DynApex: A Dynamic Dependency Checker and Pragma Injector for Source Code Parallelization

Tanvir Ahmed Khan*, Mahir Shahriar Tamim†, Mahiyat Nawar Mantaqa‡, Md. Shahriar Karim§
Department of Electrical and Computer Engineering, North South University, Bangladesh
*Student Code: 2131491642, Email: khan.tanvir01@northsouth.edu
†Student Code: 2131377042, Email: mahir.tamim@northsouth.edu
‡Student Code: 2122455042, Email: mahiyat.mantaqa@northsouth.edu
§Associate Professor, Department of Electrical and Computer Engineering, North South University, Bangladesh
Email: shahriar.karim@northsouth.edu

***Abstract*—This paper introduces DynApex, a system for automated source code parallelization, utilizing dynamic dependency checking and pragma injection. DynApex analyzes source code for parallelization opportunities, inserts OpenMP directives, and supports multiple execution modes (Normal, CPU, and GPU). A dynamic dispatcher manages the execution flow, performance logging, and profiling using Valgrind. The dependency checker uses a graph-based approach to detect loop-carried dependencies. We also discuss Amdahl's Law in context of performance analysis. Due to time constraints, resolving of loop dependencies could not be implemented. The system's workflow, code structure, and performance results are discussed.**

I. Introduction and Literature Review

In the realm of high-performance computing (HPC), the pursuit of faster execution speeds is paramount. Parallel processing is a fundamental technique to achieve this by enabling simultaneous execution of computational tasks [1]. Understanding the nuances of concurrency and parallelism is crucial for effective optimization [2]. Concurrency manages multiple tasks, potentially overlapping in execution, while parallelism executes tasks simultaneously, leveraging multiple processors or cores [3].

Dependency analysis is pivotal for determining opportunities for parallelization, as data dependencies can introduce race conditions and incorrect results [4], [5]. Profiling the execution of code helps to further improve performance and detect bottlenecks [6]. Existing compilers and tools often have limitations in automatically detecting all parallelizable sections, especially in complex code structures. Our work aims to address this by providing a dynamic dependency checker and pragma injector that simplifies the process of source code parallelization.

This paper presents DynApex, a system that automates source code parallelization via a dynamic dispatcher, dependency checker, and pragma injector. DynApex injects OpenMP pragmas into parallelizable loops. The system is integrated with performance profiling via Valgrind and Amdahl's Law is analyzed in the context of the performance results.

II. Methods

A. System Architecture

DynApex is composed of three primary components: the Dependency Checker, the Pragma Injector, and the Dynamic Dispatcher. Each component plays a specific role in the overall process of source code parallelization as shown in Fig 1.

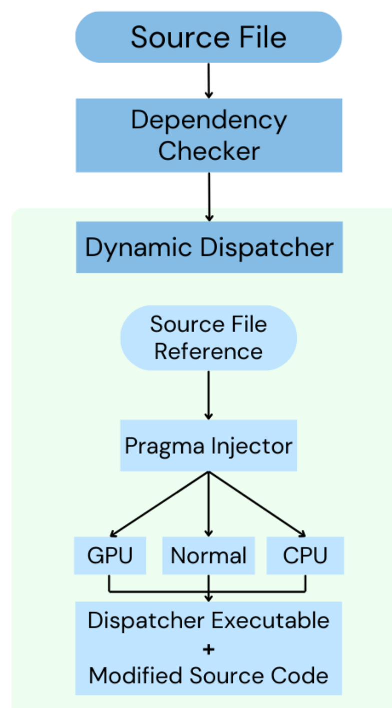

Fig. 1. Main Flow of DynApex

B. Dependency Checker

The Dependency Checker analyzes source code to identify potential loop-carried dependencies using graph theory [4]. A dependency graph is constructed to model relationships between variables within a loop [7]. The algorithm involves tracking variable scopes and detecting circular dependencies to identify cases where loops cannot be safely parallelized. This approach is rooted in compiler optimization techniques and data flow analysis. [1].

1) Dependency Graph Construction: The dependency graph *G* is represented as a map where each variable corresponds to a node. The edges between these nodes represent the flow of data between variables, which indicate dependency. The algorithm is described below in Algorithm 1.

