A Technique for Automated Parallel Optimization of Function Calls in C++ Code

Shuruq Abed Alsaedi[1], Fathy Elbouraey Eassa[2], Amal Abdullah AlMansour[3],
Lama Abdulaziz Al Khuzayem[4], Rsha Talal Mirza[5]
Department of Computer Science-Faculty of Computing and Information Technology,
King Abdulaziz University, Jeddah, Saudi Arabia[1, 2, 3, 4, 5]
Department of Computer Science-College of Computer Science and Engineering,
Taibah University, Yanbu 46421, Saudi Arabia[1]

***Abstract*—In modern software development, achieving high performance increasingly relies on effective parallelization. While much of the existing research has focused on loop-level parallelism, function-level parallelization remains relatively underutilized. Yet, in many real-world applications, function calls serve as natural units of computation that could greatly benefit from concurrent execution. To address this gap, we present an automated tool that analyzes sequential C++ code, identifies independent function calls, and evaluates their suitability for parallel execution. The tool performs three key analyses: dependency analysis to detect function calls, context analysis to understand execution conditions, and workload assessment to determine whether parallelization would result in significant performance benefits. Based on the analysis results, the tool transforms eligible function calls into parallel equivalents without altering the original program logic. Additionally, the tool generates detailed Control Flow Graphs (CFG) for each function in three formats, facilitating further structural analysis. Three benchmark programs were used in experimental testing. The evaluation measured both sequential and parallel execution times, along with the computed performance gain expressed as a percentage reduction in runtime. Results demonstrated the tool's ability to improve execution efficiency and reduce processing time. These outcomes emphasize the tool's role in advancing function-level automatic parallelization. The tool showed notable performance improvements across the three benchmark applications, with the Employee Performance System achieving the highest improvement of 54.6%, followed by the Genomic Sequence System at 48.3%, and the Book Reviews System achieving an improvement of 36.1%. Demonstrating the tool's ability to improve efficiency via automated function-level parallelization.**

Keywords*—Automatic parallelization; function-level parallelization; C++ code optimization; parallel computing; control flow graph; dependency analysis; performance optimization

I. Introduction

Improvements of multi-core processors have increased the adoption of parallel programming among software developers. However, developing parallel code remains a challenging and error-prone task. As a result, automating code parallelization has emerged as a hot topic in high-performance applications research [1]. Available parallelization techniques usually target specific sections of code that are suitable to concurrent execution. Loops are the most common candidates. Loops often contain iterative operations that can be executed in parallel without interdependencies, making them ideal for leveraging multi-core and multi-threaded architectures.

However, there are other code constructs, such as function calls, that can enhance performance and efficiency if they are properly utilized.

By focusing on function calls, parallelization strategies can effectively improve the software applications. However, despite years of research, the automatic parallelization of function calls remains an unresolved problem. Function calls often still require manual parallelization [1].

The motivation for this work stems from several critical challenges in modern software development:

1) Underutilization of parallelization opportunities: While parallelization research has focused heavily on optimizing loops, function calls have received comparatively less attention. This highlights a significant gap in current parallelization research, as function calls provide substantial opportunities for enhancing performance. In many real-world applications, computational workload is distributed across numerous function calls rather than concentrated in loops. For instance, data processing applications often involve multiple independent operations such as data extraction, filtering, aggregation, and statistical analysis, each implemented as separate functions. These functions can potentially execute concurrently without dependencies, yet they remain sequential in most codebases due to the lack of automated transformation tools. Addressing the parallelization of function calls helps improve the efficiency and scalability of code. Nonetheless, developing parallel code or converting certain sequential code to run in parallel is a challenging task that requires significant effort from programmers.

2) Complexity of manual parallelization: The complexities of parallel programming start from the problem definition, as not all problems are suited for parallelization. Programmers must carefully consider different aspects of parallel processing, such as how to distribute the current workload across parallel threads and manage communication between these threads [2]. This manual effort is time-consuming, error-prone, and requires deep expertise in both parallel programming paradigms and the application domain. Developers must analyze data

dependencies, identify race conditions, manage synchronization, and ensure thread safety tasks that significantly increase development time and maintenance costs.

3) Performance demands of modern applications: Contemporary software applications face increasing computational demands. From big data analytics and scientific computing to machine learning and real-time systems, the need for efficient parallel execution has never been greater. C++ programming language is known for its performance and fine-grained control and become an essential component in the development of high-performance computing applications, including those that leverage parallelism [8]. However, the gap between available hardware capabilities and software utilization remains substantial. Many existing sequential C++ applications could benefit significantly from parallelization but lack the resources or expertise for manual transformation.

Our proposed tool was designed to analyze and transform C++ code, which was selected due to its relevance in performance-sensitive software and its rich set of parallelization libraries. This gap is particularly significant because:

- Natural Computation Units: Functions represent natural units of computation with well-defined interfaces and encapsulated logic, making them ideal candidates for parallel execution.
- Coarse-Grained Parallelism: Function-level parallelism offers coarse-grained task parallelism that can better utilize modern multi-core architectures with lower synchronization overhead compared to fine-grained loop iterations.
- Code Maintainability: Parallelizing at the function level preserves code structure and maintainability better than aggressive loop transformations.

This study proposes a fully automated tool that offers an advanced method for analyzing and selecting suitable function calls for parallelization in C++ code, a language with unique structural and concurrency characteristics. The proposed tool integrates static analysis, control flow graph generation, and a lightweight scoring mechanism to identify and transform independent function calls into parallel equivalents using *std::async*. By applying this technique to C++, the tool bridges a critical gap in existing research. This contribution offers practical value for developers working with performance-critical C++ applications, especially in cases where loop-level parallelism is insufficient.

The main contributions of this work are:

1) Comprehensive static analysis framework: We introduce a three-phase analysis approach combining:

- Dependency Analysis: Automatically detects function calls and analyzes their data and control dependencies to ensure correctness of parallel transformations.
- Context Analysis: Evaluates execution contexts to determine parallelization safety and identify functions that can be executed asynchronously without affecting program semantics.
- Workload Assessment: Employs a work potential metric that quantifies computational complexity, enabling prioritization of parallelization candidates.

2) Automated code transformation pipeline: The tool provides end-to-end automation from analysis to code generation:

- Automatically transforms eligible sequential function calls into parallel equivalents using C++ standard library features (std::async, std::future), without requiring manual intervention or code restructuring.
- Preserves original program logic and semantics while introducing parallelism.

3) Visual Analysis and Debugging Support:

- Generates detailed Control Flow Graphs (CFGs) for each function in three formats (DOT, PDF, and TXT), facilitating structural analysis and verification.
- Provides comprehensive reporting of parallelization decisions, including work potential scores and context classifications.
- Enables developers to understand, validate, and fine-tune the automated transformations.

