

**Implementation of Computational Methods.**

**EVIDENCE 2: PARALLEL SYNTAX HIGHLIGHTER.**

We used the same syntax highlighter from the previous evidence, a lexer program for the C++ programming language. A lexer is a program that reads code and breaks it into tokens (small meaningful parts like keywords, numbers, strings, or punctuation).

It is usually the first step in a compiler or interpreter, before it goes to a parser (another program that takes as input the tokens generated previously by the lexer). The difference with a parser is that this one verifies if the written code makes sense or not, so that the statements received do an instruction for the computer, and then it goes to a compiler.

That project was already successfully developed. The new challenge is to call that function for a set of many C++ source files, to analyze them syntactically in both ways: sequence and parallel.

A parallel lexer is a type of lexical analyzer designed to tokenize (break down) text input using multiple processing threads or cores at the same time. Unlike traditional lexers that process input sequentially, a parallel lexer divides the workload to improve performance, especially for large inputs or high-throughput systems. The way it works is:

- Input Splitting: The input text is divided into chunks. This can be done by size, line, or logical boundaries (like code blocks).
- Parallel Tokenization: Each chunk is processed by a separate thread or processor, using regular expressions or finite automata to identify tokens.
- Synchronization: After tokenizing, the results are merged in the correct order to produce the final token stream.

This new program is better, because it's faster considering large inputs, and it scales well with modern multi-core CPUs.

Our proposed solution was to generate a Racket algorithm that analyzes C++ source files, extracting tokens from each file using a sequential lexer. The main program includes both sequential and parallel execution modes, and it measures the execution time of both for comparison.

The control flow of this project is divided into 4 functions:

- ❖ analyze-c++-code: Tokenizes a given C++ code snippet using tokenize-lines.
- ❖ sequential-analysis: Analyzes all code snippets one by one (sequentially). It uses map to apply the analysis function to each file.
- ❖ parallel-analysis: Uses Racket's future and touch to analyze code snippets in parallel. Each snippet is processed in a separate thread-like future.

- ❖ **main:** Defines a list of C++ source file paths. It measures and displays the execution time (CPU and GC time) for both sequential and parallel analysis.

Each rule in the lexer is checked sequentially (top to bottom) until a match is found for the current part of the input. For each character in the input, the lexer tries to match one of the defined regex patterns, once matched, it recursively calls itself with the remaining input. It continues until it reaches the end of the input or fails.

So, this program reads several C++ files, extracts and prints their tokens using a lexer, and compares how long the analysis takes when run sequentially versus in parallel. This helps to demonstrate the performance benefit of parallelism using future.

As in Evidence 1 with the sequential lexer, both sequential and parallelism have the same asymptotic computable complexity:  $O(n)$ , where  $n$  is the input stream of C++ code to analyze. In sequential-analysis, the function processes each snippet one after another; map performs  $n$  independent calls to `analyze-c++-code`, so total time is proportional to  $n$ . In parallel-analysis, each snippet is analyzed in a separate future; this still performs  $n$  analyses, but in parallel.

The main difference is execution time, not complexity. Parallelism reduces the execution time, compared with the sequential model, especially on multi-core CPUs, but not theoretical complexity. This is demonstrated with the following results, which are the output of our parallel lexer program:

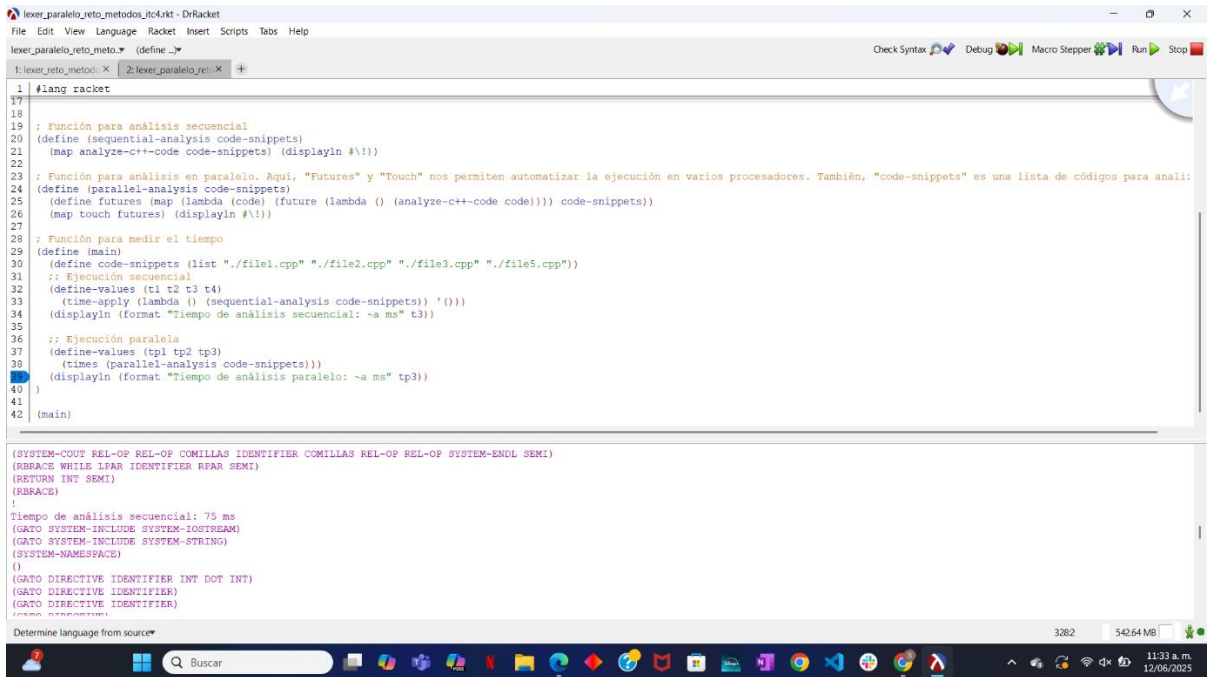
1. Average execution time of the sequential program: 75 ms.
2. Average execution time of the parallel program: 15 ms.