Algorithm 1: Dependency Checking Algorithm

```
Input: Source code file F
Output: Boolean value, true if no loop-carried dependencies,
        false otherwise
Initialize: L ← ∅ (stack), G ← ∅ (dependency graph),
  inLoop ← false;
foreach line l in F do
    if l matches for loop pattern then
        L.push(∅);
        inLoop ← true;
    if l contains ‘ then
        ’
    and inLoop is true
L.pop();
inLoop ← false;
if l matches ‘var = expression;’ pattern then
    Extract lhs_var from var;
    Extract rhs_vars from expression;
    if lhs_var is an array access a[i+c] in loop then
        Extract array base name a;
        if c ≠ 0 then
            Add self-loop in G for a;
        Add a to current scope in L;
    else
        Add lhs_var to current scope in L;
    foreach rhs_var in rhs_vars do
        Add dependency of lhs_var to rhs_var in G;
V ← ∅ (visited nodes);
foreach node in G do
    if node ∉ V then
        Perform iterative DFS starting from node;
        if DFS detects cycle then
            Return false;
Deallocate memory for all node in G;
Return true;
```

2) Cyclic Dependency Detection: A crucial step in determining whether a loop can be safely parallelized is to check for loop-carried dependencies. These dependencies occur when the result of an iteration relies on the results of a previous iteration within the same loop. Such dependencies prevent parallel execution because iterations cannot be performed independently. We employ a Depth-First Search (DFS) algorithm on the dependency graph to detect if any cycles exist, which would indicate the presence of loop-carried dependencies [7].

Why DFS for Cycle Detection?

A dependency graph illustrates variable relationships within a loop. A cycle in this graph signifies a variable's value relying on itself, either directly or through a chain of dependencies.
Such cycles denote loop-carried dependencies, which prevent safe parallelization. DFS is ideal for cycle detection as it exhaustively explores paths, quickly revealing such circular dependencies, which sequential execution can only resolve.

Algorithm 2: Iterative Depth-First Search for Cycle Detection

```
Input: A node startNode, Set of visited nodes V
Output: Boolean value, true if cycle detected, false otherwise
Initialize: Stack stack, Set of visiting nodes visiting;
stack.push(startNode);
visiting.insert(startNode);
while stack is not empty do
    node ← stack.top();
    if node.visiting is false then
        node.visiting ← true;
        foreach dependency dep of node do
            if dep in visiting then
                Return true;
            if dep not in V then
                stack.push(dep);
                visiting.insert(dep);
    else
        stack.pop() visiting.erase(node);
        V.insert(node);
Return false;
  V ← ∅ (visited nodes);
  foreach node in G do
        if node ∉ V then
              Perform iterative DFS starting from node;
          if DFS detects cycle then
            Return false;
```

C. Pragma Injector

The Pragma Injector analyzes a source code file and adds OpenMP pragmas to parallelizable loops. This involves lexical analysis to recognize loops, and inserting appropriate directives, such as `#pragma omp parallel for`. The process ensures that only loops detected by the dependency checker as parallelizable are augmented with pragmas. DynApex primarily support parallelization of for and while loops [3].

