Doctoral Dissertation

A Study on Automatic Parallelization with OpenMP using Large Language Model

Soratouch Pornmaneerattanatri
Program of Information Science and Engineering
Graduate School of Science and Technology
Nara Institute of Science and Technology

Supervisor: Professor Hajimu Iida
Software Design and Analysis Lab. (Division of Information Science)

Submitted on September 12, 2024

A Doctoral Dissertation
submitted to Graduate School of Science and Technology,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Engineering

Soratouch Pornmaneerattanatri

A Study on Automatic Parallelization with OpenMP using Large Language Model[1]

Soratouch Pornmaneerattanatri

Abstract

To fully utilize multi-core processors, the development of parallel programs is needed. However, developing parallel programs is a demanding task. Automatic parallelization techniques have been studied to simplify this process by automatically transforming sequential code into parallel code. Most existing automatic parallelization tools employ static analysis, which can identify certain types of parallel structures but fail to detect all, leading to suboptimal performance gains.

In contrast, the recent emergence of the Large Language Models (LLMs) in the Natural Language Processing (NLP) field has led software engineering researchers to adopt them, as LLMs have demonstrated state-of-the-art performance in various tasks. Motivated by these advancements, this study proposes an automatic parallelization tool based on LLMs. To replicate the functionality of existing automatic parallelization tools, two models are developed. The first model classifies parallelizable for-loops, while the second model generates OpenMP directives. Datasets are gathered from two sources, Google BigQuery public datasets and GitHub public repositories, and pre-processed to improve the quality of the OpenMP source code and to facilitate downstream tasks by fine-tuning CodeT5/CodeT5+, one of the Code LLM models.

The proposed models are evaluated using the NAS Parallel Benchmarks. The classification model achieves an F1 score of 0.713. The generation model successfully parallelizes the for-loops in 73% of cases and achieves speedup in five out of the eight NAS Parallel Benchmarks.

Keywords:

High-performance computing, Parallel software, Automatic Parallelization tools,

[1]Doctoral Dissertation, Graduate School of Science and Technology, Nara Institute of Science and Technology, September 12, 2024.

Large Language Model, Deep learning model

Contents

1 Introduction **1**

1.1 Problems in Parallel Programming 1

1.2 Motivation and Goal 5

1.3 Outline of this Dissertation 7

2 Background **8**

2.1 Automatic Parallelization Tools 8

2.1.1 Automatic Parallelization Support by Compiler 9

2.1.2 Source-to-Source Compiler 9

2.1.3 Static Analysis Techniques 10

2.2 Large Language Models 12

2.2.1 Self-attention Mechanism 12

2.2.2 Transformer Model 13

2.2.3 Code LLM 15

2.3 Related work 16

2.3.1 Automatic Parallelization Using Static Analysis 16

2.3.2 LLM and Code LLM 18

2.3.3 Automatic Parallelization Using LLM 19

3 Parallelizable For-loop Classification Model **21**

3.1 Introduction 21

3.2 Methodology 22

3.2.1 Overview 22

3.2.2 Data Collection 22

3.2.3 Data Labeling 24

3.2.4 Fine-tuning the CodeT5 Model 26

3.3 Evaluation Setup 27

3.3.1 Hardware and Software 27

3.3.2 Fine-Tuning 27

3.3.3 Evaluation Targets 28

3.3.4 Baseline 29

3.3.5 Performance Metrics 30

3.4 Evaluation Results 30
3.4.1 GitHub Projects 31
3.4.2 NAS Parallel Benchmarks 31
3.5 Discussion 32
3.5.1 Analysis of the Correctly Predicted Results 33
3.5.2 Analysis of the Incorrectly Predicted Results 34
3.5.3 Benefits of the Proposed Model as a Coding Companion . 36
3.6 Conclusion 37

4 OMP-CodeT5+: An OpenMP Directive Generation Model 38
4.1 Introduction 38
4.2 Methodology 39
4.2.1 Overview 39
4.2.2 Data Collection and Preprocessing Process 41
4.2.3 Model Training 42
4.3 Hardware and Software Setup 43
4.3.1 Model Training 43
4.3.2 Parallelized Code Evaluation 43
4.3.3 Fine-tuning 43
4.3.4 Evaluation Method for Automatic Parallelization Tools . . 44
4.3.5 Baseline Methods Compared with the Proposed Model . . 44
4.4 Evaluation Results 45
4.4.1 Performance Evaluation Using Benchmarks 45
4.4.2 Evaluation of Syntax Correctness and Benchmark Verification for Generated OpenMP Directives 47
4.4.3 The Evaluation Metrics Scores At Each Epoch 48
4.5 Discussion 49
4.5.1 Analysis Through Speedup Categories 49
4.5.2 Analysis of the Impact of Training Dataset Characteristics 55
4.5.3 Analysis of the Generated Clauses 63
4.5.4 Failures of OMP-CodeT5+ in OpenMP Directive Generation 68
4.5.5 Nested For-loops and incorporating Contextual Information in Training Data 72

4.5.6 Evaluation of the Practicality of OMP-CodeT5+ in Parallel Programming 72
4.6 Conclusion 74

5 Automatic Parallelization Tool based on LLM **75**
5.1 The Expected Results of Automatic Parallelization Tool based on LLM 75
5.2 Advantages of using the Proposed LLM-based Automatic Parallelization Tool 78
5.3 Target Users 78

6 Conclusion and Future Work **79**

Acknowledgements **81**

Reference **83**

List of Publications **89**

List of Figures

1 Single-threaded Program Execution on a Multi-core Processor . . 2
2 Parallel Program Execution using OpenMP on a Multi-core Processor 3
3 The Overview of the Proposed Automatic Parallelization using LLM 5
4 Functions of Automatic Parallelization Tools 8
5 Scaled Dot-product Attention and Multi-head Attention Mechanism 13
6 Transformer Architecture 14
7 German to English Language Translation Task Performed by T5 Model 15
8 Python to C Programming Translation Task performed by CodeT5 Model 16
9 The Overview of the Parallelizable For-loop Classification Model . 23
10 Locating For-loop and OpenMP Directive and Extracting Data . . 25
11 (a) Parallelizable code snippets are labeled as "1" (b) Non-parallelizable code snippets are labeled as "0" 28
12 Overview of OMP-CodeT5+ model Processes 40
13 Overview of OMP-CodeT5+ model Processes 41
14 Testing OpenMP For-loop Found In Microsoft/clang Repository, File: test/OpenMP/target_teams_distribute_parallel_for_simd-_ast_print.cpp Line: 113-115 41
15 Speedup Over the Original Serial NPB Runtimes Using Various Automatic Parallelization Tools and Manual Parallelization 46
16 The Exact Match and ROUGE Scores with Split Training Data At Each Epoch 48
17 Overall Pass Generated OpenMP Directives in Each Size of For-loop 57
18 Training Data Ratio divided by Tokens 60
19 Number of Clauses in Training Data 61
20 Number of Clauses in Generated Directives 62
21 Number of Passed and Failed Clauses in Generated Directives . . 63
22 Ratio of Schedule Kind in Training Data 64
23 Parallelization Success and Failure Rates in Parallel NPB For-loops by Parallelizable For-loops Classification Model and OMP-CodeT5+ Model 76

List of Tables

1 GitHub Project Evaluation Results 31
2 NPB Benchmark Evaluation Results 31
3 Evaluation of Syntax Correctness and Benchmark Verification for Generated OpenMP Directives 47
4 Number of Passed and Failed Generated OpenMP Directives in Each For-loop Size Category 56
5 Detailed Results of Parallelization in Parallel NPB For-loops: Identification and Generation Success Rates 76

1 Introduction

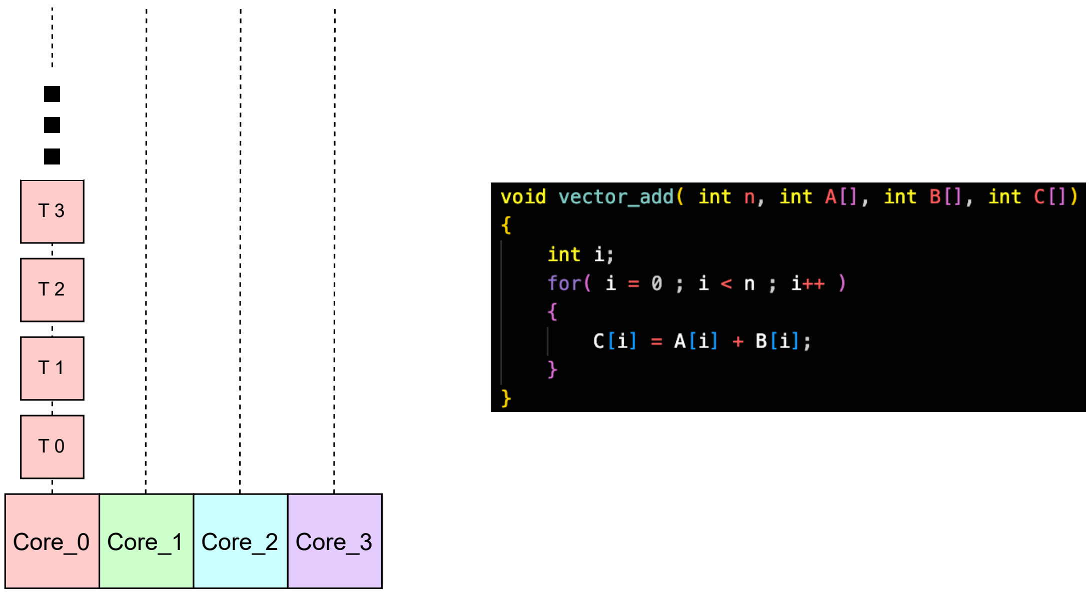
1.1 Problems in Parallel Programming

In the early stages of computer architecture, single-core processors were predominant. Single-threaded programs sufficed to leverage the computing performance of these processors. However, the performance enhancement of single-core processors is inherently constrained by physical limitations, primarily the heat dissipation associated with the increase in clock frequency. To mitigate this, processor manufacturers opted to increase the number of cores while employing lower clock frequencies to maintain acceptable thermal conditions. This innovation led to the emergence of multi-core architecture, which combines the computational power of several cores operating at reduced clock frequencies.

