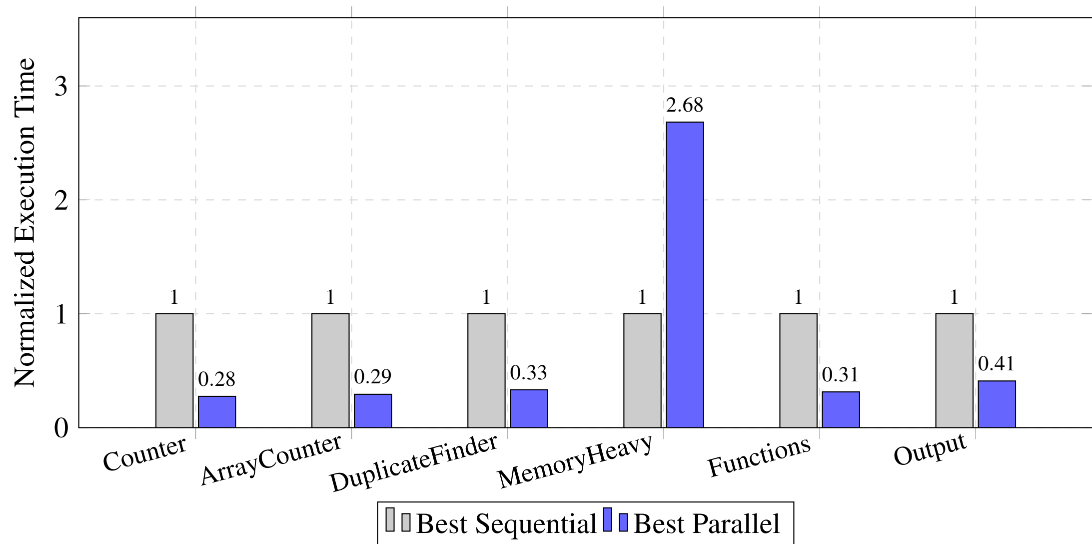

Figure 5.8: Comparison of best-performing sequential and parallel AWK implementations on the large dataset. Sequential execution time is normalized to 1 for each benchmark. Lower values indicate better performance.

Chapter 6

Conclusions and Future Work

This chapter summarizes the main findings and contributions of the thesis, reflecting on the applicability and performance of automatically parallelizing AWK programs. It revisits the key ideas introduced throughout the work and answers the research questions. The chapter also outlines the current limitations of the proposed approach and identifies several directions for future research to improve further the robustness, applicability, and performance of the proposed solution.

6.1 Contributions

This study develops a static-analysis–driven approach to automatically parallelize AWK programs and demonstrates that this approach can be integrated into a practical, production-grade interpreter. The core technical contribution is the static analysis tool that inspects the abstract syntax tree of any AWK script, identifies the relevant global variables, and determines whether the script can be executed in a parallel way without changing its original behavior and results.

A second major contribution is the design of a mapping from AWK global-variable operations to well-defined aggregators. The thesis characterizes any general assignment operation in AWK semantics in terms of multiple well-defined canonical forms and aggregation rules. The system statically identifies when a variable can be aggregated independently on different chunks before actual execution and then combined after execution in a final reduction step. This turns implicit, sequential state updates into explicit, mathematically interpretable reductions, which is a key enabler for sound parallel execution.

On the systems side, the work shows how to embed this analysis into a Rust-based AWK implementation (*frawk*) and extend the interpreter with a parallel execution mode. The interpreter is adapted to partition input into chunks, execute the main part of the program on each chunk in separate workers, and then aggregate the workers' state according to the reduction rules defined during the static analysis step. Importantly, this is achieved without changing the initial AWK program semantics or imposing additional burdens on the user: the same AWK script can be run in a sequential or parallel mode, with the system deciding whether parallel execution is possible.

Finally, the thesis contributes an empirical study that evaluates both the applicability and the performance of the current implementation. The study shows that a non-trivial fraction of real-world AWK scripts satisfy the parallelization criteria by analyzing the AWK scripts from popular open-source repositories available on GitHub. This strengthens the practical relevance of the proposed solution for real-world tasks.