In this study, three benchmark programs were selected to conduct experiments. The experiments aimed at measuring parallel execution time, sequential execution time, and computed performance improvement, representing the percentage reduction in execution time. The results show that our proposed tool successfully enhances performance and reduces execution time. The findings highlight the tool's capability as a step forward in bridging the current gap in automatic parallelization research, particularly at the function-call level.

The structure of this paper is as follows: Section II provides background information; Section III reviews related work; Section IV describes the system architecture and methodology; Section V presents the experimental results and discussion; Section VI evaluates performance improvements; and Section VII concludes the paper.

II. Background

Historically, processor design has focused on combining more advanced features, achieving higher clock speeds and increasing thermal limits. However, the pursuit of enhanced performance remains a necessary demand. This has driven the adoption of integrated multi-processor (multi-core) architectures [3].

However, transistors can't keep getting smaller indefinitely. Nowadays, as transistors become thinner, chip makers aim to manage heat generation and power usage; these are two major challenges. Additionally, performance enhancement techniques like running multiple instructions at the same time have reached their limits.

Due to these problems, the rate of processor performance improvements has started to slow down. In the 1990s, chip performance increased by 60 percent each year, while from

2000 to 2004, this growth slowed to 40 percent annually, with performance only rising by 20 percent.

In fact, improvements in computer performance have essentially resulted from making chips smaller and increasing their number of transistors. According to Moore's Law, this approach has led to faster chip speeds and reduced costs. Instead of making one super powerful core, companies are now creating chips with multiple cores that use less energy and run cooler. Even though these multicore chips might not be as fast as the top single-core ones, they improve overall performance by handling more tasks at the same time [4].

Many modern programs require more computing power than traditional single-threaded computers can offer. In order to achieve better performance, parallel processing is necessary to meet demand. Parallelism allows executing multiple tasks at the same time to solve complex problems in less time. Multicore processing requires running programs on more than one core within a single CPU chip, which is a type of parallel processing. Multicore utilization means using the CPU's multiple cores efficiently [5].

In parallel processing of programs, instructions are split across many processors to run the same program faster than with sequential processing. Some programs that require tasks such as sorting and searching are designed to use parallel processing. Such programs are often tested on large databases. Experimental results using bubble sort and linear search show that sharing the load and utilizing multicore parallel processing make tasks easier and quicker compared to sequential processing. Parallel processing and multicore processing both involve the ability to run code simultaneously on different CPUs, machines, or cores. Multicore processing is designed to perform tasks in parallel. A core can be considered as a small processor within a larger processor. Therefore, optimizing code for multicore processing is closely tied to parallelization [4] [5].

Finding opportunities to run code automatically in parallel is essential for developing efficient programs to run on many multithreaded applications. This area has been researched for many years. Even though hardware provides some fine-grained parallelism, compiler researchers have to develop techniques to automatically convert source code into suitable parallel applications.

With the growing ubiquity of distributed computing, the software industry demands compiler technologies capable of automatically parallelizing software. However, the demand for automatic parallelization in compilers is growing as clusters and other kinds of distributed computing become widely adopted, and CPU technology moves toward higher levels and coarser granular parallelism [6].

Generally, automatic parallelization involves the following three procedures [7]:

- Dependency analysis to identify potential parallelism regions: Carrying out a dependency analysis to identify regions in the code that are capable of parallelization.
- Restructuring the program into blocks that can run in parallel: Organizing the code into number of blocks that can run simultaneously. In this phase, different transformations are applied to maximize the level of parallelism achievable within the code.
- Generating parallel code suitable for specific architectures: Producing parallel code optimized for a particular architecture by assigning program blocks to available processing units and developing appropriate strategies to enable parallelism based on the targeted system.

Building on these core steps, it is essential to consider how specific programming languages, such as C++, support parallelism through built-in features and libraries. C++ supports multithreaded execution through features such as *std::async*, which play a basic role in parallelism [15]. Other standard multithreading features in C++ that simplify parallel programming include [15]:

- *std::thread*: To manually manage the creation and control of new threads of execution.
- *std::async*: Enables asynchronous function execution, with thread management handled automatically by the system.
- *std::future*: Used with *std::async* or *std::promise* to obtain results from asynchronous executions.
- *std::promise*: For sending a value from one thread to another, typically used with *std::future*.
- *std::mutex*: Prevents race conditions by ensuring that only one thread can access shared data at a time.

Several prior studies have extensively discussed the evolution of multicore architectures, the challenges in optimizing code for parallel execution, and the importance of automatic parallelization to utilize modern hardware efficiently. These works collectively emphasize the necessity of transitioning from sequential to concurrent execution, while highlighting fundamental techniques such as dependency analysis, control flow restructuring, and task distribution across cores. They also explore compiler-assisted approaches, parallel programming models, and language-specific solutions that aim to simplify or automate this transition.

While these contributions have laid important groundwork for parallel computing, particularly in loop and object-oriented contexts, there remains a noticeable gap in the automation of function-level parallelization, especially for statically analyzed, general-purpose C++ applications. This study builds on those foundational insights by introducing a tool that automates the detection, analysis, and transformation of independent function calls into parallel equivalents using C++'s native concurrency libraries. The proposed solution requires no manual intervention or external annotations, offering a fully automated pipeline that strengthens the practical adoption of parallelism in real-world C++ systems.

III. Related Works

Parallel computing has recently become more in demand due to the increasing growth of multi-threaded architecture. There are some papers that discuss certain techniques that aim to efficiently transfer a sequential program into a parallel one.

In [9], the authors introduce an approach that uses the concept of futures synthesis for pure method call parallelization. Their approach employs three strategies to achieve automatic parallelization through future synthesis: parallelism analysis, future synthesis, and threshold synthesis. In the future synthesis phase, the proposed system identifies pure method calls and labels them as async expressions. After that, the system conducts a parallelism analysis to determine which async expressions should be executed sequentially to avoid overhead. Finally, threshold synthesis utilizes the results of the parallelism analysis to synthesize predicate expressions for deciding whether a specific pure method call should be executed in parallel or sequentially.

Our tool focuses on C++ and operates at the static level, targeting functions that are independent and computationally intensive. It does not rely on runtime profiling but instead uses a scoring system to identify which functions should be parallelized. While in study [9] the tool is adaptive and runtime-aware, ours offers a lightweight and deterministic transformation pipeline that produces a ready-to-compile parallel version of the original program, making it ideal for users seeking fast and fully automatic optimization.

In [16], authors introduced Lazy-Parallel Function Calls, a compiler-level approach that enables automatic parallelization in imperative languages by combining lazy evaluation semantics with nano-thread scheduling. Their method ensures correctness by leveraging immutability and deferring function execution until results are required, enabling parallelism without altering program behavior. In contrast, our tool targets eager, function-level parallelization in C++ using *std::async*, automatically identifying independent function calls and transforming them for concurrent execution. While their work emphasizes semantic safety, our approach focuses on performance gains in compiled C++ applications.

In [10], the authors present an automatic technique that analyzes code written in the Java programming language to detect method calls that can be parallelized and transform them accordingly. The proposed technique is based on control dependence and data dependence analysis to construct the execution flow and determine accessed data.

