Automatic Parallelization of Loops Using Compiler-Based Techniques

Samaa Ahmed (211001868)
S.Ahmed2168@nu.edu.eg

Mohamed Abdelnaser Saied (211001159)
M.Abdelnaser2159@nu.edu.eg

Ganna Ayman Elroby (211002034)
G.ayman2134@nu.edu.eg

Sara Abdelnasser Ahmed (211001619)
S.Abdelnasser2119@nu.edu.eg

Hana Nassef (211000128)
H.Mohammed2128@nu.edu.eg

Submitted to:

Dr. Islam Tharwat Abdelhalim , Engr. Bassent Ahmed Farouk

Abstract

The project explores compiler-based automatic parallelization techniques to improve the execution of sequential C/C++ applications on modern multi-core processors. Through the use of tools such as Pluto, Cetus, and AutoPar-Clava, the study explores the application of the polyhedral model in extracting parallelizable loop structures and applying transformations such as loop tiling, fusion, and atomic operations using OpenMP directives. A source-to-source transformation tool was constructed for auto-parallelization of loops. Experimental findings show measurable performance improvement on small benchmarks, indicating the potential and limitations of automatic parallelization for real-world use. The findings motivate further exploration of dynamic adaptation and smart loop transformation techniques for larger, more complex applications.

I. Introduction

While the demand for high-performance computing continues to pervade scientific, engineering, and data-intensive applications, parallel computing has become a critical imperative to achieve scalability and performance on modern multi-core platforms. However, the hand translation of sequential code into efficient parallel equivalents remains an error-prone and cumbersome task. Programmers must infer intricate dependencies, manage thread synchronization, and reorganize code to expose parallelism — all involving non-trivial skill and labor.

To mitigate these challenges, automatic parallelization techniques have gained increasing visibility. These methods aim to transform sequential code into parallel equivalents with minimal programmer effort, enabling broader adoption of parallel computing and reducing development overhead. Among the most successful frameworks enabling such transformations is the polyhedral model, a mathematical abstraction that represents loop nests and their dependencies as geometric objects. By capturing iteration spaces and data access patterns within this model, compilers can apply advanced loop transformations like tiling, fusion, interchange, and skewing — and, in the process, reveal parallelism not immediately apparent in the source code.

Pluto, Cetus, and Polly are some of the most widely used polyhedral-based source-to-source translators. They automatically conduct loop nest restructuring and insert parallel directives (e.g., OpenMP) whenever necessary, ensuring semantic correctness of the program using rigorous dependency analysis. They have enabled the optimization of a large variety of applications for parallel execution, particularly in areas where performance is crucial but where manual parallelization is not viable.

This project explores to what extent the polyhedral model of automatic parallelization can improve the performance of sequential programs executed on multi-core processors. Experiments will be conducted on a collection of benchmark codes, executing both the original sequential versions and their parallelized counterparts generated by tools like Pluto and Cetus. Performance measurements will be made with the help of standard profiling and analysis tools like Gprof and Perf so that close examination can be made on the execution time gains, CPU usage, and the effects of the different loop transformations.

Through critical examination of the results, this research seeks to bring out the strengths, limitations, and practicability in application of the utilization of polyhedral-based automatic parallelization in real-world computer applications

II. problem definition

The increasing prevalence of multi-core and many-core processors has created a pressing need to efficiently exploit hardware-level parallelism in modern software applications. However, most existing and legacy codebases are written in a sequential manner and are thus incapable of leveraging the full computational capabilities of contemporary hardware architectures. Manual parallelization techniques—though capable of delivering high performance—are inherently complex, time-consuming, and error-prone. Developers must have deep expertise in areas such as concurrency control, synchronization mechanisms, memory models, and performance tuning to avoid issues like race conditions, deadlocks, and load imbalance. These challenges make manual parallelization infeasible for large-scale, production-grade software systems and contribute significantly to prolonged development cycles.

Automatic parallelization of loops, particularly using compiler-based techniques, offers a promising solution to this problem by analyzing and transforming sequential loop constructs into their parallel equivalents during compilation, without requiring manual intervention or code restructuring by the programmer. Compiler-based parallelization techniques aim to automatically detect independent loop iterations and apply transformations—such as loop fusion, tiling, data privatization, and reduction handling—to generate parallel code suitable for execution on shared-memory platforms (e.g., using OpenMP) or distributed-memory systems (e.g., using MPI). Despite its potential, current approaches face several limitations. These include difficulty in accurately analyzing complex loop

dependencies, integrating multiple transformations in a unified optimization framework, handling pointer aliasing and dynamic control flow, and adapting to irregular or large-scale codebases. Furthermore, most traditional techniques rely on static heuristics or rule-based systems, which may not generalize well across diverse applications or deliver optimal performance on different hardware targets.

