Technical Update Report on

Dynamic Dependency Checker and Pragma Injector for Source Code Parallelization

December 2, 2024

Prepared By:

Tanvir Ahmed Khan, 2131491642
khan.tanvir01@northsouth.edu

Mahir Shahriar Tamim, 2131377042
mahir.tamim@northsouth.edu

Mahiyat Nawar Mantaqa, 2122455042
mahiyat.mantaqa@northsouth.edu

Course Instructor:

Dr. Md Shahriar Karim
Associate Professor, Dept. of Computer Science

***Abstract*—This report describes a system designed to optimize and parallelize source code using dynamic dependency checking and pragma injection. The system analyzes source code for parallelization opportunities, injects parallelism directives, and supports multiple execution modes. The dispatcher controls the flow from source code to execution, including the generation of performance logs and profiling data through tools like gprof. The report highlights the roles of the dynamic dispatcher and pragma injector, with a focus on the workflow and code structure.**

I. Introduction

In high-performance computing (HPC), parallelism boosts efficiency by executing tasks simultaneously across multiple processors. Concurrency, on the other hand, manages multiple tasks in overlapping periods [1]. Understanding these concepts is essential for optimizing parallel execution [2].

Dependency analysis plays a key role in identifying efficient ways to parallelize tasks and avoid data races [3], while also profiling execution to further improve performance [4], [5].

This report describes a system that automates source code parallelization, consisting of a dependency checker, pragma injector, and dynamic dispatcher. The system as of now only accepts parallelizable loop and injects appropriate pragma directives, with performance profiling using tools like gprof [3].

II. Concurrency vs. Parallelism

In our work, concurrency and parallelism are critical in designing the parallelization system [6].

- **Concurrency**: Manages multiple tasks that may overlap in execution but not necessarily run simultaneously. It allows tasks to progress independently while sharing CPU time.
- **Parallelism**: Executes tasks simultaneously on multiple processors, improving performance in computationally intensive operations.

III. System Architecture

The system consists of three major components: the **Dependency Checker**, **Dynamic Dispatcher**, and **Pragma Injector**. Below is a detailed breakdown of each component and its function.

A. Main Diagram Overview

The main flow of the system involves the following sequence of steps:

- The source code is passed through the **Dependency Checker** which currently returns true by default, as our primary goal is not to check for dependency now, rather make the dispatcher.
- The **Dynamic Dispatcher** orchestrates the compilation, execution, and logging processes.
- The **Pragma Injector** analyzes loops in the source code and injects parallelization directives.
- Execution logs are generated to track performance.

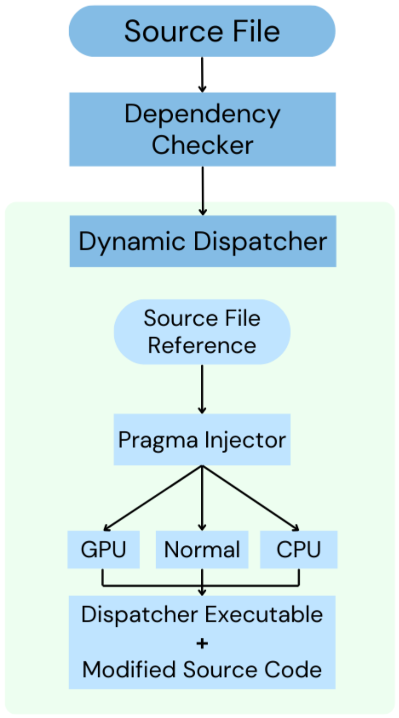

Fig. 1. Main Flow of the System

B. Dynamic Dispatcher Block

The **Dynamic Dispatcher** is responsible for handling the source file, invoking the pragma injector, managing different execution modes (Normal, CPU, GPU), and producing an executable binary. The key steps in this block include:

- **Source File Reference**: The dispatcher receives a reference to the source file that needs to be processed.
- **Pragma Injector**: The dispatcher invokes the pragma injector to modify the source code, adding parallelization directives where applicable.
- **Execution Mode**: The dispatcher allows the user to choose from Normal, CPU (OpenMP), or GPU (CUDA/OpenCL), or normal execution modes.