The advent of multi-core architecture, which enables users to run more than one single-threaded program simultaneously, has led to a decline in the computing and energy efficiency of individual single-threaded programs. As shown in Fig. 1, a single-threaded program executes on only one of the cores in a multi-core processor. Furthermore, rapid advancements in computer hardware, characterized by increasing core counts per processor, worsen the decline in single-threaded program efficiency at an accelerated pace.

While the multi-core architecture supports the concurrent execution of multiple single-threaded programs, this form of concurrency does not involve executing single instructions across cores, thereby limiting the computational efficiency of individual programs. In contrast, parallel programs that distribute a single instruction across multiple cores can fully exploit multi-core architecture to enhance program efficiency. As shown in Fig. 2, parallel programs operate by splitting a task into sub-tasks that can be executed simultaneously by multiple cores in a multi-core processor. Each core handles a portion of the task concurrently with others, thus reducing the overall processing time. These sub-tasks are then allocated to different cores for execution. Once all sub-tasks are completed, their results are aggregated to produce the final output. This method leverages the computational capabilities of multi-threaded programs to perform complex calculations more efficiently than single-threaded programs.

However, developing a parallel program is a challenging task [1–4]. There are



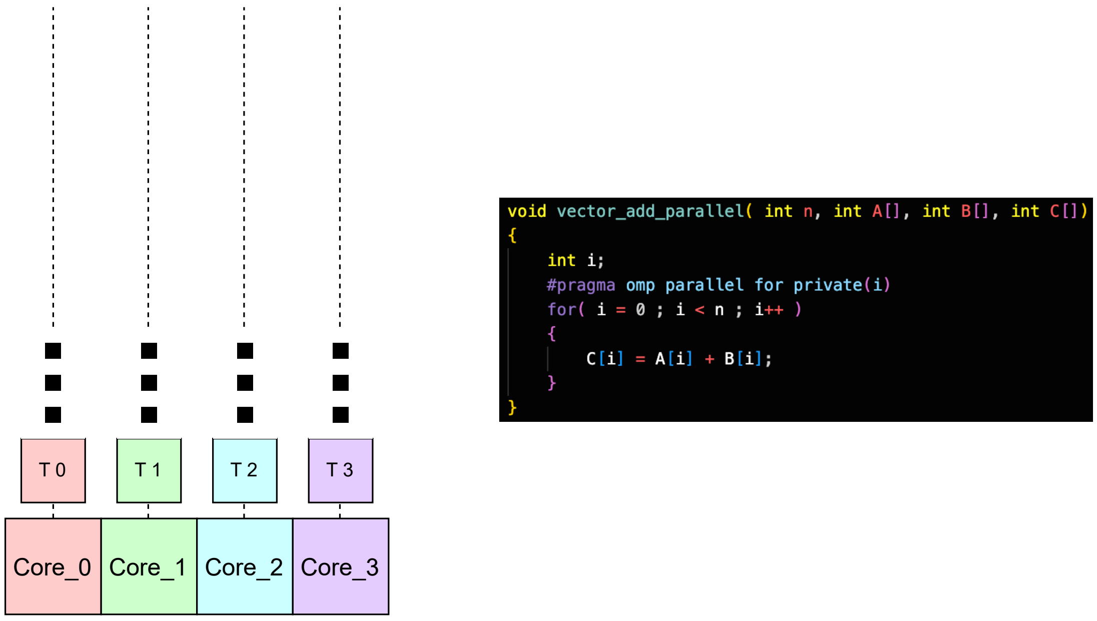
```c
void vector_add( int n, int A[], int B[], int C[])
{
    int i;
    for( i = 0 ; i < n ; i++ )
    {
        C[i] = A[i] + B[i];
    }
}
```

Figure 1: Single-threaded Program Execution on a Multi-core Processor

various implementations of parallel pattern, and each pattern is effective in specific situations. If the developer does not correctly understand the implementation of the parallel program and fails to incorporate parallel algorithms and hardware, the performance of the parallel program will suffer. To develop a performant parallel program, developers must understand both software and hardware. The process of mastering parallel programming presents a steep learning curve. Developers require significant time to become specialists in parallel programming. Even experienced parallel developers need time to design and develop the program according to the requirements and targets of the programs.

Multiple supports for parallel programming [5, 6] exist, either provided by the compiler itself or through additional libraries. Open Multi-Processing (OpenMP) [7] is offering by C/C++ and Fortran compilers. OpenMP is an Application Programming Interface (API) for C/C++ and Fortran languages that utilizes shared-memory programming patterns to transform single-threaded programs into parallel programs. Developers can enable OpenMP during the compiling process to parallelize the source code. The compiler then processes the source code and parallelizes sections containing OpenMP directives specified by the developers.

Another form of parallel programming support is the Message Passing Interface



```
void vector_add_parallel( int n, int A[], int B[], int C[])
{
    int i;
    #pragma omp parallel for private(i)
    for( i = 0 ; i < n ; i++ )
    {
        C[i] = A[i] + B[i];
    }
}
```

Figure 2: Parallel Program Execution using OpenMP on a Multi-core Processor

(MPI) [8], a message-passing API with a communication protocol and semantic specification for developing parallel programs that run across computer clusters or distributed memory systems. Developers must implement parallel sections of the source code using MPI semantics and compile them with the MPI compiler. As is common with open-source software, there are multiple MPI implementations. However, the MPI forum releases standards that all MPI implementations must follow. This allows developers to compile and run distributed memory parallel programs with any MPI implementation.

To tackle the complexity of parallel programming, many researchers and developers created automatic parallelization tools that automatically convert single-threaded programs into parallel programs. These tools shorten the development time and help inexperienced developers in parallel programming. Automatic parallelization tools function primarily by analyzing the input source code to identify parallelizable sections. Once these sections are identified, the analyzer generates and inserts the parallel code into the source code automatically. Both the analyzer and generator of automatic parallelization tools employ static analysis techniques.

Automatic parallelization tools based on static analysis techniques have shown limitations in improving performance for various for-loop patterns [9]. While some static analysis techniques can parallelize specific patterns effectively, others can handle different patterns but not all. Developing static analysis techniques that can parallelize every for-loop pattern requires updating the mechanisms to recognize other for-loop patterns that are not yet recognized. Nevertheless, each static analysis technique already contains complex logic for analyzing for-loop patterns. Updating this logic is a complicated task, leading to tools that are challenging to develop and maintain.

A recent breakthrough that has propelled the Natural Language Processing (NLP) field forward is the introduction of the transformer model, which is based on an attention mechanism. Transformer-based models have outperformed average humans in NLP tasks for the first time. The transformer model utilizes the two-stage training: pre-training for language understanding and fine-tuning for downstream tasks. In the pre-training phase, the model is trained with unlabeled data, utilizing the data itself as the label. This requires a large amount of data, thus it is named a Large Language Model (LLM). The fine-tuning phase then trains the model with traditional labeled data to address specific downstream tasks.

Following the success of transformer-based models, researchers in various fields have adopted the transformer model, utilizing domain-specific data to create downstream tasks. The software engineering field also employs the transformer model, training it with computer source code in multiple languages. The CodeT5/CodeT5+ models, based on the T5 model, have demonstrated superior performance in tasks such as code summarization, code generation, and code translation compared to previous studies.

The primary challenge in developing a deep learning model for new tasks is the availability of training data. Currently, there is no existing distributed dataset for developing a deep learning model that addresses parallel programming, parallelizable for-loop classification, and OpenMP directive generation. Therefore, in this study, the dataset was constructed from scratch, involving data gathering and preprocessing for model development. Concurrently with the development of the models, the dataset was continuously enhanced in both quantity and quality,

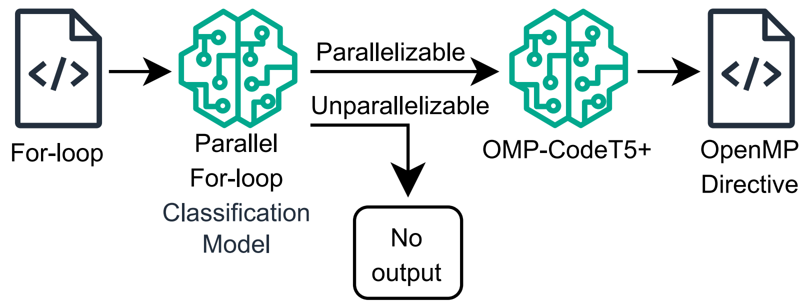


Figure 3: The Overview of the Proposed Automatic Parallelization using LLM

becoming the training data.