Performance measurements on representative workloads demonstrate that the parallel AWK interpreter can achieve state-of-the-art performance. The benchmark measurements on multiple scripts show up to x2-x3 times performance optimization of the parallelized solution over the fastest sequential AWK execution and over baseline *frawk* interpreter implementation. The system successfully exploits multiple cores to achieve significant execution speedups, especially on larger inputs, while preserving the semantics of the original scripts.

6.2 Research outcomes

The thesis investigated the feasibility of automatically parallelizing AWK programs, as well as the practical applicability and performance benefits of such an approach.

Firstly, the thesis demonstrates a system to automatically detect parallelizable AWK programs by combining static program analysis with controlled runtime execution. By analyzing dependencies related to global variables, control flow, and side effects, the system can determine whether a program can be safely executed in parallel. Programs that satisfy the defined constraints after static analysis can be executed using a MapReduce-style model, where input data is partitioned, processed independently, and later the global state is combined using predefined aggregation operations. The use of correct aggregation strategies and ordered result combination ensures that the final output remains consistent with sequential execution.

Secondly, the study shows that the proposed approach is practical for real AWK scripts by performing an applicability investigation. A study analyses 970 different real-world AWK scripts from popular open-source repositories. On this dataset, 19.4% of all scripts (27.7% of those that parse and have a main block) are found to be safely parallelizable according to the defined rules. It shows that a significant amount of actual AWK scripts could potentially benefit straight ahead from parallel execution. This confirms that the approach is not limited to synthetic examples but is relevant in realistic scenarios.

Finally, the performance evaluation demonstrates that parallel execution can provide significant improvements over traditional sequential AWK execution for most suitable workloads. For parallelizable programs, processing large input datasets using multiple cores leads to noticeable speedups, validating the effectiveness of the approach. The parallelized execution showed the best performance (with x2-x4 times speedup over the fastest sequential AWK versions) in most of the scripts. The performance gains are especially noticeable when the size of the input data increases, confirming that the parallel execution can substantially reduce execution time in data-intensive tasks.

The only case where the parallel execution was slower than the competitors' was a memory-intensive script, in which a large portion of the processed dataset was stored in

an array. This case highlights a scenario where the overhead of creating multiple thread-local copies of large data structures, combined with synchronization costs during the final aggregation phase, can outweigh the computational benefits of parallelism. While such memory-heavy workloads are relatively rare in typical AWK usage, where the language is most often applied to lightweight field extraction and simple aggregations, this limitation should be kept in mind when processing scripts that build large in-memory data structures. For these edge cases, users may need to fall back to sequential execution.

In summary, this work shows that automatic parallelization of AWK programs is both feasible and beneficial: correctness can be preserved through careful analysis and execution design, a meaningful portion of real-world programs can be parallelized, and significant performance improvements can be achieved in practice.

6.3 Limitations

This study demonstrates the feasibility of automatic parallelization in AWK interpreters. It shows noticeable improvements in comparison with sequential AWK execution; however, the current study and the presented AWK interpreter implementation have multiple limitations. This section discusses the most important limitations of the current approach that must be taken into account before further usage of the parallel AWK interpreter.

A key limitation of the current implementation is the reliance on in-memory storage for the worker output buffers during parallel execution. In the current prototype, each worker accumulates its output in memory until the completion of the parallel phase, at which point it is merged in a deterministic order. This approach ensures correct output ordering; however, it introduces a scalability constraint for workloads with very large volumes of output. A possible solution to this limitation would be to introduce a hybrid buffering mechanism that will temporarily store output to disk once a memory threshold is exceeded and retrieve it from disk during the aggregation phase.

Another important consideration is that the current approach of parallel execution is limited only to file input processing. This means that the prototype assumes the program reads data from an input file that can be partitioned into independent chunks before execution, and afterwards, is read in parallel by multiple workers. As a result, the current approach does not support dynamically generated input whose boundaries are not known in advance. This restriction simplifies data chunk production and data distribution between workers for parallel processing; however, it narrows the practical applicability, since many AWK programs are used directly in Unix pipelines rather than on static files.

