A Lightweight Static Data Race Checker for OpenMP Software

Hend Alshede

King Saud University
Riyadh, Saudi Arabia

Abstract

OpenMP has become one of the most widely adopted interfaces for parallel programming, particularly for managing shared-memory parallelism on multi-core architectures. With the increasing prevalence of multi-core systems, a large body of sequential software has been parallelized using OpenMP. However, writing correct parallel programs remains challenging, and concurrency errors such as data races and deadlocks are common. This paper presents the design and conceptual evaluation of a lightweight static data race checker for OpenMP programs. The proposed approach relies on lexical, syntactic, and semantic analysis performed at compile time, without requiring runtime instrumentation. By focusing on simplicity and early error detection, the checker aims to provide programmers with fast and useful feedback about potential data races. Compared to dynamic approaches, the proposed method can identify likely race conditions earlier in the development process with lower analysis overhead.

Keywords:
OpenMP; data race detection; Parallel programming; High-Performance Computing; Static analysis.

1. Introduction

High-Performance Computing (HPC) has become essential in many scientific, engineering, and industrial domains. The rapid development of powerful computing platforms has made large-scale parallel systems increasingly accessible, and exascale computing is expected to become feasible soon [1]. Consequently, the importance of efficient parallel software development continues to grow. Despite advances in hardware, developing correct and efficient parallel software remains difficult. Traditional programming languages provide limited native support for parallelism, which has led to the adoption of programming models that enable parallel execution. Programming models define a set of abstractions, operations, and execution rules that facilitate parallel computation [2]. Among the most widely used models are Message Passing Interface (MPI) for distributed memory systems and OpenMP for shared-memory parallelism [3]. In addition, modern programming models increasingly support heterogeneous systems, including accelerators such as GPUs [4].

OpenMP is a standard programming model designed for shared-memory parallel programming in C, C++, and Fortran. Since its first release in 1997, OpenMP has evolved significantly, with version 5.0 introducing enhanced support for complex architectures and execution environments [5]. OpenMP offers portability, scalability, and ease of use, allowing programmers to incrementally parallelize existing code.

However, OpenMP does not inherently prevent concurrency errors. In particular, data races remain a major source of incorrect behavior in parallel programs. Ensuring race-free execution is largely the programmer's responsibility, which increases the likelihood of subtle and hard-to-debug errors in HPC applications.

Although several static and dynamic tools have been proposed to detect data races in OpenMP programs, many existing approaches either rely on runtime instrumentation or involve complex analysis frameworks that increase overhead. This paper addresses this gap by proposing a lightweight static data race detection approach that focuses on early error identification at compile time. The proposed method emphasizes simplicity and conceptual clarity, making it suitable as a design foundation for practical static analysis tools.

The remainder of this paper is organized as follows. Section 2 provides background on OpenMP and data races. Section 3 reviews related work. Section 4 presents the design of the proposed static data race checker. Section 5 provides a conceptual

Manuscript received January 5, 2026
Manuscript revised January 20, 2026
https://doi.org/10.22937/IJCSNS.2026.26.1.13

comparative study. Finally, Section 6 concludes the paper and outlines future work.

2. Background

OpenMP is commonly used in shared-memory multiprocessing systems, where all threads access a common memory space. Each thread executes the same program independently and may progress at different execution stages, in contrast to the Single Instruction Multiple Data (SIMD) execution model. Synchronization in OpenMP is achieved through constructs such as barriers, critical sections, atomic operations, and locks.

Although OpenMP provides several synchronization mechanisms, correct usage remains the programmer's responsibility. A data race occurs when two or more threads access the same memory location concurrently and at least one of the accesses is a write operation. Data races often arise in parallel loops, where different iterations execute simultaneously and access shared data. Figure 1 illustrates a simple example of a data race in an OpenMP parallel loop. In this example, multiple threads access overlapping memory locations, leading to undefined program behavior. Such errors are difficult to detect through testing alone, especially in large-scale HPC applications.