In the data preprocessing step, the training and output data needed to be designed for input into the model and to produce results. To create the training and output data for tasks related to parallel programming, the sequence of code fed into the model had to be defined to avoid training on unnecessary data. This newly created training data will impact the performance of the models that classify parallelizable for-loops and generate OpenMP directives.

Lastly, the evaluation of the Code LLM related to parallel programming tasks lacks standardized and widely accepted metrics. In this study, the model was assessed using the most relevant metrics corresponding to the generated tasks. Furthermore, the evaluation of the OpenMP directive generation model was performed by executing a traditional benchmark, thereby demonstrating the effectiveness of the generated OpenMP directives.

The significant contribution of this study is the development of an automatic parallelization tool utilizing LLM trained with the newly created OpenMP parallel source code dataset. This tool comprises two models: one that classifies parallelizable for-loops and another that generates OpenMP directives.

1.2 Motivation and Goal

This research aims to develop a novel method that leverages the language understanding and generation capabilities of LLMs to identify sections of source code that can be parallelized and to generate the corresponding parallelized code. Specifically, this research focuses on generating OpenMP directives for parallelizable sections. OpenMP provides a general and widely-accepted method for expressing how parallelizable sections should be parallelized. Many existing automatic parallelization tools, based on static analysis, also target the automatic generation of OpenMP directives, and compiler-based automatic parallelization often internally uses OpenMP. By focusing on OpenMP, this research aligns with the established standards and practices in the field, ensuring compatibility and leveraging the existing ecosystem of parallelization tools and methodologies.

Traditional automatic parallelization tools and compiler-based approaches, which rely on static analysis, often fall short in capturing the diverse patterns present in source code that can be parallelized. These tools have limitations in their ability to generalize across various coding styles and complex code structures. In contrast, the proposed approach using LLMs can automatically learn from a vast dataset of source code, enabling the model to capture a broader range of parallelization patterns that static analysis might miss. By leveraging the advanced language capabilities of LLMs, this method aims to improve the accuracy and effectiveness of automatic parallelization, addressing the limitations of traditional tools.

The overview of the methodology for this study is illustrated in Fig. 3. In this study, one of the encoder-decoder architecture-based Code LLMs, the CodeT5/CodeT5+ model, is fine-tuned to identify parallelizable loops and generate corresponding OpenMP directives. To replicate and enhance the capabilities of existing automatic parallelization tools, two models will be developed. First, a classification model will be designed to identify parallelizable for-loops using the encoder function of the CodeT5 model. This model will determine whether a given code snippet containing a for-loop is suitable for parallelization. Second, an OpenMP directive generation model, referred to as OMP-CodeT5+, will be developed. This model will utilize both the encoder and decoder functions of the CodeT5+ model. The encoder will analyze the structure of the given source code snippet, while the decoder will generate the appropriate OpenMP directives based on this analysis.

1.3 Outline of this Dissertation

The outline of this dissertation is as follows. Chapter 2 focuses on automatic parallelization tools, exploring their functionalities, including automatic parallelization by compilers and source-to-source compilers. This chapter also discusses static analysis techniques as crucial methods for automatic parallelization. Additionally, it covers Large Language Models, explaining the self-attention mechanism and transformer models, with a particular emphasis on Code LLMs. The chapter wraps up with an examination of related work in the field, automatic parallelization utilizing static analysis techniques and LLM, LLMs and Code LLMs.

Chapter 3 introduces the Parallelizable For-loop Classification Model, starting with an overview and detailed methodology. This includes data collection, labeling, and fine-tuning of the CodeT5 model. The chapter then outlines the evaluation setup, describing the hardware and software used, the fine-tuning process, evaluation benchmarks, baseline methods, and performance metrics. The evaluation results are presented next, analyzing the model's performance on public GitHub repositories and NAS parallel benchmarks. This is followed by a comprehensive discussion, which includes an analysis of correct and mispredicted classifications.

Chapter 4 begins with an introduction to the model and its methodology, covering data collection, preprocessing, and model training. The chapter details the hardware and software Setup used for model training and evaluation, and the benchmarks and baseline methods employed. Evaluation results are then presented, showing the source code performance and the success rates of OpenMP directives generated. The chapter's Discussion analyzes the speedup achieved in NAS parallel benchmarks.

Chapter 5 summarizes the research findings in this dissertation and proposes directions for future research to enhance the automatic parallelization tools.

2 Background

2.1 Automatic Parallelization Tools

Automatic parallelization tools serve as aids in parallel programming, converting single-threaded programs into parallel programs automatically. As shown in Fig. 4, these tools function primarily by analyzing the input source code to identify parallelizable sections. Once these sections are identified, the analyzer generates and inserts the parallel code into the source code automatically, thereby reducing the workload and complexity of parallel program development for developers.

Usually, automatic parallelization tools employ static analysis techniques that examine the source code. These techniques identify loops that can be parallelized. By performing this analysis, the tool determines how to transform single-threaded code into parallel code, optimizing it for multi-core processors. The static analysis process includes verifying the code's syntax, data flow, and control structures to ensure that the parallelization will not introduce errors or unexpected behaviors. Upon completing the analysis, the tool generates the necessary parallel code and integrates it into the input source code. This aids developers in optimizing their programs for parallel execution without manually inserting the parallel annotation.

There are two primary approaches to using automatic parallelization tools: automatic parallelization support by the compiler and source-to-source compilers. The automatic parallelization supported by the compiler parallelizes the source code during compile time, generating the parallel program execution file from the single-threaded source code. Source-to-source compilers offer various functions, such as translating source code from one programming language to another, code

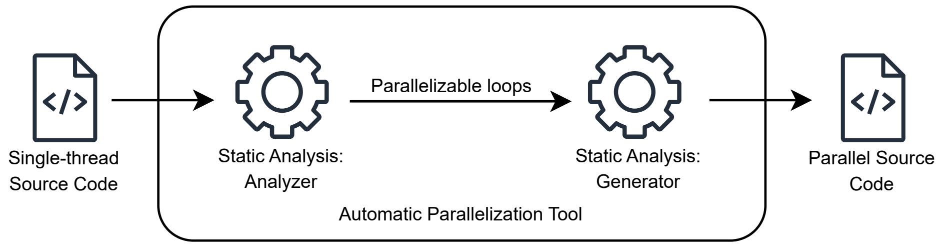

Single-thread Source Code
Static Analysis: Analyzer
Parallelizable loops
Static Analysis: Generator
Parallel Source Code
Automatic Parallelization Tool

Figure 4: Functions of Automatic Parallelization Tools

refactoring, and automatic parallelization. These tools read the input source code, identify parallelizable sections, generate parallelized source code, and annotate the parallelized code back into the input source code.

2.1.1 Automatic Parallelization Support by Compiler

Several compilers, including the Intel compiler or oneAPI, GNU C/C++ compiler, and PGI compiler, support automatic parallelization through OpenMP. During the compilation process, this feature meticulously analyzes the source code to identify segments suitable for parallel execution. The compiler subsequently generates parallel code, capable of executing on multiple processors or cores, thereby augmenting the program's performance.

A key technique in automatic parallelization is loop parallelization. The compiler examines loops within the code to determine the feasibility of concurrent iteration execution. For instance, the Intel compiler uses dataflow analysis [10, 11] to identify parallelizable code and employs a cost model to determine and parallelize the source code based on potential performance gains. Upon identifying a parallelizable loop, the compiler inserts suitable parallel constructs, such as OpenMP directives, into the code, which subsequently guide the parallel execution during runtime.

However, the efficacy of this support is dependent upon the proficiency of the compiler in accurately analyzing and transforming the code. Leading-edge compilers, such as the Intel compiler and GNU compiler, exhibit advanced capabilities in automatic parallelization, employing techniques like dataflow analysis which are static analysis techniques. Despite these advancements, inherent limitations persist, as some code sections may not be helpful to parallelization, potentially leading to negligible performance improvements or, in some instances, even performance degradation.

2.1.2 Source-to-Source Compiler

Unlike traditional compilers like the Intel compiler or the GNU compiler that convert source code directly into machine code, source-to-source compilers translate source code from one high-level language to another while still maintaining the high-level structure and semantics of the code. This characteristic makes them particularly useful for tasks such as code optimization, language migration, and adding new features to old codebases. For instance, a source-to-source compiler can translate code from an older language like Fortran into a more modern language like C++, making it easier to maintain and improve.

Automatic parallelization by a source-to-source compiler involves transforming existing source code into an optimized version that can execute in parallel on multiple processors or cores. During this process, the source-to-source compiler first parses the input source code to construct an intermediate representation, commonly in the form of an Abstract Syntax Tree (AST). The AST represents the syntactic structure of the source code in a tree format, where each node corresponds to a construct in the source code and provides information about the syntax, variable types, and code location. Utilizing this information, the compiler performs an in-depth analysis to identify potential parallelizable loops that can be executed in parallel. Following the analysis, the source-to-source compiler generates an optimized version of the source code, incorporating parallel constructs, such as OpenMP directives, to the potential parallelizable code loops indicated by the analyzer. The modified source code is then compiled using a traditional compiler to create an executable program that benefits from parallel processing.