APAC [11] is a compiler that converts sequential source code written in the C++ language into task-based parallelized code by inserting special directives. This compiler utilizes the LibTooling library and clang-tools from the LLVM framework. It encapsulates each function body with an OpenMP directive, and each function or method call with an OpenMP statement.

As authors stated in study [12], there are many recursive functions that contain a significant amount of built-in parallelism, but it is not always clearly visible. Thus, a number of studies have been conducted to detect parallelism and utilize it. Mainly by targeting calls that can be executed concurrently. However, some recursive algorithms do not permit the concurrent execution of function calls. According to the authors [12], they examined ways to extract concurrency from specific recursive function calls that are not typically parallelizable by common parallelization methods. Their primary approach involves parallelizing them at a finer level, relying on multi-threaded architecture and a multi-core as the infrastructure. The authors stated that, compared to conventional parallel systems, multi-core systems support a greater degree of fine-grained parallelism by providing thread parallelism and reducing overhead in thread communication.

Autopar [13] is a tool that targets to automatically parallelize certain recursive function calls using analysis of static programs. It detects the recursive functions inside a certain program. After that, it moves to the analysis phase by gathering related information about such functions without the need for rewriting the program or requiring developer involvement. Finally, Autopar introduces the parallel constructs necessary to achieve automatic parallelization. In fact, their tool applies to a specific class of recursive functions that use a divide-and-conquer approach. They tested four benchmarks—bionic, fractal, heat, and knapsack—using their recursive function parallelization tool.

In [14], the proposed tool applies a coarse-grained approach to task parallelism. It utilizes OpenMP directives and inserts them into the input C code. The output program is a multithreaded C code that can use different cores in a shared memory system. The system operates between the compiler and the application. The generated output is compiled just like any ordinary C code. Both the input and output of their proposed tool are source code, making it a source-to-source transformation tool.

In [17], authors propose two optimizations—Last Parallel Call Optimization (LPCO) and its generalization, Nested Parallel Call Optimization (NPCO) which is designed to improve the efficiency of and-parallel logic programming. Their methods restructure parallel goal execution in logic programs to minimize redundant parallel tasks and enable fast backtracking. These techniques are tailored for the control-parallel execution model, where correctness depends on preserving goal dependencies.

Table I presents a comprehensive comparison of the limitations identified in existing approaches and how our proposed tool addresses these gaps.

TABLE I. LIMITATIONS OF EXISTING APPROACHES AND OUR PROPOSED SOLUTION

Aspect	Existing Approaches Limitations	Our Proposed Tool Solution
Language Support	Limited to specific languages: Java [9][10], C [14], logic programming [17]	Comprehensive C++ support with standard library integration (std::async, std::future)
Function Coverage	Restricted to recursive [12][13] or pure methods [9]; ignores general function calls	Analyzes all function types: recursive, non-recursive, member functions, and standalone functions
Analysis Type	Runtime profiling required [9]; limited static analysis [13][14]	Comprehensive static analysis combining dependency, context, and workload assessment
Automation Level	Semi-automated requiring manual refinement [11]; domain-specific [17]	Fully automated end-to-end pipeline from analysis to code generation

IV. METHODOLOGY

This study introduces a tool that addresses an important parallelization technique: unlike loop parallelization, research on parallelizing function calls is limited. This tool offers an improved method for analyzing and selecting the applicable functions for parallelization in C++ code. This tool includes AST traversal and work potential metrics to evaluate computational effort and identify functions with high parallelization possibility. Moreover, it generates detailed control flow graphs as well as an analysis context in order to give valuable insights to programmers who look to optimize program efficiency through parallel execution. As Fig. 1 shows, the tool has five main functionalities, which we will discuss in detail. These main functionalities are: Function Analysis, Context Analysis for Parallelization, Control Flow Graph (CFG) Generation, Work Potential Calculation, and CFG Visualization.

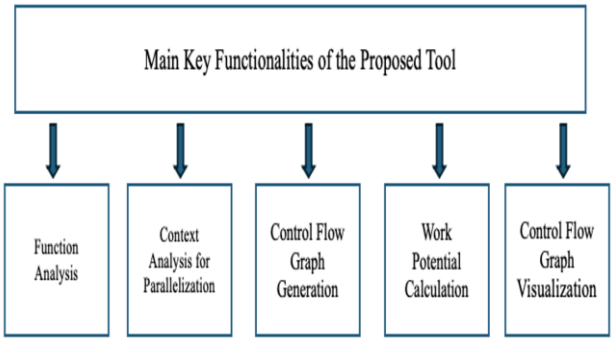

Fig. 1. Five key functionalities of the tool.

Main key functionalities of the proposed tool:

A. Function Analysis

- Parses the C++ source code using Clang's libclang to generate an abstract syntax tree (AST).
- Extracts functions and analyzes each line for function calls to understand dependencies and execution flow.

B. Context Analysis for Parallelization

- Determines whether each function call can be parallelized based on its context (e.g., whether it's inside loops or conditionals).

C. Control Flow Graph (CFG) Generation

- Generates a CFG for each function, including nodes for statements and edges representing control flow.
- Annotates each function call with contextual information (e.g., if part of a loop) to highlight non-parallelizable code.

D. Work Potential Calculation

- Each function is assigned a "work potential" score based on the complexity of its control flow and nested calls.
- Higher work potential indicates a candidate for parallelization, while certain functions (like main) are excluded due to its structural importance.

E. Control Flow Graph (CFG) Visualization

- Uses Graphviz to create a DOT file for each function's CFG, saving it as a DOT, a PDF, and TXT file.
- The CFG graphically represents the control flow, showing how different statements and function calls are interconnected.

Our tool utilizes the use of libclang for parsing C++ code (AST) and graphviz for visualizing CFGs. Libclang is a library that offers tools for parsing and analyzing C++ code. It is part of the LLVM project and enables the generation of an Abstract Syntax Tree (AST). Graphviz, on the other hand, is a visualization software used for creating graphs and diagrams. It is specifically utilized to generate Control Flow Graphs (CFGs), which visually represent the flow of execution within a program, helping programmers understand how different parts of the code are connected and identify potential bottlenecks or parallelization opportunities. Moreover, it usesC++ Parallelization and Synchronization Library (std::async: allows asynchronous function execution and enabling tasks to run concurrently in separate threads). As Fig. 2 illustrates, the tool's architecture starts with C++ source code input, which undergoes scanning to detect functions. This is followed by C++ code analysis, which performs dependency and context analysis generate Control Flow Graphs (CFGs) and calculate work potential. Finally, the tool moves to C++ code generation, completing the process [10].