The evaluation performed as part of this study has several limitations worth mentioning here. The performed applicability study captures a large amount of real-world AWK programs; however, it is limited only to the scripts extracted into separate files with *.awk* extension. In practice, AWK scripts are often embedded in Bash scripts or other source files. As a result, the measured parallelizability rates could differ if embedded AWK scripts were included in the analysis.

Additionally, many programs were excluded from the applicability evaluation because of the non-classical AWK features. These programs could not be compiled with *frawk* and could

not be evaluated. These scripts are developed mostly for interpretation using *gawk*, which has many additional features on top of classical AWK. Modernizing the *frawk* interpreter to support these AWK extensions would likely increase the measured parallelizability rates in real-world scenarios.

Lastly, the current parallelization analyzer does not support certain less commonly used AWK language constructs. For example, there is no support for the parallel analysis of custom user functions implemented yet. As a result, such constructs are currently treated conservatively, which limits the applicability of parallelization for some advanced AWK programs. This implementation limitation can be overcome in the future to increase the overall applicability of parallel AWK.

6.4 Discussion/Reflection

The study on parallelization of AWK language fits into the long-standing research direction of automatic parallelization. This topic was actively developed over the decades, primarily in the context of compiled languages and high-performance computing, where compiled programs and static typing enabled predictable analysis and transformation for successful parallelization. In contrast, the AWK tool has received limited attention, despite its continued relevance in real-world data processing tasks.

The research highlights some conceptual challenges associated with parallelizing AWK related to shared global state and side effects. This, together with the dynamic and interpreted execution environment, makes the classical parallel optimizations limited and hardly implementable for the language. On the other hand, the data-based parallelization is more feasible given the language specifics.

The language was originally designed with a strictly sequential execution model. As a result, introducing parallel execution requires careful analysis to ensure that program semantics are preserved. The approach explored in this research demonstrates that, while full parallelization is difficult, a meaningful subset of AWK programs can be safely executed in a data-parallel way using static analysis techniques. Its combination with a MapReduce-inspired execution model makes it possible to correctly handle global state in parallel settings in a lot of cases.

From a broader perspective, the research shows how the legacy tools can be modernized for exploiting modern hardware capabilities. Rather than replacing AWK with more complex frameworks or languages, the proposed approach enhances the language capabilities while preserving its simplicity and portability. The proposed approach enables a scalable solution for AWK script execution on growing datasets, thereby aligning with modern software engineering practices for large-scale data processing [37].

In practice, the ability to improve performance without requiring users to modify their existing scripts lowers the barrier to tool adoption. This is especially important for users who rely on simple and well-established tools in production environments, where stability and backward compatibility are critical. Moreover, the demonstrated performance improvements could increase the tool's popularity for new scripts and modern data pipelines, attracting developers who seek both the familiarity of AWK syntax and modern multi-core scalability.

The findings of this research are relevant to several stakeholder groups within both industry and academia. Researchers in program analysis, compiler design, and parallel computing can use the proposed approach as a case study demonstrating how automatic parallelization techniques can be adapted to dynamically typed and interpreted languages. Developers and organizations would benefit from improved performance and transparent integration into existing workflows, and gain more efficient utilization of modern multi-core systems. System administrators and DevOps engineers, who use AWK for analysis, monitoring, and automation tasks, could perform tasks more efficiently without changing established workflows. Data engineers and analysts can use a scalable AWK version to explore large datasets on modern multi-core hardware. Additionally, a lot of legacy Unix-based infrastructure could be modernized without investing significant effort into migrating it to newer frameworks.

Overall, the thesis confirms the idea that innovations in computer science are not limited to the development of new systems but also include the refinement and extension of existing technologies. The research contributes to the ongoing effort to make software systems more scalable and sustainable by extending the capabilities of established tools rather than replacing them. This research demonstrates that revisiting established technologies can still yield valuable improvements. Although AWK is a mature language with decades of refinement, its modernization through the use of modern technologies and parallelization techniques can give valuable improvements over the traditional implementations.

6.5 Future work

Several directions remain for extending this research and improving its practical usefulness. The most immediate area of improvement is a hybrid in-memory/disk output handling and scalability of the current buffering approach. In the current implementation, all output generated during parallel execution is stored in per-worker in-memory buffers. Future work could therefore focus on improving the scalability of this mechanism by introducing a hybrid buffering strategy, where large outputs are stored in temporary disk files once memory usage exceeds a threshold. This would preserve deterministic ordering while allowing the system to handle significantly larger workloads with reduced memory pressure.