Similar to automatic parallelization support provided by compilers, source-to-source compilers utilize static analysis techniques to analyze and generate parallel source code. Source-to-source compilers often employ more sophisticated analyses, resulting in higher performance outputs of the parallelized source code. Many of these compilers use multiple static analysis techniques, for instance, AutoPar-Clava utilizes dependence analysis [12] to analyze and generate the parallel source code, to effectively analyze and generate parallelized source code. Nonetheless, the performance output of parallelized source code from source-to-source compilers can sometimes experience performance degradation, though generally to a lesser extent compared to automatic parallelization support by compilers.

2.1.3 Static Analysis Techniques

Both types of automatic parallelization tools employ static analysis techniques to examine the source code without executing it to understand its behavior and identify sections that can be parallelized. These techniques inspect the code to identify potential parallelizable loops and ensure the correctness of parallelized code. There are multiple techniques have been developed, for example, dataflow analysis, dependence analysis, alias analysis, etc. Upon completing the analysis, the tool generates the necessary parallel code for the potential parallelizable loops.

Dataflow analysis [13] is a technique that examines the flow of data within a program, identifying how data is generated, modified, and utilized. Dataflow analysis helps identify dependencies between different parts of the code, ensuring that parallel tasks do not interfere with each other by accessing or modifying the same data simultaneously. By understanding data dependencies, the compiler can safely parallelize code segments that can execute in parallel.

The static analysis techniques that are commonly deployed with automatic parallelization by source-to-source compilers like dependence analysis, alias analysis, and symbolic analysis will be explained as follows.

Dependence Analysis [14] is used to determine the dependencies within loops, checking for data, control, and resource dependencies to identify whether a loop can be safely parallelized. This analysis can distinguish between true dependencies, one task depending on the result of another task, and false dependencies, it can be executed in parallel but needs to avoid the executing dependent part in parallel.

Alias Analysis [15] evaluates how different pointers or references in the code may point to the same memory location. In parallel programs, it is essential to know whether different parallel tasks are going to modify the same variable or memory location or not. Alias analysis helps identify such potential conflicts and ensures that parallel tasks work on different memory locations.

Symbolic Analysis [16] involves evaluating symbolic expressions within the code to understand the relationship between different variables. This analysis helps determine loop bounds, array accesses, and conditions for safe parallel execution. With this information, the source-to-source compiler can identify the safe potential parallelizable loops.

The automatic parallelization tools using static analysis techniques can effectively transform single-threaded programs into parallel programs. These tools ensure that the parallel source code is efficient and correct, free from errors and bugs resulting from the parallelization process. However, the assurance depends on the logic incorporated in the analysis. When the analysis encounters code that falls outside the scope of the implemented logic, the parallelizing output may produce errors or lead to performance degradation.

2.2 Large Language Models

Natural Language Processing (NLP) is a field dedicated to processing human languages using rule-based, statistical, and more recently, deep learning methods. These NLP algorithms are employed in the language model that is designed to comprehend, interpret, and generate human languages utilizing the datasets to analyze and predict linguistic patterns. The notable examples of the NLP tasks are language understanding, question and answering, and text classification.

The advancement of machine learning has enabled the development of deep learning techniques. NLP researchers have employed these techniques to address NLP tasks. The Recurrent Neural Network (RNN) [17] was once the preferred choice for solving NLP tasks, due to its short training time. However, its limitation lies in its bias towards data exposed to later during training. The introduction of the LLM has not only outperformed the RNN but also, for the first time, exceeded the average human score in various NLP tasks.

2.2.1 Self-attention Mechanism

Self-attention mechanism [18] allows for the assessment of the relative importance of words within a sentence. This enables the model to capture long-range dependencies and contextual information more effectively than traditional recurrent neural networks. As shown in Fig. 5, it operates by generating three vectors for each word in a sentence, query, key, and value. These three vectors are obtained through linear transformations of the word's embedding and used to calculate a weighted. The query vector of a given word is compared with the key vectors of all words in the sentence to calculate attention scores. These scores indicate the relation score of the query word with every word in the sentence. Subsequently, the attention scores are used to compute a weighted sum of the value vectors that represent the information of each word from the entire sentence.

The self-attention mechanism effectively understands context in long sentences and reduces the computational complexity of processing sequential data. The

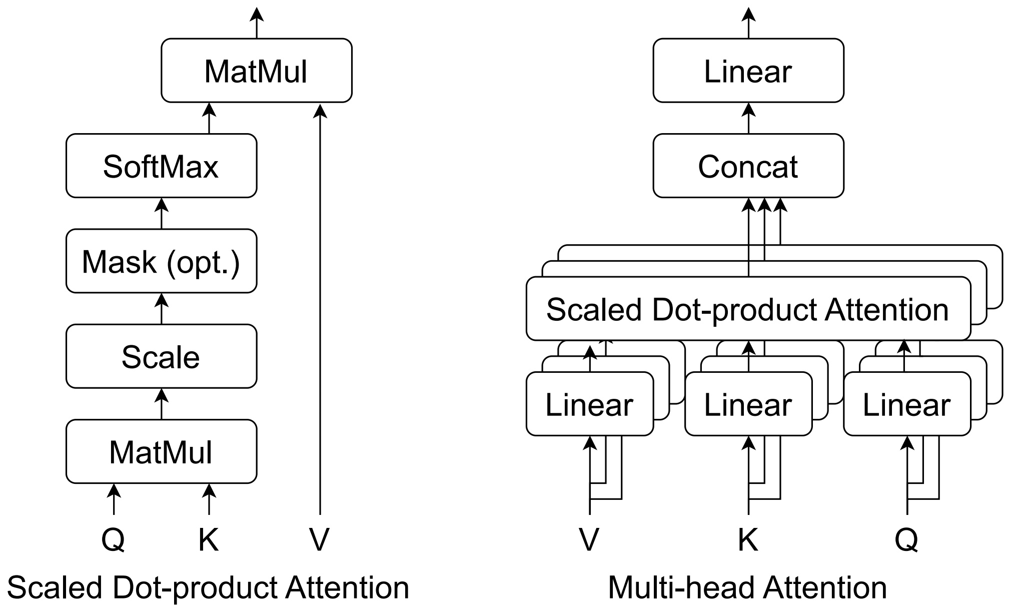


Figure 5: Scaled Dot-product Attention and Multi-head Attention Mechanism

ability to focus on significant parts of the input sequence while ignoring the less significant parts makes the self-attention mechanism powerful.

2.2.2 Transformer Model

A recent breakthrough that has propelled the NLP field forward is the introduction of the transformer model [18], which is based on a self-attention mechanism. The transformer model is an encoder-decoder architecture that utilizes a new mechanism called multi-head that relies entirely on the attention mechanism as shown in Fig. 6. The multi-head attention independently operates, focusing on different positions of the input sequences, allowing the multi-head attention mechanism to execute in parallel. The output from each calculation is concatenated to produce the final output. The multi-head mechanism effectively comprehends the complex patterns and relationships within the sentence.

The encoder and decoder architecture divide the responsibility between these two, the encoder calculates the series of weights or representations between words, and the decoder generates the output sequence from the encoder representations.