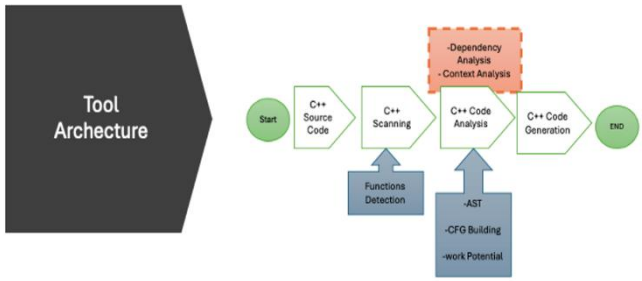

Fig. 2. The proposed tool archeticure.

A detailed description of the tool architecture:

1) Context analysis: This analyzes the context in which function calls occur, meaning it analyzes C++ source code to identify functions that can be parallelized to improve performance.

Tables II and III provide a detailed clarification of the situations under which functions are considered parallelizable and when they are not suitable for parallelization.

2) CFG building: CFG builder drives the analysis of a specified C++ code, generates control flow graphs (CFGs), saves them in a designated output directory, and classifies functions by their work potential (which will be discus1§sed) for advanced parallelization identification.

TABLE II. CONDITIONS FP;OR CONSIDERING A FUNCTION CALL AS POTENTIALLY PARALLELIZABLE

Condition	Reason
Not Part of Control Structure: The function call is not used within the condition of control structures such as if, while, for, or switch statements.	Function calls within conditions are critical for the control flow and may need to be executed sequentially to maintain program correctness
Not Inside Loops: The function call is not located within the body of for or while loops.	Parallelizing function calls inside loops can lead to synchronization issues and may not provide performance benefits due to overhead.
Not Part of Return Statements: The function call is not used directly within a return statement.	The result of the function is immediately required for the return value, so parallelizing it would necessitate waiting for its completion. Attempting to execute such a function asynchronously (in parallel) does not provide any performance improvement and may introduce unnecessary overhead. This essentially cancels out any potential advantages of parallelization in that context.
Not Part of Throw or Assert Statements: The function call is not used within throw or assert statements. **Throw**: is used to signal an exception or error condition. **Assert**: The assert function checks / validates a condition at runtime.	These statements are critical for error handling and program correctness, requiring immediate execution.
Standalone Function Calls: The function call is a standalone statement or assigned to a variable outside the contexts mentioned above.	Such calls can be executed asynchronously without affecting the program's control flow.

TABLE III. CONTEXTS WHERE A FUNCTION CALL IS NOT SUITABLE FOR PARALLELIZATION

Number	Condition
1	Control Structure Conditions
2	Inside Loops
3	Return Statements
4	Part of Throw or Assert Statements

3) Work potential metric: work potential metric is used for prioritizing parallelization by suggesting specific functions to parallelize. This process is based on calculating the computational load or complexity of each function and assigning it a score. Functions with a high work potential score are likely to benefit from parallelization, so the tool identifies and lists these functions at the end of the analysis, see Fig. 3.

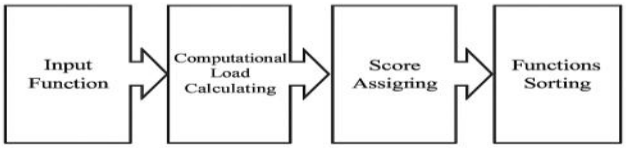

Fig. 3. Steps of the work potential metric.

4) Work potential purposes: The main purposes of the work potential metric are to quantify computational effort because functions with more statements and function calls need more computational processing, and to identify chances for parallelization, as high work potential indicates that a function may benefit from being executed in parallel which enhance performance. However, there are some excluded functions. Certain functions, like main, are excluded from parallelization even if their work potential score is high, because parallelizing the main function does not make sense. It organizes all other functions and controls the entire program's execution, so it is avoided from parallelization.

5) Work potential calculation: Calculation Method for finding Work Potential is a systematic method that requires analyzing C++ functions and utilizing Clang's libclang library to evaluate computational complexity. It uses Clang's libclang library to parse the C++ source code and generate an AST, and traverses each function's AST to analyze its structure. The Work Potential Counter (work_potential) counts statements and function calls. First, it is initialized to zero for each function. Then, it gets incremented for each statement encountered during traversal, as well as for each function call within the function.

V. RESULT AND DISCUSSION

We evaluated the parallelization tool on the BookLibrary benchmark program. BookLibrary, a C++ program, is considered a tool for data processing and analytics of book review data. It includes classes like Review, Book, Author, User, ExtractData, and ExtractDataset in order to organize and analyze information related to books, authors, users, and reviews. Our approach targets programs where function calls account for a significant portion of the workload. Therefore, we chose BookLibrary program because a substantial amount of its sequential work is executed through function calls. BookLibrary.cpp helps users load, analyze, and summarize book review data, with a particular focus on author and user review statistics, enabling insights into authors' popularity and reader engagement. Table IV shows the main components of the C++ code. In addition to those listed in Table IV, there are other functions that calculate and return the most, least, and average-reviewed books, as well as retrieve the most and least-reviewed authors based on their books' reviews.

TABLE IV. OVERVIEW OF MAIN COMPONENTS IN THE C++ CODE

Classification	Description
Data Representation	Review: Represents a review of a book with attributes like book ID and title.
	Book: Represents a book, including its title and list of reviews.
	Author: Represents an author with related books and users.
	User: Represents a user, with a user ID and related reviews.
Data Extraction and Aggregation	ExtractDatase: Reads book and review from CSV files, then stores them .
	ExtractData: create mappings between books and authors, reviews and users, then updates book review data.

A total of 53 functions were automatically extracted from the analyzed source file.

The detected functions range from getBookId, located at line 25, to the main function at the end of the code.

This demonstrates the tool's capability to comprehensively identify and list all functional components for further analysis (see Fig. 4).

```
================================================================================
           Enhanced CFG Builder with Function Call Context Analysis
================================================================================

Analyzing file: books_library.cpp
Found function: getBookId at line 25
Found function: getTitle at line 26
Found function: getUserID at line 27
Found function: getScore at line 29
Found function: getPrice at line 37
Found function: getProfileName at line 38
Found function: getHelpfulness at line 39
Found function: getTime at line 40
Found function: getSummary at line 41
Found function: getText at line 42
Found function: getTitle at line 58
Found function: getAuthors at line 59
Found function: getReviews at line 60
Found function: addReview at line 61
Found function: updateMediumScore at line 62
Found function: getMediumScore at line 76
Found function: resetReviews at line 77
Found function: operator== at line 78
Found function: getDescription at line 81
Found function: getImage at line 82
Found function: getPreviewLink at line 83
Found function: getPublisher at line 84
Found function: getPublishedDate at line 85
Found function: getInfoLink at line 86
Found function: getCategories at line 87
Found function: getRatingCount at line 88
Found function: getFullName at line 100
```

Fig. 4. Extracted function from book_library.cpp.