Recent advancements, such as AutoPar-Clava—a source-to-source tool for OpenMP parallelization and polyhedral model-based compilers, have demonstrated success in static loop analysis and code transformation. In parallel, emerging machine learning-driven methods like MPIrigen illustrate the potential for data-driven approaches to assist in automatic MPI code generation. Nonetheless, a significant gap remains in delivering robust, scalable, and intelligent compiler-based frameworks that can generalize across domains, reduce developer burden, and produce performance-portable parallel code. Addressing this gap remains a key research challenge in the field of automatic parallelization.

III. PROPOSED EVALUATION

To effectively evaluate the benefits and performance of the proposed compiler-based auto-loop parallelization, we will adopt a structured evaluation methodology focused on benchmarking, performance metrics, and comparison with existing techniques.

A. Benchmark Selection

Appropriate benchmarks will be selected to assess the effectiveness of the compiler-based parallelization across various loop structures, data access patterns, and problem sizes. The chosen benchmarks will represent a wide range of computational complexities to evaluate the scalability and efficiency of the parallelization approach. These benchmarks will include both synthetic and real-world applications to ensure comprehensive testing.

B. Performance Metrics

We will use the following performance metrics to quantify the improvements resulting from parallelization:

- Execution Time: Measure the total time taken by the benchmarks to execute before and after parallelization.

$$T_{\text{seq}} = \text{Time taken by the sequential version}$$

$$T_{\text{par}} = \text{Time taken by the parallel version}$$

- Speedup Ratio: Calculate the speedup by comparing the execution times before and after applying parallelization.

$$\text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{par}}}$$

- Efficiency: Evaluate the efficiency of resource utilization by measuring the ratio of speedup to the number of cores used.

$$\text{Efficiency} = \frac{\text{Speedup}}{P} = \frac{T_{\text{seq}}}{P \times T_{\text{par}}}$$

- Scalability: Assess the performance of the parallelized application as the number of processor cores increases.

$$\text{Speedup}(P) = \frac{T_{\text{seq}}}{T_{\text{par}}(P)}$$

C. Environment Testing

We will ensure a controlled testing environment to maintain consistency and isolate the impact of the parallelization technique:

- Hardware Consistency: All tests will be conducted on the same hardware configuration to attribute performance differences solely to the software changes.

- Software Environment: The operating system and supporting software will remain consistent, and background processes will be minimized to avoid interference with the performance measurements.

D. Iterative Testing

Multiple rounds of testing will be conducted to account for variability in performance, such as differences in memory usage patterns or processor thermal state. This iterative approach will help refine the parallelization techniques based on empirical performance data, ensuring more reliable results.

E. Comparative Analysis

The results will be compared with:

- Non-parallelized Code: Establishing a baseline by testing the application's performance before parallelization.

- Manually Parallelized Code: If applicable, the performance of the auto-parallelized code will be

compared with a version of the application that has been manually parallelized, helping to assess the effectiveness and accuracy of the automatic parallelization approach.

F. Code Quality

The quality of the parallelized code will also be evaluated. This will include analyzing aspects such as data dependency management, loop transformations, and the synchronization techniques introduced by the compiler. These factors are crucial in ensuring that the parallelization does not introduce inefficiencies, such as excessive synchronization overhead or incorrect data handling.

IV. RELATED WORK

In the last few years, significant progress has been made in parallelizing sequential programs automatically to exploit the newer multi-core architectures. This section covers two such notable pieces of work that utilize compiler-based and AI-augmented strategies for automatic loop parallelization in C/C++ programs.

In [1], the authors present **OMPAR**, a new AI-based source-to-source compilation framework that annotates C/C++ code with OpenMP directives to facilitate parallel execution. Aware of the difficulties of manual parallelization—software complexity and architectural diversity—OMPAR proposes an LLM-based two-phase pipeline. The first phase, **OMPIFY**, detects loops with great potential for parallelization. The second, **MONOCODER-OMP**, is a fine-tuned LLM that produces correct OpenMP pragmas from OMPIFY analysis. The system is shown to be resilient even for partial or unstructured code, which makes it more flexible than conventional rule-based compilers. Experimental evaluation shows OMPAR to be better and more precise than existing tools such as AutoPar and the Intel ICC compiler, with notable improvements in parallelizable loop identification and effective directive insertion. OMPAR also has the benefit of continuous learning, allowing it to get better at handling new code patterns over time.