A second promising direction is to extend the current static-analysis approach with dynamic mechanisms and automatic code restructuring to extend the applicability. The present design is still conservative on the constraints, and some scripts are rejected while they can still be safely parallelized with small runtime adjustments. For example, the common pattern detected in multiple scripts is shown in Figure 6.1. This example is non-parallelizable currently, due to the condition relying on a global value stored in *NR* variable. This variable is specially built in to access the line number. However, this condition will always be false for all parallel executors except the first one. This script can be parallelized by dynamically replacing the condition with false for all parallel workers except the first one. Similar dynamical code adjustments can potentially improve the current research approach and drastically boost the applicability of AWK parallelization.

Additionally, future work can focus on exploring parallel AWK execution within Shell

```
{
  if (NR < 1) {
  } else {
    ...
  }
}
```

Figure 6.1: Dynamically parallelizable pattern

pipelines. Some efforts on the automatic parallelization of Shell data pipelines are explained in the paper by Vasilakis et al. [34]. A tool explored in the paper can be potentially integrated with the AWK parallelizer. The integration would create a comprehensive framework that enables both record-level parallelism within AWK scripts and coordinated data-parallel execution across entire Unix-style processing chains.

Another long-term research direction is the exploration of distributed AWK execution. While the current prototype focuses on shared-memory parallelism within a single node, the same principles of static dependency analysis and deterministic aggregation could be extended to distributed environments. The current research shows an application of the MapReduce principle for AWK script execution. The same principle can be applied in the distributed settings. In that case, the main script will be executed in the distributed settings, and the variable aggregation will be executed as part of the reduction step. Such an approach would allow AWK programs to process datasets that exceed the memory and computational limits of a single machine, making the language applicable to large-scale data engineering scenarios. Additionally, it would open the possibility of lightweight, AWK-based alternatives to more complex modern big-data frameworks.

Overall, the future of AWK parallelization lies in broadening both its applicability and execution scope. Improvements in static and dynamic analysis, support for pipeline-based processing, and potential expansion toward distributed systems could significantly strengthen the practical relevance of this research. Together, these directions would transform the presented prototype from a proof of concept into a mature execution framework capable of supporting modern large-scale text processing workloads.

Bibliography

[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988. ISBN 0-201-07981-X.

[2] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. The one true awk, 2022. URL `https://github.com/onetrueawk/awk`. Original AWK implementation (often referred to as nawk). Accessed: 2026-06-01.

[3] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, San Francisco, CA, 2001. ISBN 1-55860-286-0.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, 1967. doi: 10.1145/1465482.1465560.

[5] Mike Brennan. *mawk - pattern scanning and text processing language*. mawk project, 1991. URL `https://invisible-island.net/mawk/mawk.html`. Accessed: 2026-06-01.

[6] William Bugden and Ayman Alahmar. Rust: The programming language for safety and performance. 2022. doi: 10.48550/arXiv.2206.05503.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36 (4):28–38, 2015.

[8] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 2007. doi: 10.1145/1247480.1247602.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. doi: 10.1145/1327452.1327492.

[10] Gábor Dombay. Practical awk benchmarking: Performance analysis of gawk, mawk, and nawk using pareto frontier. https://awklab.com/practical-awk-benchmarking, 2026. Accessed: 2026-06-01.

[11] ezrosent. frawk: an efficient AWK-like language. https://github.com/ezrosent/frawk, 2025. Accessed: 2026-06-01.

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. 9(3):319–349, 1987. doi: 10.1145/24039.24041.

[13] Al Geist, Adam Beguelin, and Jack Dongarra. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press. ISBN 978-0-262-57108-1.

[14] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly - polyhedral optimization in LLVM. 2011. URL https://www.research.ed.ac.uk/en/publications/polly-polyhedral-optimization-in-llvm/. Accessed: 2026-06-01.