Beyond the computable implications, the development of innovative technologies could have some important ethical implications for society. In the previous evidence, multiple ethical impacts were presented. For parallelism, I would add the following ones:

- **Malicious Code Analysis and Privacy:** A parallel lexer can quickly analyze large volumes of code to detect vulnerabilities or malware. The same tool could be used to scan proprietary or sensitive source code without consent.
- **Environmental Impact:** Parallel lexers leverage multiple cores or machines, increasing CPU usage, which can lead to higher energy consumption and a greater carbon footprint, especially when deployed at scale (e.g., in data centers).
- **Education and Accessibility:** Parallel lexers could improve compiler education by enabling real-time feedback on large codebases. But if the technology is proprietary or hard to use, it may widen the gap between resource-rich institutions and under-resourced communities.
- **Dual-Use Technology:** Like many tools, a parallel lexer is dual-use, with legitimate uses such as in compilers and tools for static analysis; or harmful uses such as in reverse engineering, scanning stolen data bases, or the development of malware.

In conclusion, it is important that we take advantage of new upgraded tools and learn from them, considering all their power and new features to be implemented in our studies, but we must never use them improperly. Also, new computational developments such as parallelism seem to be fantastic, but we should always take into consideration the bases, such as in the sequential model, from the basic to the advanced, to fully understand this kind of tool.

## Program execution for the sequential lexer:



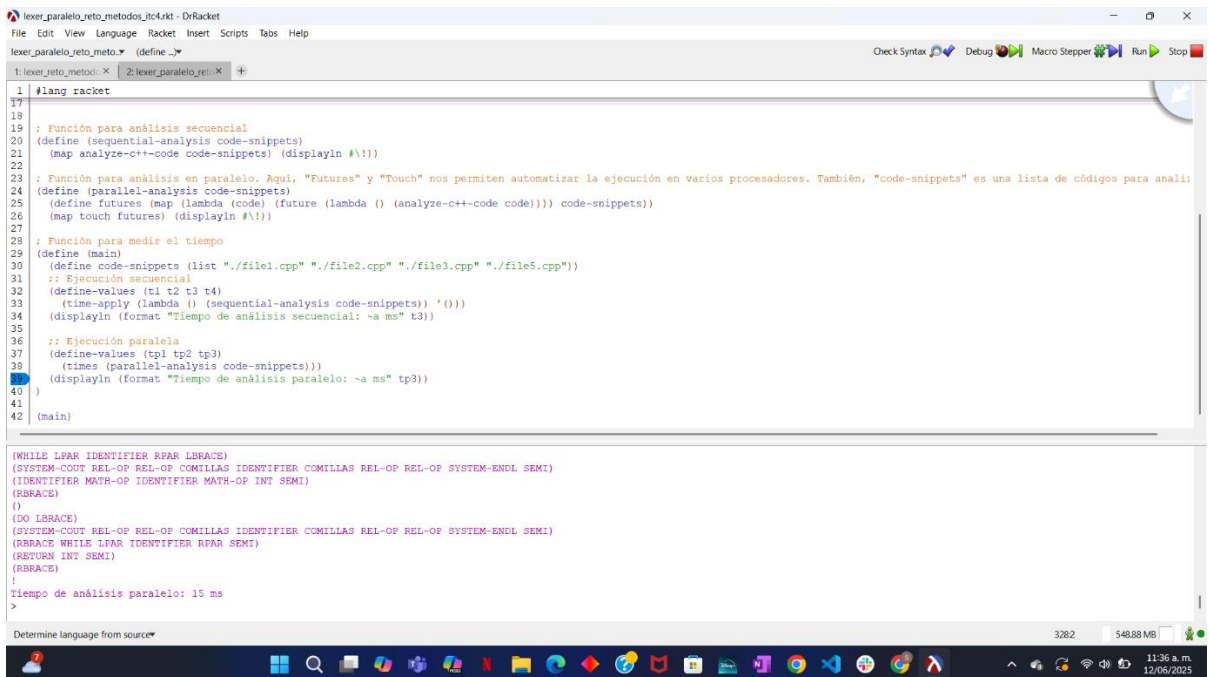
```
1 #lang racket
17
18
19 ; Función para análisis secuencial
20 (define (sequential-analysis code-snippets)
21   (map analyze-c++-code code-snippets) (displayln #\!))
22
23 ; Función para análisis en paralelo. Aquí, "Futures" y "Touch" nos permiten automatizar la ejecución en varios procesadores. También, "code-snippets" es una lista de códigos para analizar.
24 (define (parallel-analysis code-snippets)
25   (define futures (map (lambda (code) (future (lambda () (analyze-c++-code code)))) code-snippets))
26   (map touch futures) (displayln #\!))
27
28 ; Función para medir el tiempo
29 (define (main)
30   (define code-snippets (list "./file1.cpp" "./file2.cpp" "./file3.cpp" "./file5.cpp"))
31   ;; Ejecución secuencial
32   (define-values (t1 t2 t3 t4)
33     (time-apply (lambda () (sequential-analysis code-snippets)) '()))
34   (displayln (format "Tiempo de análisis secuencial: ~a ms" t3))
35
36   ;; Ejecución paralela
37   (define-values (tp1 tp2 tp3)
38     (times (parallel-analysis code-snippets)))
39   (displayln (format "Tiempo de análisis paralelo: ~a ms" tp3))
40 )
41
42 (main)
```