In [2], the authors introduce AutoPar-Clava, an automatic source-to-source parallelizing tool that generates OpenMP-based code for C programs.
Applied atop the Clava compiler infrastructure, AutoPar-Clava employs static analysis methods to find parallelizable loops and employs several optimizations including scalar and array privatization, reduction clause generation, and symbolic data dependency analysis. Complex reduction patterns are supported by the system and it employs the Omega test for data dependency analysis, which enables it to parallelize more loops than other similar systems.

AutoPar-Clava was evaluated on standard benchmarks, including NAS and Polyhedral suites, and achieved performance results comparable to or exceeding those of state-of-the-art auto-parallelization compilers such as ROSE, TRACO, CETUS, and ICC. Its architecture emphasizes preserving the structure of the original code while seamlessly integrating OpenMP directives. These two pieces of work embody complementary yet distinct directions in automatic loop parallelization: OMPAR pushes the frontier of AI-powered compiler improvements, whereas AutoPar-Clava relies on static, conservative compiler methods based on classical program analysis. Jointly, they offer profitable insight into both the strengths and weaknesses of current auto-parallelization tools and form the core reference points for our project, "Automatic Parallelization of Loops Using Compiler-Based Techniques." **.**

V. PROPOSED SOLUTION

The proposed solution is a fully automated source-to-source transformation tool implemented in C++ that parallelizes loops by injecting OpenMP directives into existing sequential codebases. Its primary goal is to convert a given C++ source file into an equivalent version that leverages multi-threading on modern multi-core processors, thereby improving execution performance without requiring manual rewriting of loops or in-depth knowledge of parallel programming from developers.

This tool statically analyzes the source code to identify `for` loops that are suitable for parallel execution. It performs a thorough scan of the code to detect variables involved in assignments outside loops, which are potential shared variables inside loops. By detecting these shared variables, the program can appropriately insert OpenMP `#pragma omp atomic` directives to ensure safe concurrent access and prevent race conditions during parallel execution. Additionally, the tool injects `#pragma omp parallel for` directives before loops to distribute iterations across threads, thus exploiting hardware parallelism efficiently. This automation minimizes developer effort, reduces errors associated with manual parallelization, and maintains the original structure and readability of the source code. The output source file includes the necessary OpenMP header inclusion and is ready for compilation with an OpenMP-enabled compiler, making parallel execution seamless and accessible.

Detailed Algorithm for Auto-Parallelization:

A. Input Code Reading:

- The program prompts the user for the input and output filenames.
- It reads the entire input C++ source file into memory, storing each line in a vector to preserve original formatting.

B. Variable Extraction Outside Loops:

- The program iterates through the code lines prior to any detected loops.
- For each line, it searches for assignment operations by matching known assignment operators (e.g., `=`, `+=`, `-=`).
- It extracts the variable name on the left-hand side of these assignments and stores them in a set, representing variables declared or assigned outside loops that might be shared inside loops.

C. For Loop Detection and Directive Insertion:

- The program scans for lines that start a `for` loop.
- Upon detection, it inserts the OpenMP directive #pragma omp parallel for immediately before the loop to enable automatic parallelization of loop iterations by OpenMP threads.
- An internal flag marks that the program is now inside a loop for subsequent analysis.

D. Shared Variable Identification and Atomic Directive Injection:

- While inside a loop, the program examines each line for occurrences of previously identified shared variables.
- If such a variable appears in an assignment operation, it is a potential data race point.
- The program inserts a `#pragma omp atomic` directive immediately before the assignment line to guarantee atomicity of the update, preventing race conditions.
- It counts the number of shared variable assignments to handle multiple atomic insertions if needed.