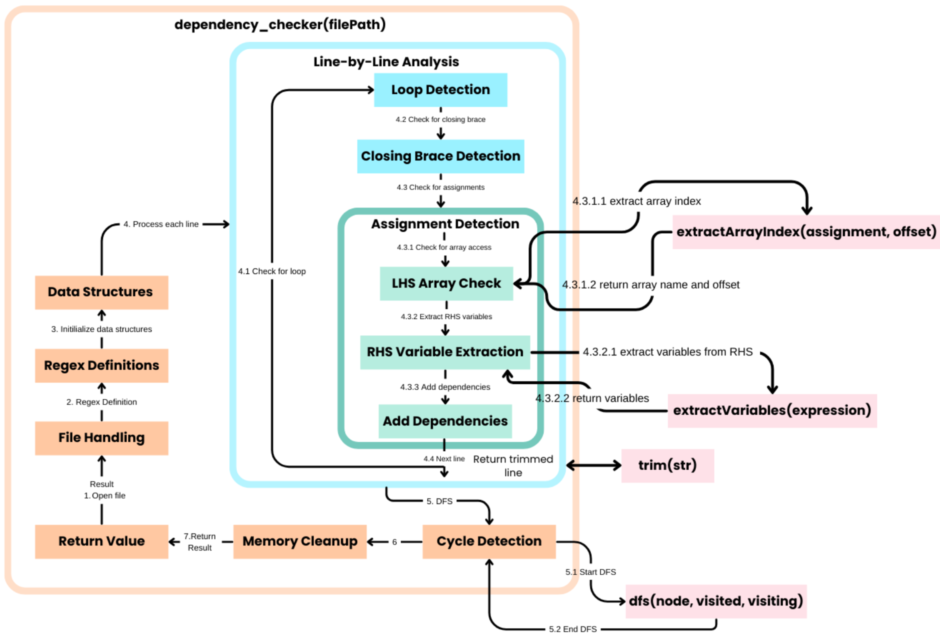

Fig. 2. Dependency Checker Process

Algorithm 3: Pragma Injection Algorithm

Input: Source code file F, Execution Mode $mode \in \{c, g\}$
Output: Modified source code file with pragmas
Initialize: $modifiedCode \leftarrow$ "", $braceBalance \leftarrow \emptyset$ (stack), $inLoop \leftarrow false$
foreach *line l in F* **do**
  if *outerLoop is true and l matches for loop or while loop pattern* **then**
    if *mode is c* **then**
      $modifiedCode \leftarrow modifiedCode +$ "#pragma omp parallel for";
    else if *mode is g* **then**
      $modifiedCode \leftarrow modifiedCode +$ "#pragma omp target teams distribute parallel for";
    $inLoop \leftarrow true$, $outerLoop \leftarrow false$;
  if *inLoop is true* **then**
    foreach *character c in line* **do**
      if *c is* { **then**
        $braceBalance$.push(1);
      else if *c is* } *and braceBalance is not empty* **then**
        $braceBalance$.pop();
        if *braceBalance is empty* **then**
          $inLoop \leftarrow false$,
          $outerLoop \leftarrow true$;
  $modifiedCode \leftarrow modifiedCode + l +$ "\n";
Overwrite F with $modifiedCode$;

D. Dynamic Dispatcher

The Dynamic Dispatcher is responsible for orchestrating the entire process of parallelization, including dependency checking, pragma injection, compilation, and execution. It supports various execution modes: Normal, CPU (OpenMP), and GPU (using OpenMP target offloading). The dispatcher integrates with Valgrind for performance profiling and produces performance logs.

E. Amdahl's Law

Amdahl's Law provides a theoretical framework for understanding the limits of parallel speedup [1]. It posits that the potential speedup of a program is limited by the fraction of the program that cannot be parallelized. If *p* is the fraction of a program that can be parallelized and *n* is the number of processors, the theoretical speedup *S* is:

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

This law highlights the diminishing returns of parallelization as the sequential portion of a program becomes more significant [2]. DynApex uses Amdahl's law to estimate speedup in CPU and GPU modes assuming that 70% of the code is parallelizable and uses 4 cores on CPU mode and 100 cores on GPU mode.

F. Performance Analysis

The execution logs demonstrate the performance of the program under different modes: Normal (sequential), CPU

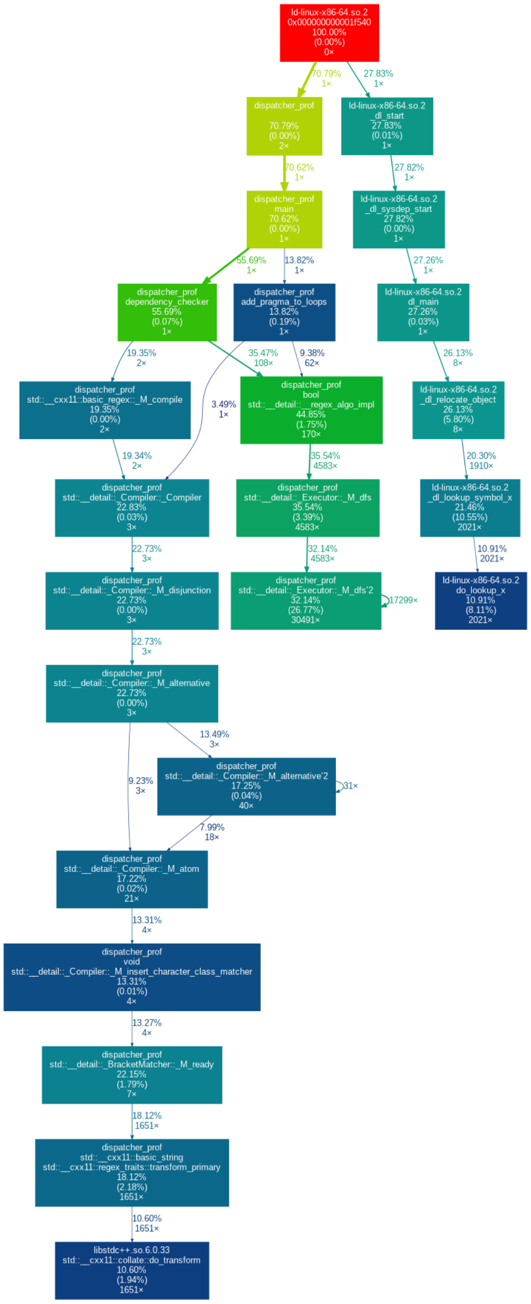

Fig. 3. Data Dependency Graph visualization

(parallel using OpenMP), and GPU (parallel using OpenMP target offloading). The execution times and speedup calculations are summarized in Table I.