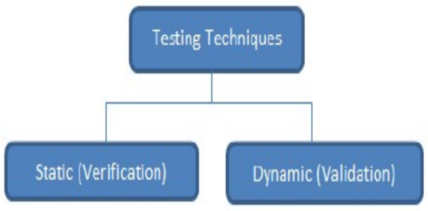


Figure 1. Categories of testing techniques.

Software testing techniques can be broadly classified into static and dynamic approaches. Static testing is performed at compile time without executing the program and aims to identify defects early in the development process. In contrast, dynamic testing involves executing the program with specific inputs and analyzing runtime behavior. While dynamic techniques can provide precise information, they often incur significant runtime overhead and may fail to cover all execution paths.

3. Related work

Numerous approaches have been proposed to detect data races in OpenMP and parallel programs. Atzeni et al. [6] introduced Archer, a hybrid approach combining static and dynamic analysis. Archer classifies program regions as race-free, certainly racy, or potentially racy, and applies runtime analysis only to ambiguous regions. Although effective, this approach still relies on runtime instrumentation. Figure 2 illustrates Archer architecture.

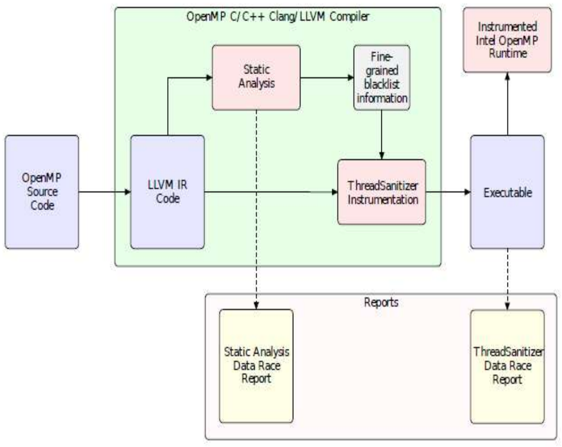
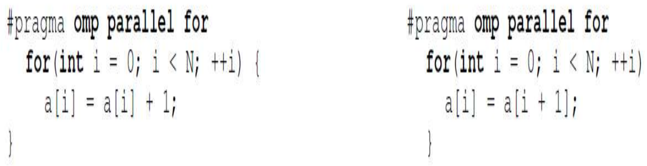

OpenMP C/C++ Clang/LLVM Compiler
Static Analysis
Fine-grained blacklist information
Instrumented Intel OpenMP Runtime
OpenMP Source Code
LLVM IR Code
ThreadSanitizer Instrumentation
Executable
Reports
Static Analysis Data Race Report
ThreadSanitizer Data Race Report

Figure 2. Archer architecture.

The input in their module is a source code file of an OpenMP program. The output is a report of the possible data race part. They called their module "Archer". Archer used a set of static techniques to determine the categories of code as one of three: race-free regions, certainly racy regions, and potentially racy regions. Figure 3a shows a simple OpenMP code where the Archer will determine it as race-free, while Figure 3b shows certainly racy regions. The dynamic analyzer is used to detect potentially risky regions and to check the code at runtime for a more accurate diagnosis.



```
#pragma omp parallel for
  for(int i = 0; i < N; ++i) {
    a[i] = a[i] + 1;
}
```

```
#pragma omp parallel for
  for(int i = 0; i < N; ++i)
    a[i] = a[i + 1];
}
```

Figure 3. OpenMP parallel for loop.

On the other hand, Swain, B., and Huang, J., in [7] (2018), presented an incremental approach for checking data races in OpenMP. Their proposed approach was static analysis and consisted of two steps:

(i): Array Index checking step used to check if two arrays made from different processors may access the same location in memory.

(ii): An Incremental Graph step used to determine the possibility of overlapping occurring concurrently by building a May Happen in Parallel (MHP) graph.

The result has shown that the first part, which is array index analysis on its own, is very useful for checking data races. The second part, a graph step, kept the runtime low through incremental updates.