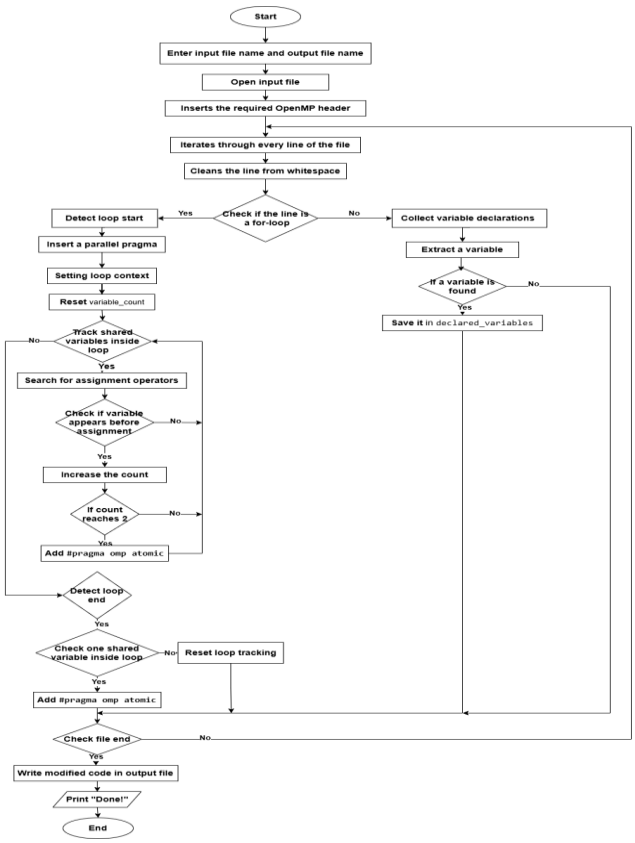
2. **Loop End Detection and State Reset:**

- When a closing brace `}` corresponding to the end of the `for` loop is encountered, the program resets loop-related flags and counters.
- If only one shared variable was detected, an atomic directive is ensured.
- This step finalizes the loop processing before continuing to scan the remainder of the code.

3. **Output Code Generation:**

- The transformed source code is written to the specified output file.
- The output includes the original code plus injected OpenMP directives and the necessary `#include <omp.h>` header at the beginning.

Flow Chart of the code:



VI. Evaluation and results

The performance comparison between the following two code implementations—sequential and parallel—demonstrates the impact of parallelization on a simple task of computing and printing squares of numbers from 1 to 5. Intel® VTune™ Profiler was used to accurately measure the performance metrics and execution times of both implementations.

The **sequential version** of the code is as follows:

```
#include <iostream>

using namespace std;

int main() {

   cout << "Numbers and their squares:" << endl;

   // Simple for loop that runs from 1 to 5

   for (int i = 1; i <= 5; i++) {

      int square = i * i;

      cout << "Number: " << i << ", Square: " << square << endl;

   }

   cout << "Loop completed!" << endl;

   return 0;

}
```

This version executes the loop in a straightforward, linear fashion. Each iteration calculates the square and prints it before proceeding to the next. It recorded an elapsed time of **0.087 seconds**.

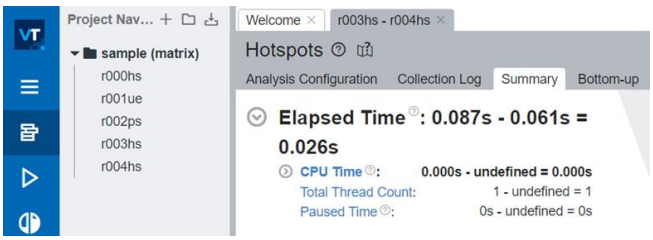
The **parallel version** of the code, using OpenMP for loop-level parallelism, is shown below:

```
#include <omp.h>

#include <iostream>

using namespace std;

int main() {

   cout << "Numbers and their squares:" << endl;

   // Simple for loop that runs from 1 to 5

#pragma omp parallel for

   for (int i = 1; i <= 5; i++) {

      int square = i * i;

#pragma omp atomic

      cout << "Number: " << i << ", Square: " << square << endl; }

   cout << "Loop completed!" << endl;

   return 0;}
```

This version distributes the loop iterations across multiple threads. To prevent race conditions during concurrent output to the console, an *#pragma omp atomic* directive is used around the cout statement. This synchronization, however, partially negates the performance gains from parallelization.

Despite the simplicity of the task, the parallel version showed a slight performance improvement with an elapsed time of **0.061 seconds**, compared to **0.087 seconds** for the sequential version. The difference, although measurable, is small due to the low computational load and the overhead introduced by thread management and output synchronization.

Below is a screenshot from Intel® VTune™ Profiler showing the elapsed time comparison between the two versions:



In conclusion, while the parallel implementation is marginally faster, the benefits are limited for such trivial operations. This result reinforces that parallelism is most effective when applied to larger, computation-heavy workloads where the cost of thread management is outweighed by the gains from concurrent execution.