Fig. 5 below presents the tool's output that demonstrates the automated generation of Control Flow Graphs (CFGs) for several detected functions such as *getBookId*, *getTitle*, *getScore*, *getText*, *getReview*, and *getPrice*. For each function, the tool produces three distinct representations: DOT, PDF, and an extended TXT format to support both visual inspection and structural analysis. This comprehensive output enables the examination of the internal logic and control dependencies within each function. Additionally, it allows tracing execution paths, detecting redundant operations, and better understanding of the relationships among function calls. Overall, this automated CFG generation provides a valuable foundation for program comprehension, debugging, and performance optimization.

```
DOT file for function 'getBookId' saved to cfg_output/getBookId_cfg.dot
PDF file for function 'getBookId' saved to cfg_output/getBookId_cfg.pdf
TXT file for function 'getBookId' saved to cfg_output/getBookId_cfg.txt

Extended CFG for function 'getBookId' saved to cfg_output/getBookId_extended_cfg.pdf
DOT script for extended CFG saved to cfg_output/getBookId_extended_cfg.dot
Extended CFG details for function 'getBookId' saved to cfg_output/getBookId_extended_cfg.txt

DOT file for function 'getTitle' saved to cfg_output/getTitle_cfg.dot
PDF file for function 'getTitle' saved to cfg_output/getTitle_cfg.pdf
TXT file for function 'getTitle' saved to cfg_output/getTitle_cfg.txt

Extended CFG for function 'getTitle' saved to cfg_output/getTitle_extended_cfg.pdf
DOT script for extended CFG saved to cfg_output/getTitle_extended_cfg.dot
Extended CFG details for function 'getTitle' saved to cfg_output/getTitle_extended_cfg.txt

DOT file for function 'getUserID' saved to cfg_output/getUserID_cfg.dot
PDF file for function 'getUserID' saved to cfg_output/getUserID_cfg.pdf
TXT file for function 'getUserID' saved to cfg_output/getUserID_cfg.txt

Extended CFG for function 'getUserID' saved to cfg_output/getUserID_extended_cfg.pdf
DOT script for extended CFG saved to cfg_output/getUserID_extended_cfg.dot
Extended CFG details for function 'getUserID' saved to cfg_output/getUserID_extended_cfg.txt

DOT file for function 'getScore' saved to cfg_output/getScore_cfg.dot
PDF file for function 'getScore' saved to cfg_output/getScore_cfg.pdf
TXT file for function 'getScore' saved to cfg_output/getScore_cfg.txt

Extended CFG for function 'getScore' saved to cfg_output/getScore_extended_cfg.pdf
DOT script for extended CFG saved to cfg_output/getScore_extended_cfg.dot
Extended CFG details for function 'getScore' saved to cfg_output/getScore_extended_cfg.txt

DOT file for function 'getPrice' saved to cfg_output/getPrice_cfg.dot
PDF file for function 'getPrice' saved to cfg_output/getPrice_cfg.pdf
TXT file for function 'getPrice' saved to cfg_output/getPrice_cfg.txt
```

Fig. 5. Control Flow Graph (CFG) files generated for multiple functions.

```
Processing call: {'name': 'resetReviews', 'context': 'Potentially Parallelizable', 'line': 172}
Found call 'resetReviews' at line 172
Processing call: {'name': 'addReview', 'context': 'Potentially Parallelizable', 'line': 177}
Found call 'addReview' at line 177
Processing call: {'name': 'updateMediumScore', 'context': 'Potentially Parallelizable', 'line': 181}
Found call 'updateMediumScore' at line 181
Processing call: {'name': 'updateBooksReviews', 'context': 'Potentially Parallelizable', 'line': 186}
Found call 'updateBooksReviews' at line 186
Processing call: {'name': 'getReviews', 'context': 'Not Parallelizable: Part of return statement', 'line': 189}
Found call 'getReviews' at line 189
```

Fig. 6. Sample tool output showing identified function calls and their parallelization context.

Fig. 6 illustrates the tool's intermediate output during function call analysis. It captures many functions such as resetReviews, addReview, and updateMediumScore, and precise source code line numbers. Each function is reported along with its name, classification context, and source code line number.

```
Parallelizing top 4 functions:
extractMostReviewedAuthor: 11
extractLeastReviewedAuthor: 11
extractAverageReviewedAuthor: 11
getUserForAuthor: 5
CFG generation with function calls completed.
================================================================================
                                Analysis Complete
================================================================================

All CFGs have been generated and saved in the 'cfg_output' directory.
```

Fig. 7. Selected functions for parallelization.

Fig. 7 shows the final summary and highlights the four functions identified as suitable for parallelization based on their work potential score. The functions extractMostReviewedAuthor, extractLeastReviewedAuthor, and extractAverageReviewedAuthor were selected due to their significant contribution to the overall processing load. Control Flow Graph (CFG) generation was completed for all, and the results were saved in the cfg_output directory.

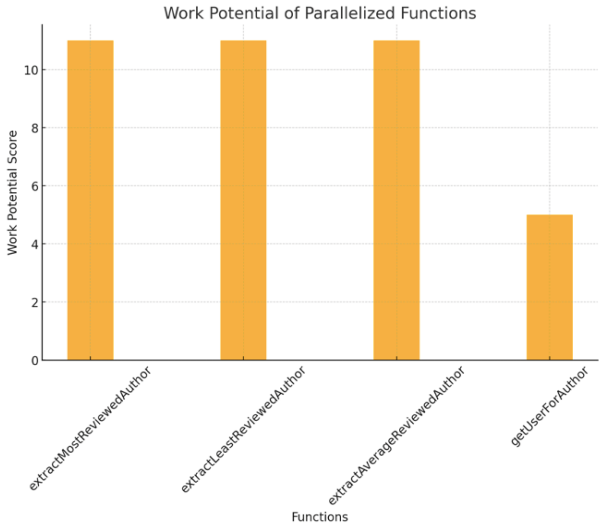

Fig. 8. Work potential scores for selected functions identified for parallelization.

Fig. 8 presents a bar chart showing the work potential scores of selected functions identified by the tool. The x-axis displays four functions: extractMostReviewedAuthor, extractLeastReviewedAuthor,

extractAverageReviewedAuthor, and getUserforAuthor, while the y-axis quantifies their corresponding scores. The first three functions each achieved a high potential score of 11, indicating strong suitability for parallelization due to their computational intensity. In contrast, getUserforAuthor received a lower score of 5, suggesting a relatively smaller parallel workload.

Fig. 9 shows the completion stage of the tool's transformation process, where selected functions have been successfully parallelized. The tool logs the transformed functions: extractMostReviewedAuthor, extractLeastReviewedAuthor, extractAverageReviewedAuthor, and getUserforAuthor and saves the modified source code as books_library_parallelized.cpp.

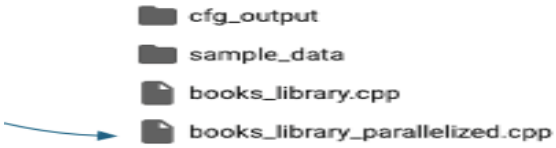

Fig. 9. Generation of the parallelized output file.