Run Mode	**Exec Time (s)**	**Calculated Speedup**	**Actual Speedup**
Normal	6.40378	-	-
CPU	4.68323	2.10526	1.36739
GPU	2.75337	3.25733	2.3258

TABLE I
PERFORMANCE ANALYSIS

1) Analysis:

1) **Normal Mode:** This serves as the baseline. The sequential execution takes 6.40378 seconds.
2) **CPU Mode:**
   - The program executes in 4.68323 seconds.
   - **Calculated Speedup:** Using Amdahl's Law with $p = 0.7$ and $n = 4$:

$$S = \frac{1}{(1-0.7) + \frac{0.7}{4}} \approx 2.10526$$

   - **Actual Speedup:** The measured speedup relative to normal mode is 6.40378 / 4.68323 = 1.36739. The gap from the theoretical result is due to parallelization overheads and other factors.
3) **GPU Mode:**
   - Execution time is 2.75337 seconds.
   - **Calculated Speedup:** With $p = 0.7$ and $n = 100$:

$$S = \frac{1}{(1-0.7) + \frac{0.7}{100}} \approx 3.25733$$

   - **Actual Speedup:** Actual speedup relative to normal is 6.40378/2.75337 = 2.3258. The better speedup is because of better parallel resource in GPU, but still below theoretical limit.

Amdahl's Law provides a theoretical speedup limit, while the actual speedup observed is dependent on practical implementation factors. The execution results demonstrate that GPU offloading gives better acceleration.

III. Challenges and Remedies Used

A. Challenge 1: Loop Dependency Detection

Description: Initially, our dependency checker did not effectively detect complex loop-carried dependencies, especially those involving array indices. **Remedy:** We implemented a graph-based dependency analysis (as described in section 3.2). This graph is based on nodes and edges representing variables and their data dependencies. Additionally, our system can handle array index dependencies with a constant offset. The DFS algorithm is implemented using iterative approach instead of recursive for memory optimization [7].

B. Challenge 2: Accurate Pragma Injection

Description: The initial pragma injector would sometimes add directives incorrectly, causing compilation errors or incorrect program behavior, especially with nested loops. **Remedy:** The injector was refined to use stack-based tracking of braces to correctly identify loop scopes and inject directives at the correct level as described in Algorithm 3.

C. Challenge 3: Performance Profiling Integration

Description: Getting meaningful profiling data was a challenge, especially when integrating with Valgrind. Initially we used `gprof` which had some limitations in generating call graphs effectively. **Remedy:** We switched to Valgrind's Callgrind tool for more detailed profiling information and call graph generation [8]. The profile data is parsed using `gprof2dot` and `dot` to visualize the results.

D. Challenge 4: Dynamic Mode Switching

Description: Dispatching execution to different modes (Normal, CPU, GPU) dynamically required proper command handling and logging to distinguish the performance results. **Remedy:** We implemented dynamic checks based on command line arguments to correctly direct execution to normal, CPU, or GPU modes as described in section 3.3. Additionally, DynApex can choose between CPU or GPU mode randomly. The logs are formatted to give clear output.

IV. Unsolved Challenges and Potential Solutions

A. Challenge 1: Complex Dependency Patterns

Description: Our current dependency analysis does not handle all complex patterns, like interprocedural dependencies and complex array access patterns. **Potential Solution:** Exploring more advanced static analysis techniques, such as symbolic execution and interprocedural analysis to handle more complex dependencies [4].

B. Challenge 2: GPU Offloading Limitations

Description: While OpenMP target offloading is used, DynApex does not fully optimize data transfer to the GPU and does not support CUDA/OpenCL code generation. **Potential Solution:** Integrating CUDA/OpenCL code generation and optimizing GPU memory management for efficient data transfer and computation.

C. Challenge 3: Auto-Tuning of Parallel Parameters

Description: Currently, DynApex uses a fixed value for the estimated parallelizable portion of a code for calculating speedup. **Potential Solution:** Implementing auto-tuning capabilities that adjust thread counts and other parameters based on system characteristics and application profiles.