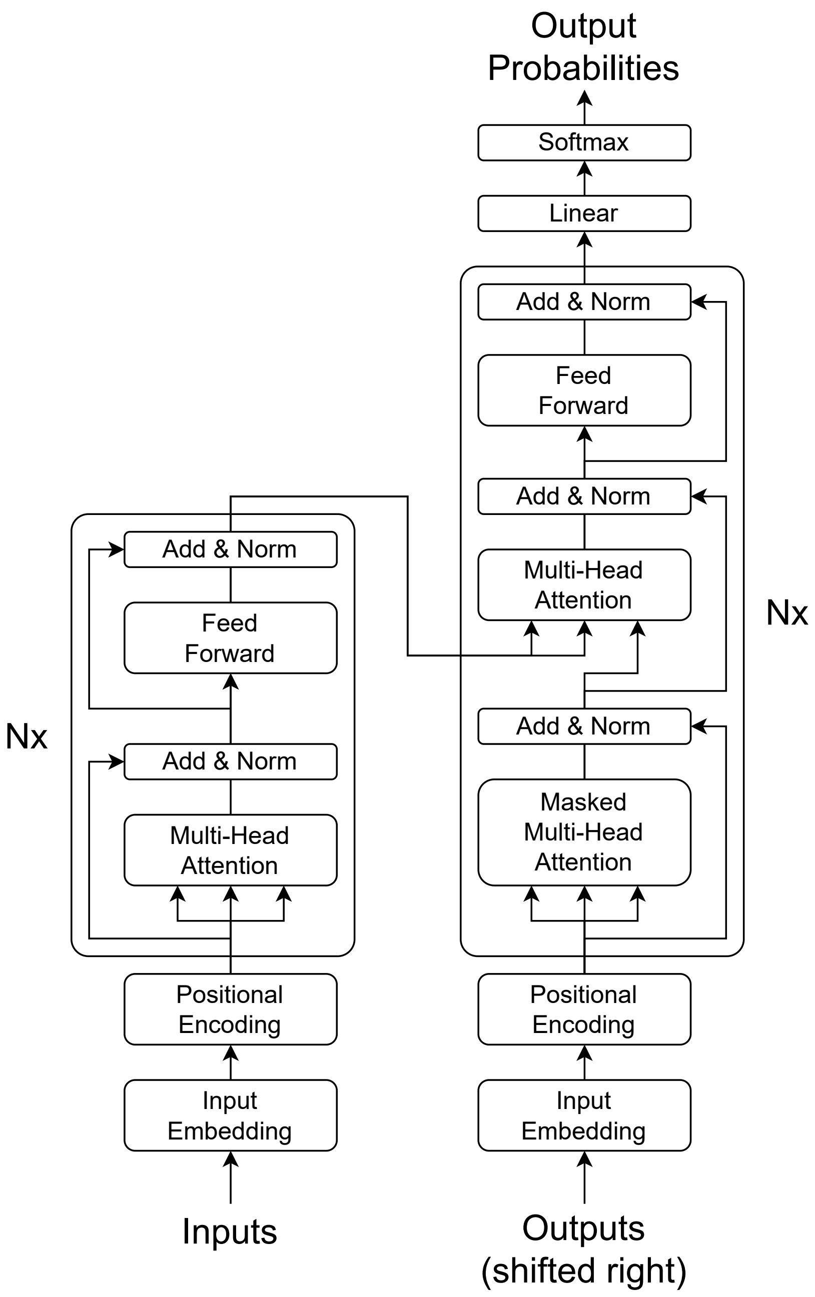


Figure 6: Transformer Architecture

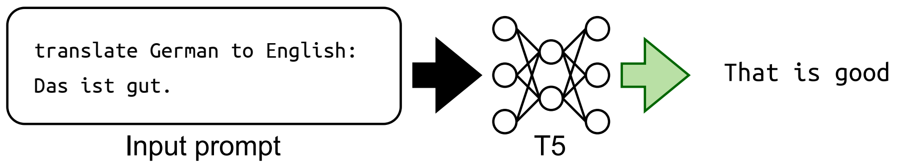


Figure 7: German to English Language Translation Task Performed by T5 Model

The encoder utilizes the multi-head mechanism that allows the model to focus on relevant parts of the input sequence, capturing dependencies between words irrespective of their positions in the sequence. The decoder also utilizes the multi-head mechanism like the encoder, producing the output sequence based on the entire context of the input and previously generated words.

Further development of the transformer model, known as the Bidirectional Encoder Representations from Transformers (BERT) [19] model, has utilize two-stage training: pre-training for language understanding and fine-tuning for developing downstream tasks. During the pre-training phase, the model is trained with unlabeled data, treating the training data as labeled data. The volume of data required for this phase is substantial, hence it is named the Large Language Model. In the subsequent phase, the model's final layer is fine-tuned using traditional labeled data to generate answers for the intended downstream task. Numerous studies have adopted this two-stage training approach, including the Text-To-Text Transfer Transformer (T5) [20] model and the Generative Pre-trained Transformer (GPT) [21] model. These models have consistently demonstrated state-of-the-art performance across various downstream tasks, for example, Fig. 7 shows one of the T5 downstream tasks, language translation from German to English.

2.2.3 Code LLM

The success of the transformer-based model has inspired researchers from various fields to adapt transformer models and train them with domain-specific data, such as image processing, audio processing, and software engineering. Instead of training transformer models with human languages, software engineering researchers train them with programming languages and replace vocabulary with syntax lists. The performance of these models is comparable to their counterparts, LLMs, and they

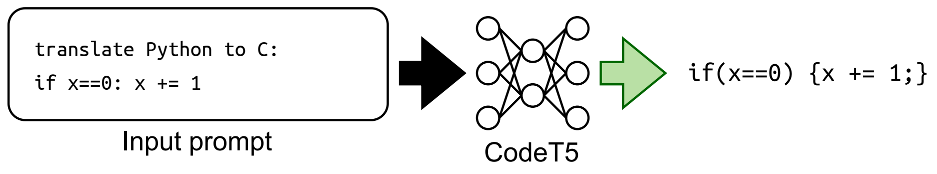


Figure 8: Python to C Programming Translation Task performed by CodeT5 Model

surpass previous studies in software engineering tasks.

Researchers from Google Brain trained BERT with the Python language without altering the BERT architecture, calling it Code Understanding BERT (CuBERT) [22]. The five examples demonstrated by CuBERT outperform previous studies. Another model developed by Salesforce is CodeT5 [23]. This model is based on the T5 model, pre-trained with various programming languages and code comments. Since the model is unaltered, it retains the ability to comprehend both human and programming languages. The downstream tasks that the CodeT5 model can perform include Natural Language (NL) to Programming Language (PL) tasks, PL to NL tasks, and PL to PL tasks, such as code generation from code comments, generating code comments from code, and code translation as shown in Fig. 8. These downstream tasks outperform previous studies, as indicated by multiple benchmarks.

2.3 Related work

2.3.1 Automatic Parallelization Using Static Analysis

Each automatic parallelization tool has distinct capabilities and functionalities that are explained in the following.

Automatic parallelization with Intel Compilers [10, 11] is a feature that transform single-threaded source code into parallel source code. This is the most simplest automatic parallelization tool, the parallel program will be create from single-threaded source code without need to modify the original source code. The developers need to enable the automatic parallelization function with the compiler flags, "`-parallel`" in Linux, at the compile time. The compiler will read the input single-threaded source code, perform the dataflow analysis to identify the safe parallelizable loops and deploy the cost model to determine which loops will be parallelize based on potential performance gains, partitions the data for threaded code generation as needed in programming with OpenMP directives, and create the parallel program binary that incorporate the input single-threaded source code with generated OpenMP directives. Automatic parallelization with Intel compiler can run on multiple operating system that Intel provide the compiler, Linux, MacOS, and Windows.

Cetus [24, 25] is the source-to-source compiler infrastructure to transform the C source code, one of the Cetus function is automatic parallelization. This tools is written in Java, can run with any machine that have JVM supported. Cetus deploy privatization, reduction, and alias analysis to identify and parallelize the potential parallelizable loops.

Rose [26,27] is the source-to-source compiler to transform the C/C++ language. The static analysis techniques deploy by ROSE is dependence analysis that utilizes a Gaussian elimination algorithm to determine parallelized loops by solving a set of linear integer equations of loop induction variables to access arrays in the loops.

DawnCC [28] is the source-to-source compiler only for automatic parallelization that support C/C++ languages. The distinguish of DawnCC from other automatic parallelization is it support various parallel programming support, OpenMP, OpenCL, and CUDA. DawnCC deploy the symbolic range analysis to determine parallelizable source code and transform it into parallelized source code.

Clava [9,29] is the recent source-to-source compiler that had multiple functions written in Java, allow it to run in any machine that support by JVM. To run any functions of Clava, user need to develop a configuration file contain query statements for select part of the input source code and feed it to the Clava function for analysis and transform. One of Clava function name AutoPar is the automatic parallelization function to parallelize the source code deployed with three algorithm consist of multiple static analysis techniques. The first and second algorithm utilize scaler privatization analysis and array privatization analysis and if the first two algorithm could not determine the query code. The third algorithm will be apply with scalar and array reduction analysis. After the finish analyzing, AutoPar-Clava generate the OpenMP directives utilize the dataflow analysis and dependence analysis and insert the OpenMP directives at the locate parallelizable loops.

2.3.2 LLM and Code LLM

BERT [19] or Bidirectional Encoder Representations from Transformer is one of the further development of transformer model. This model introduce additional tasks to pre-training phase, Masked Language Model (MLM) and Next Sentence Prediction (NSP). MLM involves randomly masking a portion of input tokens and training the model to predict these masked tokens using contextual clues. This task facilitates the model's to understand word relationships within a sentense. In NSP, the model is given pairs of sentences and trained to predict whether the second sentence follows the first. This task allows the model to understand sentence-level relationships. In the fine-tuning phase, BERT can be fine-tuned for specific tasks by adding a simple output layer on top of the pre-training model because BERT is a encoder-only model, making it highly versatile for various NLP applications such as question answering, sentiment analysis, and named entity recognition.

T5 [20] or Text-To-Text Transfer Transformer is one of the further development of transformer model. This model try to simplify tasks by framing all as text-to-text problems with encoder-decoder architecture. This model try to increase the downstream tasks, like summaization, question answering, and text classification using a single, unified model and training procedure.

CuBERT [22] or Computer Understand BERT is the un-altering BERT architecture trained with computer languages, specifically designed to understand and process programming languages by treating code as a sequence of tokens analogous to natural language. The principal aim of CuBERT is to enhance the performance of various software engineering tasks by leveraging the strengths of the BERT model, effectively capture the syntactic and semantic information intrinsic to programming languages. Like BERT, CuBERT also has two-stages training, pre-training, and fine-tuning. In the pre-train phase, CuBERT is pre-trained on an extensive corpus of Python code, enables CuBERT to develop a comprehensive understanding of Python's structure and patterns. In the fine-tuning phase, the pre-trained model is adapted to specific software engineering tasks using labeled datasets, code completion, bug detection, code summarization, code classification, and code translation. CuBERT's efficacy has been validated through various benchmarks and evaluations, demonstrating significant improvements over previous models.