Fig. 10. Snapshot of Generated Control Flow Graph (CFG) files.

Fig. 10 presents a visual overview of the directory structure containing the Control Flow Graph (CFG) outputs generated by the tool. For each analyzed function, such as addUser, extractBooks, and extractMostReviewedAuthor, there are many output formats created:

- .dot: the raw DOT graph description format
- .pdf: a visual representation of the control flow
- .txt: a textual report of the CFG
- _extended_ versions: enriched CFGs that include additional semantic or contextual data

These files are organized systematically within the cfg_output directory, enabling developers to inspect the control flow structure of each function in both graphical and textual forms.

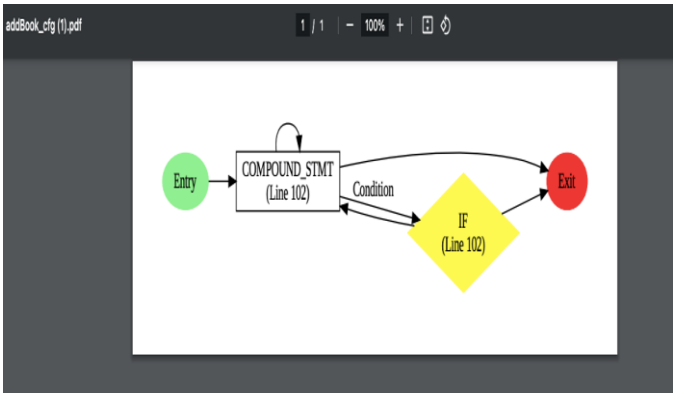

Fig. 11. PDF format Generated Control Flow Graph (CFG) for the Function addBlockFunction.

Fig. 11 presents a sample Control Flow Graph (CFG) generated in PDF format for the selected function addBlock. The graph begins at the Entry node and transitions to a Compound Statement at line 102, indicating a block of grouped instructions. It then leads to a conditional IF node, representing a branching decision in the function's logic.

Fig. 12 represents another format which is DOT format, while Fig. 13 represents TXT format.

Fig. 12. DOT format generated Control Flow Graph (CFG) for the function.

addBook_cfg.dot addBook_cfg.txt

```
Control Flow Graph for Function: addBook
==========================================

Nodes:
  addBook_entry
  addBook_exit
  addBook_stmt_0
  addBook_stmt_0
  addBook_decision_0
  addBook_stmt_0

Edges:
  addBook_entry --> addBook_stmt_0
  addBook_stmt_0 --> addBook_stmt_0
  addBook_stmt_0 --> addBook_decision_0
  addBook_decision_0 --> addBook_stmt_0
  addBook_stmt_0 --> addBook_exit
  addBook_decision_0 --> addBook_exit

```

Fig. 13. TXT format Generated Control Flow Graph (CFG) for the function addBlock( ) function.

```
std::optional<Author> extractMostReviewedAuthor(
    const std::map<std::string, Book>& books,
    const std::multimap<std::string, Review>& reviews) {

    ExtractData extractor(books, reviews);
    std::map<std::string, Author> authors = extractor.getAuthors();
    std::optional<Book> book = extractor.getMostReviewedBook();
```

Fig. 14. Original sequential implementation of extractMostReviewedAuthor function.

Fig. 14 shows the original sequential version of the extractMostReviewedAuthor function, where operations are executed in order. Fig. 15 presents the parallelized version automatically generated by the tool, using std::async to execute getAuthors and getMostReviewedBook concurrently. Our proposed tool identifies these opportunities without user intervention, which enhance the optimization process and facilitate it.

```
std::optional<Author> extractMostReviewedAuthor(
    const std::map<std::string, Book>& books,
    const std::multimap<std::string, Review>& reviews) {

    ExtractData extractor(books, reviews);
    auto authorsTask = std::async(std::launch::async, &ExtractData::getAuthors, &e
    std::map<std::string, Author> authors = authorsTask.get();

    auto bookTask = std::async(std::launch::async, &ExtractData::getMostReviewedBo
    std::optional<Book> book = bookTask.get();
```

Fig. 15. Parallelized Version of extractMostReviewedAuthor Using std::async.

VI. Evaluation Metrics

This section presents a performance analysis and evaluation through comparing serial and parallel implementations, with a focus on improvements achieved via function-level parallelization. The evaluation measures the impact of parallelization across three distinct data processing applications:

A. Book Reviews Analysis System

Processes large datasets of book reviews to extract insights such as the most and least reviewed authors, average review counts, and user-specific data. Ideal for evaluating function-level parallelism in data filtering and aggregation tasks.

B. Employee Performance Analysis System

Analyzes performance metrics of employees to compute scores, rankings, and performance categories. Suitable for testing parallel function execution in systems with structured, repetitive computations.

C. Genomic Sequence Analysis System

Performs operations such as sequence matching, frequency analysis, and data summarization on long character sequences. This benchmark represents workloads with high computational demand and pattern processing.

The result shows the impact of parallel execution in enhancing computational efficiency across different workloads and domains.

In this study, the Improvement metric was selected to quantitatively assess the performance gains achieved through function-level parallelization. The metric calculates the percentage reduction in execution time between the original serial implementation and the parallelized version, using the following formula:

$$\text{Improvement}\ (\%) = [(T_{serial} - T_{parallel}) / T_{serial}] \times 100 \quad (1)$$

Where *Tserial* denotes the execution time of the serial version, and *Tparallel* represents the time taken by the parallel implementation.

D. Book Reviews System Performance Metrics

Table V and Fig. 16 compare the execution times of serial and parallel implementations across various functional components. The results demonstrate that all operations benefited from parallelization, with notable reductions in execution time. The most significant improvement was observed in the Data Extraction function, which executed 43.7% faster compared to the serial version. In contrast, the Least Reviewed Author function showed the smallest improvement at 7.9%. Overall, the total execution time was reduced by 36.1%, confirming that the tool effectively enhanced computational performance.

Fig. 17 presents the performance data in a different format, emphasizing the percentage improvement for each function. Represented as a line graph, this view highlights the relative efficiency gains from parallelization.

TABLE V. BOOK REVIEWS SYSTEM PERFORMANCE METRICS (MILLISECONDS)

FUNCTION	SERIAL EXECUTION TIME	PARALLEL EXECUTION TIME	IMPROVEMENT (%)
DATA EXTRACTION	144.615	81.354	43.7
MOST REVIEWED AUTHOR	21.551	14.312	33.6
LEAST REVIEWED AUTHOR	20.847	19.196	7.9
AVERAGE REVIEWED AUTHOR	24.492	17.826	27.2
USER-AUTHOR ANALYSIS	33.697	23.941	29.0
TOTAL EXECUTION	245.205	156.629	36.1

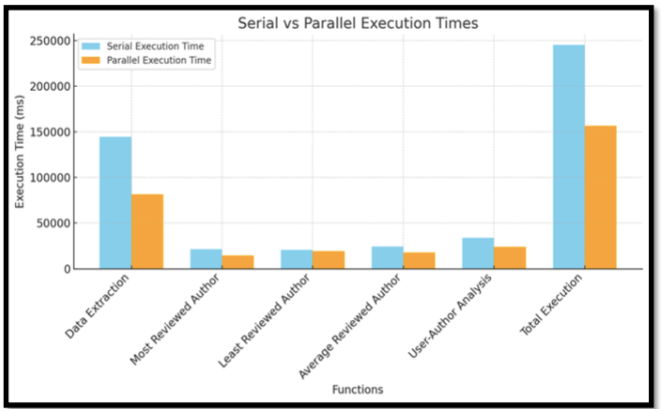

Fig. 16. Serial vs. Parallel execution time (book reviews analysis system).