D. Challenge 4: Loop Dependency Resolution

Description: Due to time constraints, the system is not able to resolve loop dependencies. **Potential Solution:** Implementing techniques like loop transformations and data restructuring.

V. Conclusion

DynApex provides an automated solution for source code parallelization, using a dynamic dependency checker and pragma injector. The dispatcher manages the program execution by providing a flexible and configurable system that supports different execution modes and performance profiling using Valgrind. The dependency checker accurately detects loop-carried dependencies via graph-based analysis, and the pragma injector adds OpenMP pragmas to loops for parallel execution. Our implementation also takes Amdahl's Law into account for speedup estimation. However, it is important to mention that resolving of loop dependency is still an open challenge.

Future work includes enhancing the dependency analysis for more complex patterns, developing specific GPU code generation, incorporating auto-tuning capabilities for optimal parallel performance, and resolving the loop dependencies by applying loop transformations.

Contribution

Name	Contribution
Tanvir Ahmed Khan	Implemented the core logic of the dynamic dispatcher, integrated Amdahl's Law for speedup calculation, designed the logging function, and contributed to the documentation of the project.
Mahir Shahriar Tamim	Developed the dependency checking algorithm using graph theory and DFS, focusing on detecting loop-carried dependencies, and worked on refining the dependency checker and helped in project documentation.
Mahiyat Nawar Mantaqa	Designed and implemented the pragma injector, including lexical analysis and OpenMP pragma injection, and tested the system for various execution modes.

TABLE II
Member Contributions

Signatures

- ______________________________
  Tanvir Ahmed Khan
- ______________________________
  Mahir Shahriar Tamim
- ______________________________
  Mahiyat Nawar Mantaqa

Appendix

A. Example 1: Loop with No Data Dependency (Multiple Nested)

```
#include <iostream>
#include <vector>

int main() {
    const int N = 50;
    std::vector<std::vector<std::vector<int>>> cube
     (N, std::vector<std::vector<int>>(N, std::
     vector<int>(N)));

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            for(int k = 0; k < N; ++k){
                 cube[i][j][k] = i * j + k;
            }
        }
    }
    return 0;
}
```

Listing 1. Example 1: Loop with No Data Dependency (Multiple Nested)

B. Example 2: Loop with Cyclic Dependency (Multiple Nested)

```
#include <iostream>
#include <vector>

int main() {
  const int N = 50;
  std::vector<std::vector<std::vector<int>>> cube(N
    , std::vector<std::vector<int>>(N, std::vector<
    int>(N)));


    for (int i = 0; i < N; ++i) {
       for(int j = 0; j<N; ++j){
            cube[i][j][0] = i+j; // Initialize the
    first plane
          for (int k = 1; k < N; ++k) {
                cube[i][j][k] = cube[i][j][k-1] +
    1; // Dependency within k loop
            }
       }

    }
    return 0;
}
```

Listing 2. Example 2: Loop with Cyclic Dependency (Multiple Nested)

C. Example 3: Loop with More Complex Dependency

```
#include <iostream>
#include <vector>

int main() {
    const int N = 100;
    std::vector<int> a(N);
    std::vector<int> b(N);

    a[0] = 10;
    b[0] = 5;

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

        a[i] = a[i - 1] + b[i-1];
         if(i%2 == 0){
             b[i] = a[i-2] + 2;
        }
         else{
           b[i] = b[i-1] - 1;
         }
    }
    return 0;
}
```

Listing 3. Example 3: Loop with More Complex Dependency

References

[1] CMU, "Runtime implementation of parallelism," https://www.cs.cmu.edu/afs/cs/academic/class/15418-s19/www/lectures/20b_runtime_impl.pdf, 2019, accessed: 2024-11-29.

[2] I. C. Society, "Concurrency vs parallelism," https://www.computer.org/publications/tech-news/research/concurrency-vs-parallelism, 2023, accessed: 2024-11-29.

[3] 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.

[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_mono/gprof.html, 2021, accessed: 2024-11-29.

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

[7] GeeksforGeeks, "Find dependencies of each vertex in a directed graph," https://www.geeksforgeeks.org/find-dependencies-of-each-vertex-in-a-directed-graph/, 2024, accessed: 2024-11-29.

[8] V. Developers, "Callgrind user manual," https://valgrind.org/docs/manual/cl-manual.html, 2024, accessed: 2024-11-29.