[15] John L. Gustafson. Reevaluating amdahl's law. 31(5):532–533, 1988. ISSN 0001-0782. doi: 10.1145/42411.42415.

[16] Felipe Hoffa. GitHub on BigQuery: Analyze all the open source code. URL https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code. Accessed: 2026-06-01.

[17] Ben Hoyt. GoAWK, an AWK interpreter written in go. URL https://benhoyt.com/writings/goawk/. Accessed: 2026-06-01.

[18] Ben Hoyt. The state of the AWK. *LWN.net*, May 2020. URL https://lwn.net/Articles/820829/?utm_source=chatgpt.com. Accessed: 2026-06-01.

[19] IEEE 754 Committee. IEEE standard for floating-point arithmetic. IEEE Std 754-2008, Institute of Electrical and Electronics Engineers (IEEE), Microprocessor Standards Committee, New York, NY, USA, August 2008. URL https://www.dsc.ufcg.edu.br/~cnum/modulos/Modulo2/IEEE754_2008.pdf. Accessed: 2026-06-01.

[20] Vinay K, Thushar S, Vempalli Surya, and Meena Belwal. Comparative analysis of parallel compiler optimizations: A survey. doi: 10.2139/ssrn.5089085.

[21] Shiwang Kalkhanda. *Learning AWK Programming: A fast, and simple cutting-edge utility for text-processing on the Unix-like environment*. Packt Publishing Ltd, 2018. ISBN 978-1-78839-708-7.

[22] Ivans Kravcevs. Design and implementation of parallel AWK, 2026. URL `https://doi.org/10.5281/zenodo.20580749`. Accessed: 2026-06-11.

[23] Jimmy Lin, Chris Dyer, and Graeme Hirst. *Data-Intensive Text Processing with MapReduce*. doi: 10.1007/978-3-031-02136-7.

[24] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. 44(9):65–78, August 2009. doi: 10.1145/1631687.1596563.

[25] Samuel P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Synthesis Lectures on Computer Architecture. Springer International Publishing, Cham, Switzerland, 2012. doi: 10.1007/978-3-031-01736-0.

[26] L. Prechelt. An empirical comparison of seven programming languages. 33(10):23–29. doi: 10.1109/2.876288.

[27] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. doi: 10.1109/HPCA.2007.346181.

[28] Arnold Robbins. *GAWK: effective AWK programming : a user's guide for GNU Awk*. Free Software Foundation, Boston, MA, ed. 3 edition, 2003. ISBN 978-1-882114-27-6.

[29] Shreyas Sakharkar. Systematic review: Analysis of coding vulnerabilities across languages. *Journal of Information Security*, 14:330–342, 01 2023. doi: 10.4236/jis.2023.144019.

[30] Vivek Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 16–30. doi: 10.1007/3-540-57502-2_37.

[31] Diomidis Spinellis, Panos Louridas, Maria Kechagia, and Tushar Sharma. Broken windows: Exploring the applicability of a controversial theory on code quality. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 400–412, 2024. doi: 10.1109/ICSME58944.2024.00044.

[32] Tim Süß, Lars Nagel, Marc-André Vef, André Brinkmann, Dustin Feld, and Thomas Soddemann. Pure functions in c: A small keyword for automatic parallelization. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 552–556, 2017. doi: 10.1109/CLUSTER.2017.32.

[33] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. doi: 10.1145/79173.79181.

[34] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilleas Benetopoulos, and Lazar Cvetković. PaSh: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 49–66. doi: 10.1145/3447786.3456228.

[35] Vijay A. Verma. Sample CSV files / data sets for testing (till 5 million records) - sales. URL `https://excelbianalytics.com/downloads-18-sample-csv-files-data-sets-for-testing-sales/`. Accessed: 2026-06-01.

[36] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly Media, 4 edition, 2012. ISBN 978-1449393090.

[37] James Warren and Nathan Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster. ISBN 978-1-63835-110-8.

[38] Siwei Wei, Guyang Song, Senlin Zhu, Ruoyi Ruan, Shihao Zhu, and Yan Cai. Discovering parallelisms in python programs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 832–844, New York, NY, USA, 2023. doi: 10.1145/3611643.3616259.

[39] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10. doi: 10.5555/1863103.1863113.