- **Compilation and Execution**: The dispatcher compiles the modified source code, generates the executable, and runs it.
- **gprof Integration**: The dispatcher integrates profiling through gprof to generate data dependency graphs, helping to analyze the parallelization's effectiveness.
- **Execution Log**: After execution, the dispatcher logs performance data (e.g., execution time) in an execution log.

C. Pragma Injector

The **Pragma Injector** is responsible for analyzing the source code, detecting loops, and injecting parallelism directives such as OpenMP pragmas. The main steps in the cycle are as follows:

- **Source File Reference**: The pragma injector receives a reference to the source file to analyze.
- **Execution Mode Selection**: The user selects the desired execution mode (Normal, CPU, or GPU).

Cycle Steps:

1) The **Lexical Analyzer** scans the source code to identify constructs such as loops and functions.
2) The **Loop Detector** identifies loops that can be parallelized.
3) If a parallelizable loop is found, the **Pragma Block Injector** adds the appropriate pragma directive (e.g., `#pragma omp parallel for`) to the loop.
4) The cycle repeats until all loops have been analyzed. If no more parallelizable loops are found, the injector exits.

D. Log Execution Function

The `log_execution` function logs the execution time and mode of the program. This function appends logs to a file `execution_log.txt` for later analysis.

```
Execution Log ========================
Run Mode       : Normal
Execution Time : 512.414 seconds
Execution Log ========================
Run Mode       : CPU
Execution Time : 249.261 seconds
Execution Log ========================
Run Mode       : GPU
Execution Time : 275.628 seconds
```

Listing 1. Execution Log

These logs demonstrate the execution time for each mode, showing the performance difference between running the program in Normal, CPU, and GPU modes.

Algorithm 1: Code used for benchmarking

```
Input: $N = 10000$, $M = 500$, $K = 200$: integers,
       array1: 3D array of size $N \times M \times K$
Output: Updated array1 with trigonometric values
for $i = 0$ to $N - 1$ do
    for $j = 0$ to $M - 1$ do
        for $k = 0$ to $K - 1$ do
            array1$[i][j][k] \leftarrow \sin(i) + \cos(j) + \tan(k)$;
return array1;
```

Algorithm 2: LexicalAnalysisAndLoopExtraction

```
Input: lines: array of strings, executionMode: char
Output: modifiedCode: string
Initialize: modifiedCode ← "", braceBalance ←
  stack(), inLoop ← false;
foreach line in lines do
    if outerLoop and RegexMatch(line,
     R"((for|while)*.*)") then
        if executionMode = 'c' then
            modifiedCode ← modifiedCode +
             #pragma omp parallel for;
        else if executionMode = 'g' then
            modifiedCode ← modifiedCode +
             #pragma omp target teams
             distribute parallel for;
        inLoop ← true, outerLoop ← false;
    if inLoop then
        foreach character c in line do
            if c = { then
                braceBalance.push(1);
            else if c = } and not braceBalance.empty()
             then
                braceBalance.pop();
                if braceBalance.empty() then
                    inLoop ← false, outerLoop ← true;
    modifiedCode ← modifiedCode + line + "\n";
return modifiedCode;
```

If the loop detection is true, the pragma block is injected, and the system returns to the lexical analyzer. Otherwise, it proceeds directly to the next section.

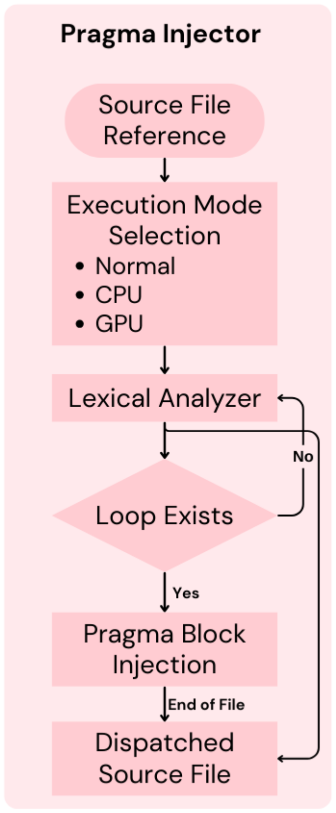

Fig. 2. Pragma Injector Workflow

IV. Code Implementation

A. Compile and Execute Function

The `compile_and_execute` function is responsible for compiling and executing the program in different modes based on user input. It times the execution and logs the result.