CodeT5 [23] is based on un-altering T5 architecture trained with various computer languages. Like CuBERT, it treat code as a sequence of tokens comparable to natural language. CodeT5 that inherit the fundamental from T5 model approach code-related tasks as text-to-text problems, NL-to-PL, PL-to-NL, and PL-to-PL. In the pre-traine phase, CodeT5 train on an extensive corpus of diverse programming languages and their associated comments. In the fine-tune phase, CodeT5 adapt the pre-trained model to specific tasks using labeled datasets, such as, code summarization, code generation, and code translation. CodeT5's performance has been evaluated with various code-related tasks. This demonstrates its effectiveness in understanding and generating code.

CodeT5+ [30] is an advancement of CodeT5 by integrating advanced methodologies and employing a more extensive and varied corpus of programming languages and code-related data. A distinguishing feature of CodeT5+ is its have multiple modes, encoder-decoder, encoder-only, decoder-only. Each mode capability is suit for different downstream tasks. Encoder-only suit for tasks relate to retrival and detection. Decoder-only suit for tasks relate to code generation. Encoder-decoder is suit for tasks relate to code summarization. CodeT5+ performance demonstrates its effectiveness in each type of tasks that suit their mode.

2.3.3 Automatic Parallelization Using LLM

The HPC community has started to explore the feasibility of utilizing Code LLMs for automatic parallelization. OMPGPT [31] uses Code LLM for generating OpenMP directives for a given code snippet. OMPGPT is based on the GPT-Neo 2.7B model downsized to 0.76B parameters and trained using the HPCorpus dataset[2]. They proposed Chain-of-OMP, based on the Chain-of-thought [32], a step-by-step approach to help generate OpenMP directives more accurately by prompt the model to generate the OpenMP directives three times. First, generating the directive prefix that is "#pragma omp". Second, generating the

[2]https://github.com/Scientific-Computing-Lab-NRCN/HPCorpus

main directives and clauses. Last, generating the control structure of the OpenMP directives. They evaluated the model by generating the OpenMP directives and compared it with the split HPCorpus dataset by using the F1 score and accuracy. The baseline model they compared with is GPT-3.5. The result of OMPGPT has generated the OpenMP directives more accurately than the GPT-3.5.

HPC-Coder [33] is another approach to generate source codes parallelized with OpenMP. The authors gathered source codes from public GitHub repositories and fine-tuned three models, which are GPT-2 1.5B, GPT-Neo 2.7B, and PolyCoder 2.7B, and compared them with the PolyCoder model without further fine-tuning with the gathered dataset. The model is supplied with the name, arguments, and comment of a function and generates the function body parallelized with OpenMP. The results showed that the fine-tuned PolyCoder achieves the highest accuracy.

While the above approaches leverage GPT-based models, which are decoder-only architectures, the proposed method is built upon the T5 model, an encoder-decoder architecture. The encoder-decoder framework of T5 allows for a more comprehensive understanding of the input code such as the context and semantics of the code, enhancing the model's ability to generate accurate and effective OpenMP directives. Consequently, the proposed approach is expected to yield more reliable and contextually appropriate OpenMP directives.

3 Parallelizable For-loop Classification Model

3.1 Introduction

One of the critical steps in the parallelization process is conducting a loop dependence analysis. This analysis is essential because loops with dependencies cannot be parallelized without specific considerations or modifications. Loop dependencies occur when tasks within a loop rely on the results of other tasks, necessitating that these tasks wait for one another, thereby obstructing the possibility of concurrent execution.

The degree to which a program can achieve speedup through parallelization heavily depends on the number of parallelizable loops identified in the source code. If all potential loops are parallelized, the program can significantly benefit from increased execution speed. Conversely, if parallelizable loops are overlooked, the program may not achieve the desired speedup, undermining the benefits of parallelization efforts. Hence, the identification of potential parallelizable loops is a crucial determinant in realizing the full advantage of parallel programming.

In practice, the expertise required to identify parallelizable loops is not always present within development teams. This gap can be addressed by automatic parallelization tools designed to assist in the identification and transformation of serial loops into parallel constructs. However, the effectiveness of these tools is often limited by the capabilities of the underlying static analysis techniques. These techniques may not always produce optimal results, and in some cases, parallelized code may perform worse than its single-threaded version due to various inefficiencies.

To address these challenges, this study introduces an automatic parallelization tool enhanced by a code LLM specifically fine-tuned with an OpenMP source code dataset. This tool aims to classify whether a given for-loop can be parallelized or not. The base model employed in this study is CodeT5, a transformer-based LLM pre-trained with various programming languages code including C language. By leveraging the classification capabilities of this model, developers can more accurately and efficiently transform single-threaded programs into parallel programs, thereby optimizing performance and reducing development time.

This chapter details the development and implementation of this approach, providing a comprehensive analysis of its effectiveness in identifying parallelizable loops. The rest of the chapter is organized as follows. Section 3.2 explains the data collection and labeling, and fine-tuning workflow of the CodeT5 model. Section 3.3 provides the evaluation setup and dataset. Section 3.4 presents the evaluation results. Section 3.5 discusses the advantages and disadvantages of using our approach, and Section 3.6 concludes this chapter and discusses future work.

3.2 Methodology

3.2.1 Overview

This section describes the processes involved in developing the parallelizable for-loop classification model. Fig. 9 illustrates the steps of this study. Initially, C/C++ source files containing OpenMP directives are gathered from public GitHub repositories. For-loops are then extracted from these collected source files and labeled based on the presence or absence of OpenMP directives. Finally, the labeled for-loops are used to fine-tune a pre-trained CodeT5 model to classify whether a given for-loop can be parallelized or not.

3.2.2 Data Collection

To the best of current knowledge, there is no open dataset specifically dedicated to building models capable of predicting parallelizable for-loops (i.e., a collection of labeled for-loops). An extensive data collection process was implemented using the GitHub code search API, focusing on files that utilize OpenMP directives. This section details the data collection methodology.

The GitHub API code search function, while offering a wealth of data, does not support searching for codes with certain keywords in combination with complex search queries, such as creation date, size, or number of stars. Furthermore, the output limit is restricted to 1,000 results, making direct extraction of source files containing OpenMP directives challenging. Therefore, data collection was conducted in the following steps: 1) collect C/C++ repositories, 2) identify repositories containing OpenMP directives, and 3) extract files containing OpenMP directives.

The first step was to identify suitable C/C++ repositories. Using the GitHub

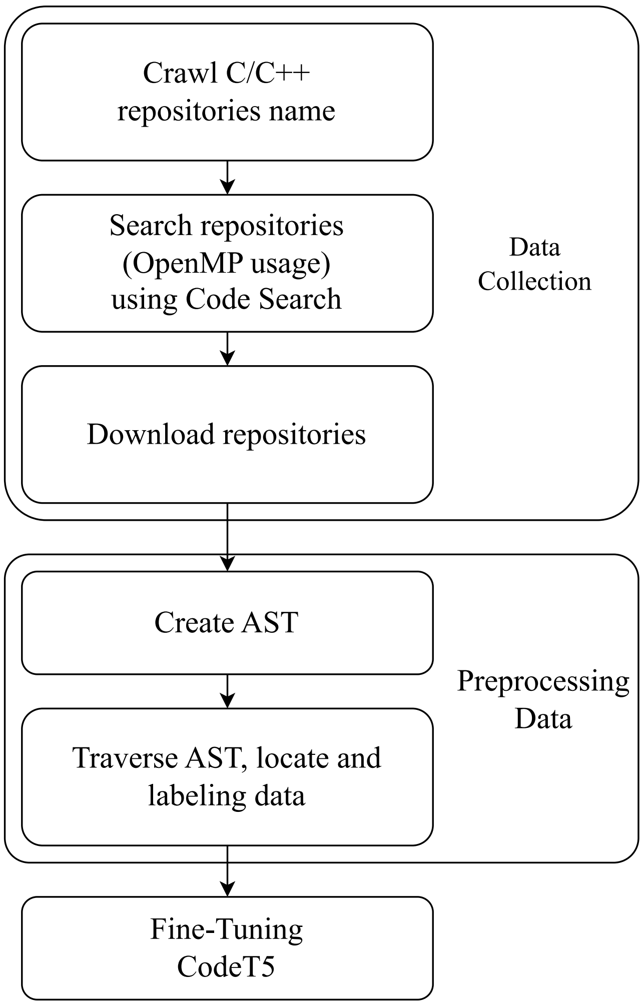

Crawl C/C++ repositories name
Search repositories (OpenMP usage) using Code Search
Download repositories
Data Collection
Create AST
Traverse AST, locate and labeling data
Preprocessing Data
Fine-Tuning CodeT5