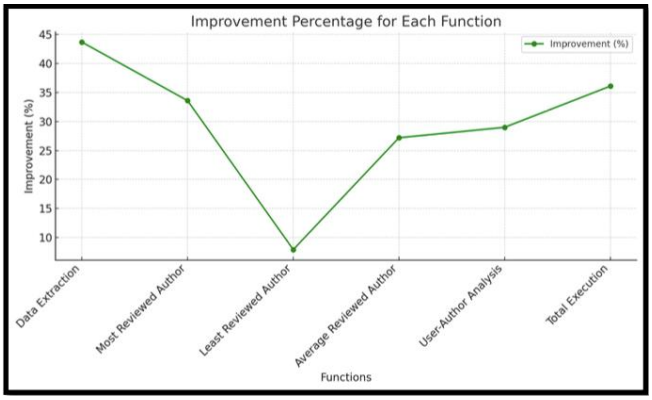

Fig. 17. Improvement percentage across functions (book reviews analysis system).

E. Employee Performance Analysis System

As Table VI, Fig. 18 and Fig. 19 show, most functions show significant performance improvements with parallelization, especially Top Performer Analysis (68.6%) and Data Extraction (64.1%). Skills Analysis also benefited notably (65.8%). However, Department Performance slightly worsened (-3.8%), due to the overhead and the low parallel potential. Overall, total execution time was reduced by 54.6%.

TABLE VI. EMPLOYEES' PERFORMANCE SYSTEM PERFORMANCE METRICS (MILLISECONDS)

Function	Serial Execuion Time	Parallel Execution Time	Improvement (%)
Data Extraction	107.516	38.627	64.1
Top Performer Analysis	34.632	10.873	68.6
Department Performance	17.922	18.611	-3.8
Skills Analysis	485	166	65.8
Department Statistics	15.850	11.766	25.8
Total Execution	176.405	80.049	54.6

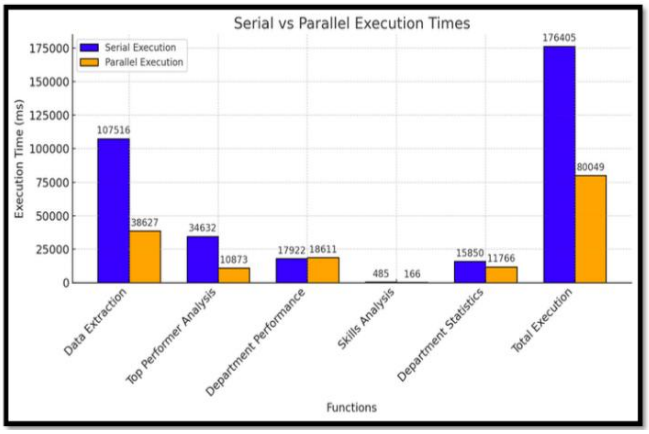

Fig. 18. Serial vs. Parallel execution time (employee performance analysis system).

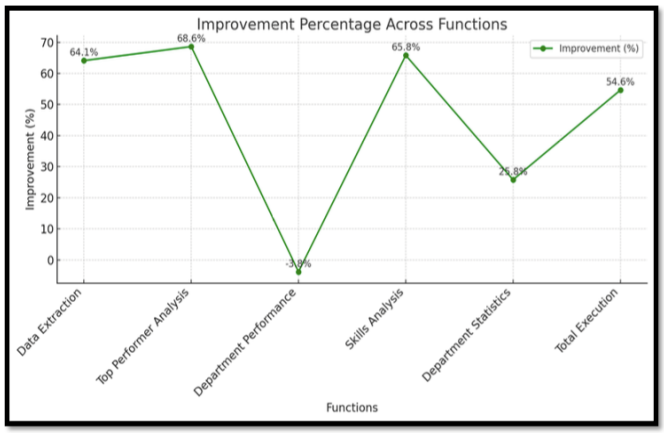

Fig. 19. Improvement percentage across functions (employee performance analysis system).

F. Genomic Sequence Analysis System

As Table VII, Fig. 20 and Fig. 21 show, Data Extraction achieved the highest gain at 54.6%, followed by GC Content Analysis (37%) and Coverage Analysis (30.6%). Overall, total execution time was reduced by 48.3%.

TABLE VII. GENOMIC SEQUENCE SYSTEM PERFORMANCE METRICS (MILLISECONDS)

Function	Serial Execuion Time	Parallel Exection Time	Improvement (%)
Data Extraction	118.149	53.670	54.6
GC Content Analysis	63.252	39.827	37.0
Coverage Analysis	1.902	1.320	30.6
Total Execution	183.303	94.820	48.3

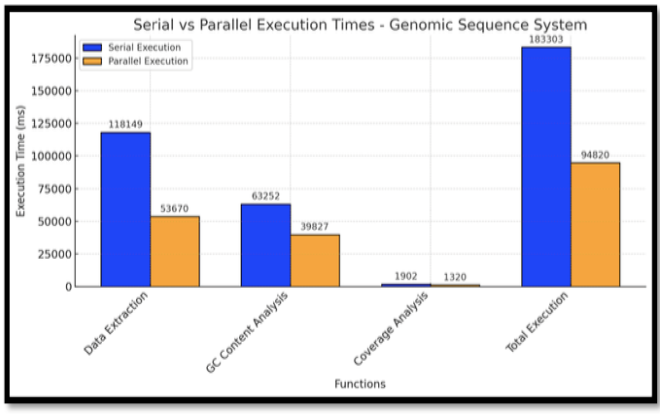

Fig. 20. Serial vs. Parallel execution time (genomic sequence analysis system).

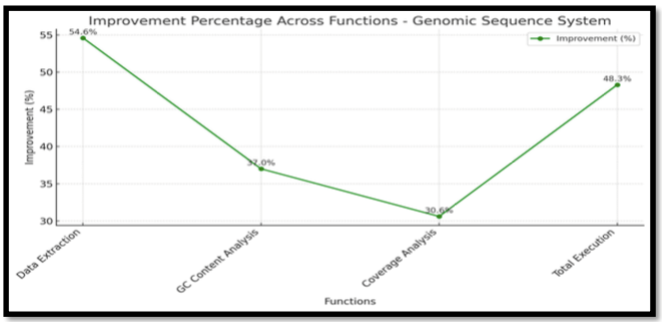

Fig. 21. Improvement percentage across functions (genomic sequence analysis system).