(SYSTEM-COUT REL-OP REL-OF COMILLAS IDENTIFIER COMILLAS REL-OP REL-OF SYSTEM-ENDL SEMI)  
(RBRACE WHILE LPAR IDENTIFIER RPARG SEMI)  
(RETURN INT SEMI)  
(RBRACE)  
(  
Tiempo de análisis secuencial: 75 ms  
(GATO SYSTEM-INCLUDE SYSTEM-IOSTREAM)  
(GATO SYSTEM-INCLUDE SYSTEM-STRING)  
(SYSTEM-NAMESPACE)  
(  
(GATO DIRECTIVE IDENTIFIER INT DOT INT)  
(GATO DIRECTIVE IDENTIFIER)  
(GATO DIRECTIVE IDENTIFIER)  
(GATO DIRECTIVE)  
)

Determine language from source▼

3262 542.64 MB 11:33 a.m. 12/06/2025

## Program execution for the parallel lexer:



```
1 #lang racket
17
18
19 ; Función para análisis secuencial
20 (define (sequential-analysis code-snippets)
21   (map analyze-c++-code code-snippets) (displayln #\!))
22
23 ; Función para análisis en paralelo. Aquí, "Futures" y "Touch" nos permiten automatizar la ejecución en varios procesadores. También, "code-snippets" es una lista de códigos para analizar.
24 (define (parallel-analysis code-snippets)
25   (define futures (map (lambda (code) (future (lambda () (analyze-c++-code code)))) code-snippets))
26   (map touch futures) (displayln #\!))
27
28 ; Función para medir el tiempo
29 (define (main)
30   (define code-snippets (list "./file1.cpp" "./file2.cpp" "./file3.cpp" "./file5.cpp"))
31   ;; Ejecución secuencial
32   (define-values (t1 t2 t3 t4)
33     (time-apply (lambda () (sequential-analysis code-snippets)) '()))
34   (displayln (format "Tiempo de análisis secuencial: ~a ms" t3))
35
36   ;; Ejecución paralela
37   (define-values (tp1 tp2 tp3)
38     (times (parallel-analysis code-snippets)))
39   (displayln (format "Tiempo de análisis paralelo: ~a ms" tp3))
40 )
41
42 (main)
```

(WHILE LPAR IDENTIFIER RPARG LBRACE)  
(SYSTEM-COUT REL-OP REL-OF COMILLAS IDENTIFIER COMILLAS REL-OP REL-OF SYSTEM-ENDL SEMI)  
(IDENTIFIER MATH-OP IDENTIFIER MATH-OP INT SEMI)  
(RBRACE)  
(  
(DO LBRACE)  
(SYSTEM-COUT REL-OP REL-OF COMILLAS IDENTIFIER COMILLAS REL-OP REL-OF SYSTEM-ENDL SEMI)  
(RBRACE WHILE LPAR IDENTIFIER RPARG SEMI)  
(RETURN INT SEMI)  
(RBRACE)  
)  
Tiempo de análisis paralelo: 15 ms  
>

Determine language from source▼

3262 548.88 MB 11:36 a.m. 12/06/2025