Figure 9: The Overview of the Parallelizable For-loop Classification Model

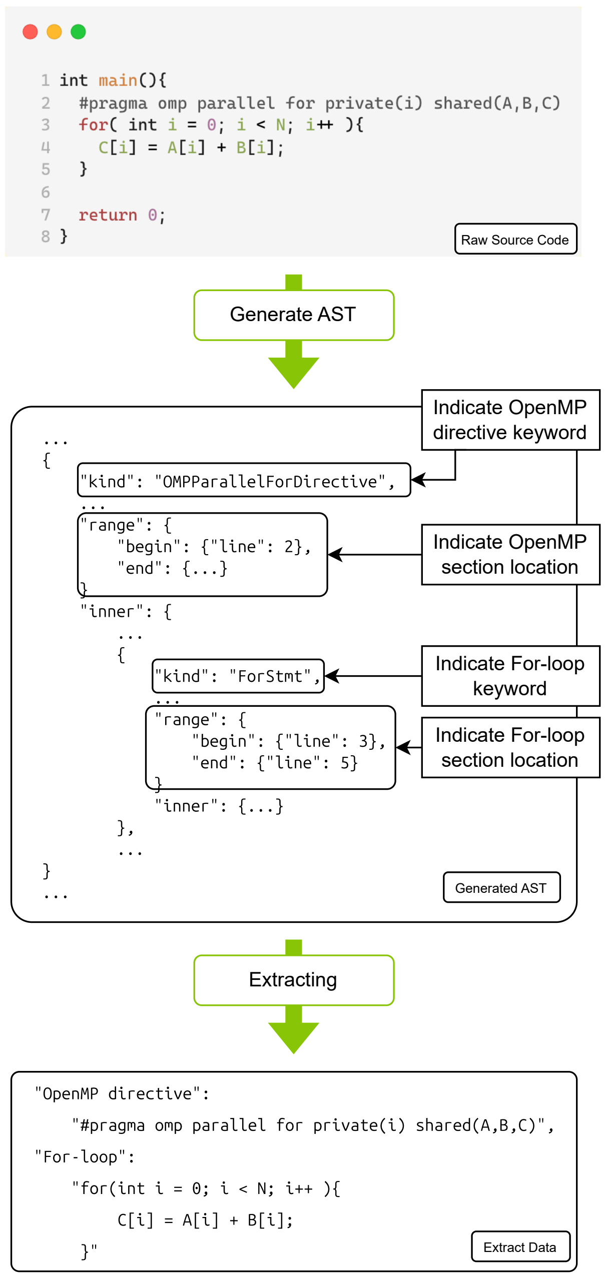
API, all C/C++ repositories on GitHub were searched, avoiding the output limit constraint. To ensure a certain level of code quality [34], repositories with 10 or fewer stars were excluded. After securing this subset of repositories, another search was performed to identify repositories showing traces of OpenMP usage by searching for the keyword "`#pragma omp`" with the GitHub API code search. This process resulted in identifying 2,249 C and 6,836 C++ repositories that use OpenMP. All of these repositories were subsequently downloaded for further analysis.

Within the downloaded repositories, the next step was to extract the files relevant to the research. Files containing the string "`#pragma omp`" were searched, as they indicate OpenMP utilization. The reason for focusing only on these files is twofold: 1) They represent code written by developers familiar with OpenMP, hence ensuring a level of quality in parallelization. 2) While non-parallelizable for-loops could technically be gathered from files without the "`#pragma omp`" keyword, these files might have been written by developers not well-acquainted with OpenMP and may have quality issues. For example, they might contain many loops that should have been parallelized but were not. For these reasons, the focus was only on files where OpenMP was used. After filtering, approximately 140,000 files were collected.

3.2.3 Data Labeling

The process of extracting for-loops from the collected files and categorizing them into two classes: parallel or serial for-loops, involves several steps. These steps are outlined depicted in Fig. 10. First, the Abstract Syntax Tree (AST) of the source code is generated using Clang, a compiler front end for the C, C++, and Objective-C programming languages. Clang provides tools for source code analysis, making it suitable for generating ASTs. This AST provides a detailed syntactic structure and location information for each code element for identifying and analyzing the for-loops and their associated OpenMP directives within the source code.

Traversing the AST, nodes representing for-loops and OpenMP directives are located. Specifically, for-loops are identified by `ForStmt` nodes, while OpenMP directives are identified by nodes with names beginning with `OMP` and ending with


Raw Source Code
Generate AST
Indicate OpenMP directive keyword
Indicate OpenMP section location
Indicate For-loop keyword
Indicate For-loop section location
Generated AST
Extracting
Extract Data

```
int main(){
  #pragma omp parallel for private(i) shared(A,B,C)
  for( int i = 0; i < N; i++ ){
    C[i] = A[i] + B[i];
  }

  return 0;
}
```

```
...
{
  "kind": "OMPParallelForDirective",
  ...
  "range": {
      "begin": {"line": 2},
      "end": {...}
  }
  "inner": {
      ...
      {
        "kind": "ForStmt",
        ...
        "range": {
            "begin": {"line": 3},
            "end": {"line": 5}
        }
        "inner": {...}
      },
      ...
}
...
```

```
"OpenMP directive":
    "#pragma omp parallel for private(i) shared(A,B,C)",
"For-loop":
    "for(int i = 0; i < N; i++ ){
         C[i] = A[i] + B[i];
    }"
```

Figure 10: Locating For-loop and OpenMP Directive and Extracting Data

`Directive` (e.g., `OMPParallelForDirective`, `OMPParallelForSimdDirective`, `OMPGenericLoopDirective`). For each identified for-loop and directive, key information such as the start line, end line, and the content (i.e., source code) is recorded. During this traversal, each node is also annotated with the level of nested loop. While this nesting level data is not utilized during the fine-tuning of the parallelizable for-loop classification model, it will be used in determining which levels of nested loops should be parallelized.

The extracted for-loops are then labeled as either parallel or serial based on the parent node within the AST. If the parent node corresponds to an OpenMP directive, the for-loop is labeled as parallel. Otherwise, it is labeled as serial.

To ensure the integrity of the dataset, any for-loops or content that do not adhere to language syntax or contain invalid OpenMP directives are excluded. This exclusion prevents the model from learning invalid syntax, thereby maintaining the quality and reliability of the classification model.

3.2.4 Fine-tuning the CodeT5 Model

To predict the parallelizability of for-loops, a pre-trained transfer model is fine-tuned using the custom-built dataset. The CodeT5 model, developed by Salesforce, is employed for this task due to its state-of-the-art performance on various code-related tasks across multiple programming languages [23]. Specifically, the smallest version of this model, CodeT5-small, is utilized to reduce the computational cost associated with fine-tuning. Although using a smaller model may compromise performance compared to larger versions, it provides a baseline performance consistent with previous studies utilizing CodeT5 [35].

The training data structure comprises two components: the for-loop code snippets and their corresponding labels. Each for-loop code snippet includes only the for-loop section, formatted as a string. These snippets are labeled as either "0" or "1", where "0" denotes a non-parallelizable for-loop and "1" denotes a parallelizable one, as illustrated in Fig. 11. Prior to fine-tuning the model, each code snippet is tokenized, converting the code into a sequence of tokens. This tokenization is essential for processing the snippets in the model.

Despite the model being pre-trained on 1,000,000 C source files, it lacks training on C++ source files [23]. Given that C++ is a superset of C with similar syntax, additional pre-training on C++ source files was deemed unnecessary in this study.

To manage memory constraints during training, the gradient accumulation technique is employed. This technique involves dividing a large batch into several smaller mini-batches. Instead of updating the model weights after each mini-batch, the gradients are accumulated over several mini-batches. After accumulating the gradients over a pre-defined number of mini-batches, the model weights are then updated. This approach effectively simulates a larger batch size without requiring excessive memory, thus enabling the training process to handle larger datasets more efficiently.

Additionally, early stopping is implemented to prevent overfitting. Early stopping involves monitoring the model's performance on a validation dataset during training. If the model's performance on the validation dataset does not improve for a specified number of iterations, the training process is halted. This technique ensures that the model does not continue to learn from the training data to the point where it starts to perform poorly on unseen data, thus improving its generalization capability.

3.3 Evaluation Setup

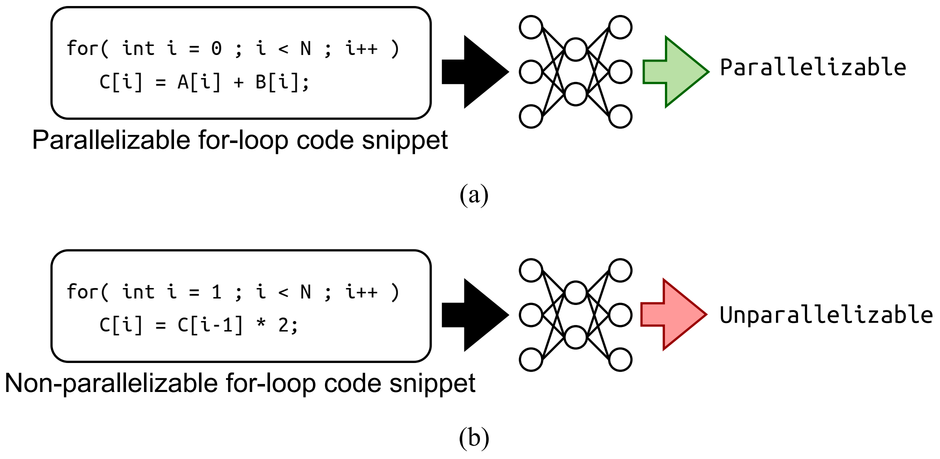
3.3.1 Hardware and Software

The fine-tuning process was conducted on a high-performance server specifically configured to handle intensive computational tasks. This server was equipped with dual Intel Xeon Gold 6230R CPUs, 256 GB of RAM, and two NVIDIA A100 40 GB PCIe GPUs.