V. Profiling and Dependency Graph Generation

The profiling process is integrated into the dynamic dispatcher to generate performance data for different execution modes. After running the program in any of the available modes—Normal, CPU, or GPU—a profiling output is produced, which is used to generate a dependency graph. This graph visualizes the performance characteristics and data dependencies of the program, aiding in further optimization efforts. [3], [5].

The profiling and graph generation are done through the following steps:

- **Profiling the Program**: The profiling tool, `gprof`, is used to record runtime statistics of the application. It is invoked after compiling the program with profiling flags. The profiling output is stored in a file, specified by the `PROFILING_OUTPUT` variable.
- **Generating the Dependency Graph**: Once profiling is complete, the `gprof2dot` tool is used to convert the raw profiling data (`gmon.out`) into a format suitable for generating a graphical representation of the program's performance. The `dot` tool is then used to convert the data into a PNG image. The final output is saved in the `LOGS_DIR` directory as a `.png` file, with the name `dependency_graph_$(OUTPUT_NAME).png`.
- **Graphical Output**: The generated dependency graph provides a visual representation of the execution flow and data dependencies within the program. This graph helps identify performance bottlenecks and areas of the program that could benefit from parallelization.
- **Post-Execution Clean-up**: After generating the graph, the profiling output file `gmon.out` is removed to prevent clutter and ensure that the system remains clean for subsequent runs.

The following Makefile rule defines the process:

```
profile-dispatcher: $(PROFILING_OUTPUT)
        @echo "Running dispatcher with profiling in
     mode $(MODE)..."
        $(PROFILING_OUTPUT) $(SRC) $(MODE)
        gprof $(PROFILING_OUTPUT) gmon.out |
    gprof2dot -w | dot -Tpng -o $(LOGS_DIR)/
    dependency_graph_$(OUTPUT_NAME).png
        rm -f gmon.out
```

Listing 2. Makefile Rule for Profiling and Dependency Graph Generation

This rule runs the dispatcher with profiling in the specified mode, processes the output with `gprof`, and generates the corresponding dependency graph as a PNG file.

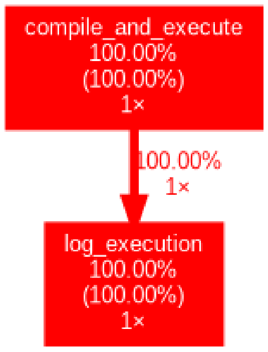

Fig. 3. Dependency Graph Normal Mode

VI. Future Work

Final report will focus on the following areas:

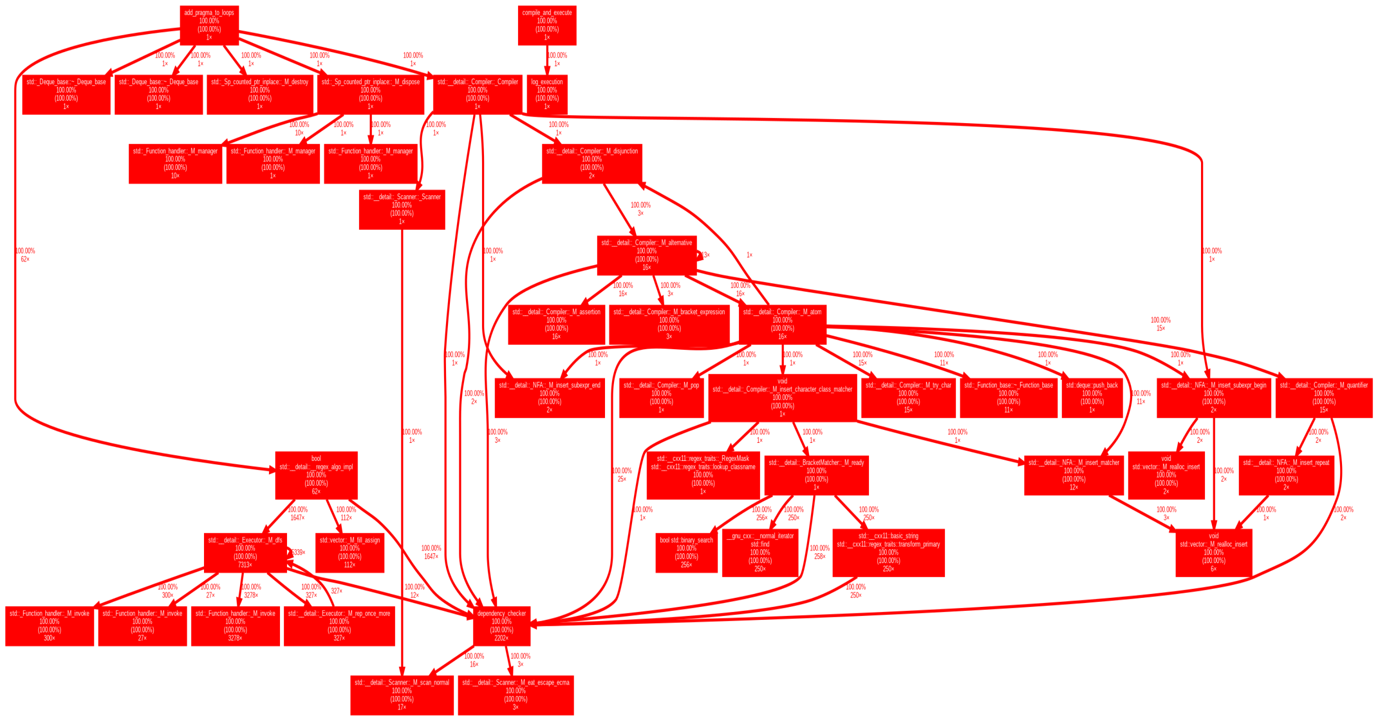
Fig. 4. Dependency Graph CPU GPU Mode

- **Energy Profiling**: Investigating energy consumption during execution, with a focus on optimizing power usage.
- **Heavy Benchmarks**: Adding more complex benchmarks to evaluate the system's performance under heavy loads.
- **Stack and Heap Effects**: Analyzing memory usage and optimizing stack and heap allocations.
- **Matrix Multiplication**: Enhancing parallel performance for matrix multiplication operations.
- **3D Matrix Rotation**: Optimizing 3D matrix rotation for parallel execution.
- **Floating-Point and String Comparison**: Comparing the performance of floating-point operations and string manipulations in different execution modes.

VII. Conclusion

The system provides an automated pipeline for source code parallelization using dependency checking and pragma injection. The dynamic dispatcher plays a crucial role in managing execution and logging, while the pragma injector ensures that parallelism is correctly applied to suitable loops in the source code.

References

[1] CMU, "Runtime implementation of parallelism," https://www.cs.cmu.edu/afs/cs/academic/class/15418-s19/www/lectures/20b$_r$*untime*$_i$*mpl.pdf*, 2019, *accessed* : 2024 – 11 – 29.

[2] X. Author and Y. Author, "Title of the ieee paper," *IEEE Conference Proceedings*, vol. 2023, no. 9741290, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9741290

[3] P. Contributors, "gprof2dot," https://pypi.org/project/gprof2dot/2019.11.30/, 2019, accessed: 2024-11-29.

[4] G. Project, "Dependency analysis in gcc," https://gcc.gnu.org/onlinedocs/gccint/Dependency-analysis.html, 2021, accessed: 2024-11-29.

[5] ——, "gprof: The gnu profiler," https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html$_m$*ono/gprof.html*, 2021, *accessed* : 2024 – 11 – 29.

[6] OpenMP, "Openmp offload programming introduction," https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf, 2021, accessed: 2024-11-29.