As well as in [8] (2020), BORA, U et al., proposed the LLOV module. It was a static approach to check data race for OpenMP programs. It was built using the framework of the LLVM compiler. Attract as a fast, lightweight, language-agnostic module. When comparing LLOV with other existing data race checkers, the result shows that LLOV provides the same percentage of precision and accuracy while being faster. LLOV can verify a C++ or FORTRAN code only. The architecture of LLOV is shown in Figure 4 below.

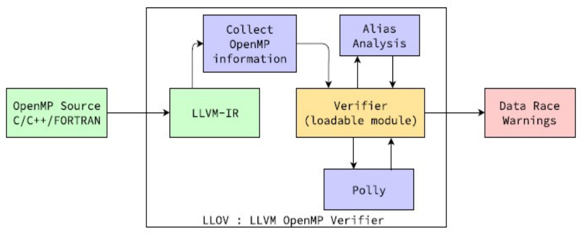

Collect OpenMP information
Alias Analysis
OpenMP Source C/C++/FORTRAN
LLVM-IR
Verifier (loadable module)
Data Race Warnings
Polly
LLOV : LLVM OpenMP Verifier

Figure 4. The architecture of LLOV.

LLOV has two main steps: analysis, then verification.

1) Analysis step. It collects different OpenMP pragmas and other information needed for race checking. The importance of this step is to constructs are lowered. LLVM-IR is sequential and does not have support for parallel constructions.

2) Verification step. LLOV checks the regions for data races only marked parallel by one of the structured parallelism. A crucial property of OpenMP constructs is that its specification allows only structured blocks within a pragma. A structured block must not contain arbitrary jumps into or out of it. In other words, a structured block closely resembles a Single-Entry Single Exit (SESE) region used in loop analyses.

At the end of this section, Table 1 summarizes the four reviewed papers, while Table 2 compares their approaches based on the main attributes.

Table 1. Summary of related work

Study Reviewed	Year	Summary Note	Notes
Atzeni, S et.al. in [6]	2013	Static and dynamic approach	Archer module.
Swain, B., and Huang, J in [7]	2018	Static approach	Incremental module
BORA, ,U et.al,[8]	2020	Static approach	LLOV module

Table 2. Related work comparative study

Study Reviewed	Accuracy	Complexity	Time
Atzeni, S et.al. in [6] (2013),	High	High	Fast
Swain, B., and Huang, J in [7] (2018),	High	High	Fast
BORA, ,U et.al, [8] 2020	High	High	Very Fast

4. Static data race checker

The proposed static data race checker analyzes OpenMP programs written in C++ at compile time. Figure 5 illustrates the overall architecture of the proposed tool.

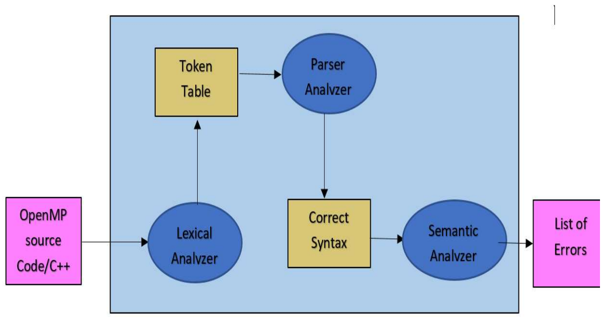


Figure 5. The architecture of a static data race checker.

The analysis process consists of three main stages:

1. **Lexical Analysis:**

   The input source code is scanned and transformed into a sequence of tokens. Comments and whitespace are removed, and relevant lexical elements are identified.

2. **Parsing Analysis:**

   The parser validates the token stream against the grammar rules of the language and constructs a structured representation of the program.

3. **Semantic Analysis:**

   The semantic analyzer examines program constructs to identify parallel regions and loop structures. Shared memory accesses within parallel loops are analyzed to detect potential data race conditions.

When a potential data race is detected, the tool reports the corresponding source code location. If no race conditions are found, the program is reported as race-free within the analyzed scope.

The flowchart that represents the algorithm to detect the data race error inside openMP program is represented in the following Figure 6.