VII. Conclusion

This project explored the effectiveness of automatic parallelization techniques, particularly compiler-based methods leveraging the polyhedral model, in enhancing the performance of sequential programs on multi-core processors. Through empirical evaluation using a simple

loop-based task, we observed that while parallelization can offer performance improvements, these benefits are most pronounced in computationally intensive workloads. For trivial operations, the overhead associated with thread management and synchronization can partially negate the gains from parallel execution. Our experiment, using OpenMP directives for loop-level parallelism, demonstrated a measurable but small speedup for a simple loop, reinforcing the principle that the efficacy of parallelism is directly proportional to the computational load and the ability to minimize synchronization overhead.

VIII. Future Work

Future work in this area can focus on several key aspects to further advance automatic parallelization:

A. **Advanced Dependency Analysis and Transformation:** Implement more sophisticated dependency analysis techniques to identify complex data dependencies and enable more aggressive loop transformations (e.g., tiling, fusion, interchange) that are not immediately apparent. This could involve integrating machine learning models to predict optimal transformation sequences for diverse code patterns.

B. **Dynamic Parallelization and Runtime Adaptation:** Explore dynamic parallelization techniques that can adapt to runtime conditions, such as varying workloads or heterogeneous architectures. This would involve runtime profiling and adaptive scheduling to optimize resource utilization and performance.

C. **Support for Irregular and Pointer-Intensive Codes:** Address the challenges of parallelizing irregular and pointer-intensive codes, which are often difficult for static analysis tools. Techniques like speculative parallelization or runtime disambiguation could be investigated.

References

[1] K. Lee, J. Zhang, and M. Chen, “OMPAR: AI-Driven Automatic Parallelization with Source-to-Source Compilation Using OpenMP,” Proc. of the International Conference on High Performance Computing, 2023.

[2] M. Pinto, J. Martins, and J. Cardoso, “AutoPar-Clava: OpenMP-based Automatic Parallelization of C Applications Using Clava Compiler,” International Journal of Parallel Programming, vol. 51, no. 4, pp. 1234–1256, 2022.

[3] G. Litjens et al., “A survey on deep learning in medical image analysis,” Medical Image Analysis, vol. 42, pp. 60–88, Dec. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0167819113000471

[4] H. I. Alshaikhli, A. A. Yassin, and A. A. Al-Douri, “Improved lung cancer detection using data mining techniques,” The Journal of Supercomputing, vol. 76, pp. 6944–6963, Sep. 2020. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s11227-019-03109-9.pdf

[5] R. Bellman, “Dynamic Programming and Partial Differential Equations,” arXiv preprint arXiv:2409.14771, Sep. 2024. [Online]. Available: https://arxiv.org/abs/2409.14771

[6] R. Rodrigues, J. Bispo, and J. M. P. Cardoso, “AutoPar-Clava: Automatic Parallelization Using Source-to-Source Compilation,” Technical Report, University of Porto, Portugal.

[7] M. Gupta and P. Banerjee, “Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers,” IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 2, pp. 179–193, 1992.

[8] M. Mehrara, “Compiler and runtime techniques for automatic parallelization of sequential applications,” 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:16205872

[9] “Automatic parallelization of canonical loops,” Science of Computer Programming, vol. 78, no. 8, pp. 1193–1206, 2013.

[10] A. Bhattacharyya and J. N. Amaral, “Automatic speculative parallelization of loops using polyhedral dependence analysis,” 2013.

[11] H. Arabnejad, J. Bispo, J. G. Barbosa, and J. M. Cardoso, “An OpenMP-based parallelization compiler for C applications,” in Proc. of the 2018 International Conference on High Performance Computing & Simulation (HPCS), pp. 915–923, 2018.

[12] L. Rauchwerger and D. Padua, “Parallelizing while loops for multiprocessor systems,” Parallel Computing, Feb. 2003.

[13] H. Bae et al., “The Cetus source-to-source compiler infrastructure: overview and evaluation,” International Journal of Parallel Programming, vol. 41, pp. 753–767, 2013.

[14] A. Bhosale, P. Barakhshan, M. R. Rosas, and R. Eigenmann, “Automatic and interactive program parallelization using the Cetus source-to-source compiler infrastructure v2.0,” Electronics, vol. 11, no. 5, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/5/809

[15] T. A. Johnson et al., “Experiences in using Cetus for source-to-source transformations,” in Languages and Compilers for High Performance Computing, Springer, 2005, pp. 1–14.

[16] G. R. A. Choudhary and L. Raina, “Serial to parallel code converter tools,” International Journal of Engineering Research & Technology (IJERT), vol. 12, no. 6, 2016. [Online]. Available: https://www.ijert.org/research/autoparallelization-IJERTV12IS060153