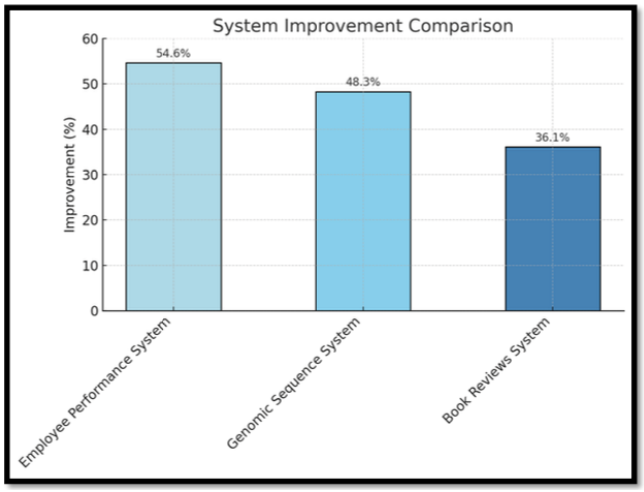

Fig. 22. System improvement comparison.

Fig. 22 presents a bar chart that shows a final comparison of overall system performance improvements achieved through parallelization. The Employee Performance System showed the highest gain at 54.6%, followed by the Genomic Sequence System at 48.3%, while the Book Reviews System achieved a more modest improvement of 36.1%. This confirms the tool's effectiveness across varied application domains. Table VIII provides a comparative overview of our tool and several related tools. The comparison is based on key attributes like parallelization method, analysis type (static or runtime), and language support. [9][12][13].

TABLE VIII. COMPARISON OF OUR TOOL WITH RELATED TOOLS ACROSS VARIOUS ATTRIBUTES

Work	Lazy-Parallel Function Calls	Automatic Parallelization of Pure Method Calls	Fine-Grained Recursive Parallelim	Autopar	OurTool
Languag Focus	Java, C#	Java	C	ANSI C Codes	C++
Optimization Target	Lazy evaluation, closures, nano-threads	Pure method calls	Special Recursive Function Calls	Special Recursive Function Calls	C++ source code, Function calls based on three fundament-al analyses
Automation Level	Partial; compiler-assisted	automatic, compiler-based	automatic	automatic	Fully automated pipeline: detects, classifies, and transforms suitable function calls to parallel equivalents.

VII. CONCLUSION

This paper presents a tool that successfully automates function-level parallelization in C++ through a pipeline that includes dependency analysis, context analysis, and workload assessment. These analyses enable the identification of independent, high-workload function calls that can be safely parallelized without modifying the original program logic. The tool also generates detailed Control Flow Graphs (CFGs) for each function in DOT, PDF, and TXT formats, offering valuable insights into code structure and parallelization opportunities.

Experimental evaluations on three benchmark programs demonstrated consistent improvements in execution efficiency. Performance gains were particularly notable in compute-intensive functions such as data extraction and analysis. The Employee Performance System achieved the highest

improvement at 54.6%, followed by the Genomic Sequence System at 48.3%, and the Book Reviews System at 36.1%. These results confirm the effectiveness of the tool's analysis and transformation strategies and underscore its contribution to advancing practical, automated parallelization in performance-critical C++ applications.

Future work may focus on extending the tool's capabilities to support parallelization of additional code constructs and the use of MPI, enabling distributed-memory execution and scalability across multiple nodes. Such an extension would broaden the tool's applicability beyond shared-memory systems.

REFERENCES

[1] R. C. O. Rocha, L. F. W. Góes, and F. M. Q. Pereira, "Automatic parallelization of recursive functions with rewriting rules," *Sci. Comput. Program.*, vol. 173, pp. 128–152, 2019.

[2] C. Navarro, N. Hitschfeld, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using GPU architectures," *Commun. Comput. Phys.*, vol. 15, pp. 285–329, 2013.

[3] J. Parkhurst, J. Darringer, and B. Grundmann, "From single core to multi-core: Preparing for a new exponential," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, 2006, pp. 67–72.

[4] D. Geer, "Industry trends: Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[5] K. Sujatha, P. V. N. Rao, A. A. Rao, V. G. Sastry, V. Praneeta, and R. K. Bharat, "Multicore parallel processing concepts for effective sorting and searching," in *Proc. PACES-2015, Dept. of ECE, KL Univ.*, 2015.

[6] M. Sohal and R. Kaur, "Automatic parallelization: A review," *Int. J. Comput. Sci. Mobile Comput. (IJCSMC)*, vol. 5, no. 5, pp. 17–21, 2016.

[7] J. Kwiatkowski and R. Iwaszyn, "Automatic program parallelization for multicore processors," in *Lecture Notes in Computer Science*, vol. 6081, pp. 236–245, 2010.

[8] A. Barve, S. Khomane, B. Kulkarni, S. Ghadage, and S. Katare, "Parallelism in C++ programs targeting objects," in *Proc. Int. Conf. Adv. Comput., Commun. Control (ICAC3)*, 2017, pp. 1–6.

[9] R. Surendran and V. Sarkar, "Automatic parallelization of pure method calls via conditional future synthesis," Rice Univ., Houston, TX, USA, 2011.

[10] A. Midolo and E. Tramontana, "An automatic transformer from sequential to parallel Java code," Univ. of Catania, Italy, 2023.

[11] G. Kusoglu, B. Bramas, and S. Genaud, "Automatic task-based parallelization of C++ applications by source-to-source transformations," in *Compas 2020: Parallélisme/Architecture/Système/Temps Réel*, Lyon, France, 2020.

[12] D. Saougkos, A. Mastoras, and G. Manis, "Fine-grained parallelism in recursive function calls," Univ. of Ioannina, Greece, 2012.

[13] M. E. Kalender, C. Mergenci, and O. Ozturk, "AutopaR: An automatic parallelization tool for recursive calls," Dept. Comput. Eng., Bilkent Univ., Turkey, 2014.

[14] M. Mathews and J. P. Abraham, "Implementing coarse-grained task parallelism using OpenMP," Dept. Comput. Sci. Eng., Mar Athanasius Coll. of Eng., India, 2015.

[15] P. Mehta, S. Singh, D. Roy, and M. M. Sarma, "Comparative study of multi-threading libraries to fully utilize multi-processor/multi-core systems," *Int. J. Current Eng. Technol.*, 2014.

[16] S. S. Chatterjee and R. Gururaj, "Lazy-parallel function calls for automatic parallelization," in *Proc. Int. Conf. Comput. Intell. Inf. Technol. (CIIT)*, vol. CCIS 250, pp. 811–816, Springer, 2011.

[17] E. Pontelli and G. Gupta, "Last parallel call optimization and its generalization," *J. Logic Program.*, vol. 27, no. 1, pp. 1–43, 1996.