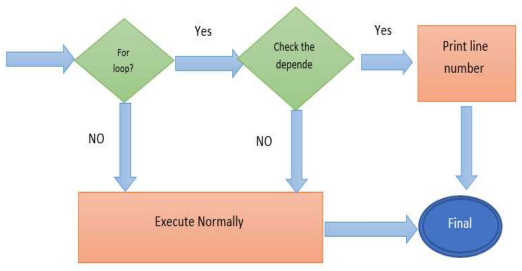


Figure 6. The Flowchart algorithm for the data race checker.

The proposed pseudo-code outlines the conceptual logic used to track memory accesses and identify conflicting read/write operations in parallel regions.

```
Var G_address
Int Flage_Error =0
while ( # progrma openmp parallel )
{Scan (for)?
If write, then
G_address = current address
  If  (read  or  write)  and  address  =  G_address, then
Data race error in # line
Flage_Error =1
End if
G_address=0
End if
}
If Flage_Error =0 then
No data race
End if
End
```

5. Comparative Study

Data race detection tools can be evaluated based on complexity, analysis time, and accuracy. Dynamic approaches typically provide high accuracy but incur significant runtime overhead. Hybrid approaches reduce overhead but increase system complexity. Static approaches, while potentially conservative, offer faster analysis and early feedback. The proposed checker adopts a static approach to balance simplicity and efficiency. Rather than competing with advanced compiler-level tools, the design serves as a lightweight alternative that can complement existing methods or act as a foundation for further development.

6. Conclusion

OpenMP has become a standard choice for parallel programming on shared-memory systems, but data races remain a significant challenge. This paper presented the design and conceptual evaluation of a lightweight static data race checker for OpenMP programs. By performing analysis at compile time and avoiding runtime instrumentation, the proposed approach can provide early feedback to developers with minimal overhead. Future work will focus on implementing the proposed checker, extending its analysis to support additional OpenMP constructs, and evaluating its effectiveness using standard benchmark suites such as the NAS Parallel Benchmarks.

References

[1] Alghamdi, A and Eassa, F " Software Testing Techniques for Parallel Systems: A Survey ", IJCSNS International Journal of Computer Science and Network Security,2019.

[2] Jianjiang, L et al.," Analysis of Factors Affecting Execution Performance of OpenMP Programs ", TSINGHUA SCIENCE AND TECHNOLOGY, ISSN 1007-0214 05/21 pp304-308Volume 10, Number 3, June 2005.

[3] Ma, H et al., " Symbolic Analysis of Concurrency Errors in OpenMP Programs", 42nd International Conference on Parallel Processing, 2013.

[4] Kirk, D., and Hwu, W., Programming Massively Parallel Processors, Morgan Kaufmann, 2016.

[5] OpenMP Architecture Review Board, OpenMP Application Programming Interface Version 5.0, 2018.

[6] Atzeni, S et al., " Archer: A Low Overhead Data Race Detector for OpenMP", School of Computing – University of Utah, 2013.

[7] Swain, B and Huang, J," Towards Incremental Static Race Detection in OpenMP Programs", IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness), 2018.

[8] BORA, U et al., " LLOV: A Fast Static Data-Race Checker for OpenMP Programs", ACM Transactions on Architecture and Code Optimization, Vol. 17, No. 4, Article 35. Publication date: November 2020.

[9] Anwar, N and Kar, S " Review Paper on Various Software Testing Techniques & Strategies ", Double Blind Peer-Reviewed International Research Journal, Vol. 19, 2019.

Hend Alshede was born in Riyadh, Saudi Arabia. She received the B.Sc. degree in computer science and education from Princess Nora Bint Abdulrahman University, Riyadh, Saudi Arabia, in 2007, the M.Sc. degree in sciences in computer science from King Saud University, Riyadh, Saudi Arabia, in 2013, and PhD degree in computer science from King Abdulaziz University, Jeddah, Saudi Arabia, in 2025. She is currently working as an Assistant Professor at King Saud University. His research interests include network security, Internet of Things, cybersecurity, smart grid, machine learning, and deep learning.