Clang version 16 was utilized for parsing the source codes and extracting the ASTs. The CodeT5 model fine-tuning was performed using the Transformers library version 4.26, PyTorch version 1.13, and CUDA version 11.6.

3.3.2 Fine-Tuning

The dataset curated for fine-tuning consists of 2,017,111 parallel for-loops and 1,117,493 serial for-loops. However, training a model on this imbalanced dataset could lead to biased performance. To address this issue, an under-sampling approach was implemented to balance the dataset. Specifically, 500,000 parallel



```
for( int i = 0 ; i < N ; i++ )
   C[i] = A[i] + B[i];
```

```
for( int i = 1 ; i < N ; i++ )
   C[i] = C[i-1] * 2;
```

Parallelizable
Parallelizable for-loop code snippet
(a)
Unparallelizable
Non-parallelizable for-loop code snippet
(b)

Figure 11: (a) Parallelizable code snippets are labeled as “1” (b) Non-parallelizable code snippets are labeled as “0”

for-loops and 500,000 serial for-loops were selected, not only balancing the dataset but also reducing the overall training time.

The balanced dataset was then partitioned into training and testing subsets in an 80:20 ratio. The partitioning process ensured that the proportionality between parallel and serial for-loops was maintained within both subsets, providing a consistent representation of both classes during training and evaluation.

The fine-tuning of the CodeT5 model involved experimenting with various batch sizes: 128, 256, 512, and 1,024. The learning rate was fixed at 2.0×10^{-4} for all configurations. Through these experiments, the optimal number of epochs was determined for each batch size: 12 epochs for batch sizes of 128 and 256, 25 epochs for a batch size of 512, and 10 epochs for a batch size of 1,024, respectively.

3.3.3 Evaluation Targets

This study evaluates the proposed parallelizable for-loop classification model using the following two distinct targets to ensure robustness and reliability in various scenarios.

GitHub Projects Dataset: This was specifically constructed as the testing dataset for this research and consists of a balanced set of parallelizable and serial for-loops. To mitigate bias, any duplicate for-loops were removed, resulting in a final refined testing set of 50,000 parallelizable for-loops and 50,000 serial for-loops. This dataset was employed to assess the model's capability in predicting whether developers had parallelized a given for-loop with OpenMP directives.

NAS Parallel Benchmarks: The second target is the NAS Parallel Benchmarks (NPB) [36, 37], a renowned and widely accepted suite for parallel performance evaluation. Although the official NPB is developed in Fortran, this study utilizes its C++ port [36] for the evaluation. The benchmarks include single-threaded source code as well as parallelized versions based on different parallel programming models, including an OpenMP version. It facilitates a thorough assessment of the model's performance.

The NPB suite is composed of eight distinct benchmarks, categorized into kernels and pseudo-applications. Each category is designed to test different aspects of parallel computing performance and capability. The kernels consist of five benchmarks: Embarrassingly Parallel (EP), Multi-Grid (MG), Conjugate Gradient (CG), discrete 3D fast Fourier Transform (FT), and Integer Sort (IS). The pseudo-applications consist of three benchmarks: Block Tri-diagonal solver (BT), Scalar Penta-diagonal solver (SP), and Lower-Upper gauss-seidel solver (LU).

3.3.4 Baseline

In this study, Clava [9] was utilized as the baseline for comparison. Clava is a source-to-source compiler that includes an integrated library named AutoPar, which is specifically designed to facilitate automatic parallelization through static analysis. Clava is recognized for achieving state-of-the-art performance in this domain.

To accurately replicate the results obtained by Clava, a custom script was developed to extract for-loops from NPB Benchmarks. The script extracts all the loops, identifies parallelizable for-loops, formulates a corresponding OpenMP directive for each parallelizable for-loop, and when encountering a nested loop structure, retains the outermost parallelized loop while commenting out the inner one.

3.3.5 Performance Metrics

The performance of binary classification in determining whether a loop is parallelizable or serial was evaluated using four key metrics: accuracy, F1 score, precision, and recall.

Accuracy measures the overall correctness of the classification model in distinguishing between parallel and serial loops. It is calculated using the following formula:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}. \tag{1}$$

F1 score is the harmonic mean of precision and recall, offering a balance between the two. Precision indicates the accuracy of the predicted parallel loops, while recall assesses the completeness of identifying all parallel loops. Given the trade-off between precision and recall, the F1 score provides a single metric that balances both. The formulas for these metrics are as follows:

$$\begin{aligned}
\text{F1} &= \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, & (2)\\
\text{Precision} &= \frac{\text{Number of correctly predicted parallel for loops}}{\text{Total number of predicted parallel for loops}}, & (3)\\
\text{Recall} &= \frac{\text{Number of correctly predicted parallel for loops}}{\text{Total number of actual parallel for loops}}. & (4)
\end{aligned}$$

Note that Clava analyzes the source code and suggests suitable OpenMP directives for parallelization (e.g., "`#pragma omp parallel for`", "`#pragma omp for`" and "`#pragma omp for private(i)`"). However, the primary focus of this evaluation is on determining the parallelizability of loops rather than the correctness of the specific OpenMP directives suggested. Therefore, even if Clava proposes an incorrect OpenMP directive, the classification is considered correct as long as the loop is identified as parallelizable.

3.4 Evaluation Results

The results on two evaluation targets, GitHub Projects and NAS Parallel Benchmarks are detailed below.

Table 1: GitHub Project Evaluation Results

Model Batch size	Accuracy	F1	Precision	Recall
128	0.841	0.859	0.771	0.970
256	0.842	0.858	**0.779**	0.954
512	**0.843**	0.859	**0.779**	0.950
1024	0.841	**0.860**	0.770	**0.973**

Table 2: NPB Benchmark Evaluation Results

Model Batch size	Accuracy	F1	Precision	Recall
128	0.912	0.549	**0.893**	0.396
256	**0.943**	**0.764**	0.858	0.688
512	0.936	0.744	0.811	0.688
1024	**0.943**	0.737	0.793	0.688
Clava	0.905	0.669	0.628	**0.716**

3.4.1 GitHub Projects

Table 1 provides an evaluation of the proposed models' performance on the dataset of GitHub projects, specifically analyzing the impact of different batch sizes during training. Note that the baseline model requires compilable source code; however, a significant portion of the dataset comprises non-compilable files. Consequently, the baseline performance results are not included in this comparison.

The performance metrics indicate that all models achieve high accuracy, approximately 0.84. Notably, the model trained with a batch size of 1,024 outperforms others, recording the highest F1 score of 0.860 and a recall of 0.973. Although the variation between the highest and lowest performance scores is minimal, less than 2%, models trained with larger batch sizes (i.e., 512 and 1,024) exhibit marginally superior results in terms of both accuracy and F1 score.

3.4.2 NAS Parallel Benchmarks

Table 2 presents a detailed performance evaluation of the proposed models on the NPB benchmarks, compared against the baseline approach, AutoPar-Clava. The models demonstrated high accuracy scores, approximately 0.9, across all batch sizes, consistent with the findings from the evaluation on the dataset of GitHub projects. However, for metrics such as F1 score, precision, and recall, the variance was more pronounced in the evaluation with NPB compared to that with the GitHub dataset.

Specifically, the model trained with a batch size of 128 exhibited a 21.5% difference in F1 score compared to the best-performing model, while other models showed a gap of around 3% in their F1 scores. Additionally, the recall of AutoPar-Clava was superior by 2.8% compared to the best-performing proposed model. These variations highlight the sensitivity of certain performance metrics to batch size adjustments in the training process.

In addition, the model with a batch size of 256 achieved the highest accuracy and F1 score, highlighting its overall effectiveness. Interestingly, the model with a batch size of 128 stood out in terms of precision but had significantly lower recall scores, adversely affecting its F1 values. This trade-off between precision and recall suggests the need for careful batch size selection depending on the specific performance metric prioritization.

Compared to the baseline model, AutoPar-Clava, the proposed models generally outperformed it across all metrics, except for recall. Even the lowest-performing proposed model, except for the one with a batch size of 128, demonstrated superior performance relative to the baseline. Specifically, the proposed models showed improvements of 2.9% and 6.8% in accuracy and F1 score, respectively, compared to the baseline.

3.5 Discussion

In this discussion section, the evaluation of the proposed model is explored. The analysis is structured into three key sub-sections. Firstly, the correctly predicted results are examined, highlighting the factors contributing to the model's accuracy and identifying patterns that signify parallelizability. Secondly, the incorrectly predicted results are explored to uncover potential limitations and areas for improvement in the model. Lastly, the benefits of the proposed model as a coding companion are discussed, emphasizing its practical applications and potential to assist developers in optimizing their code for parallel execution.

Listing 1: An example of parallelizable code that the proposed method correctly predicted, but Clava did not due to dependency analysis failure (line 616 in rank function from IS benchmark OpenMP version).

```
#pragma omp for schedule(static)
for( i=0 ; i < NUM_KEYS ; i ++ )
{
    k = key_array[i];
    key_buff2[bucket_ptrs[k >> shift]++ ] = k;
}
```

Listing 2: An example of parallelizable code that the proposed method correctly predicted, but Clava did not due to out-of-memory (line 2323 in x_solve function from BT benchmark).

```
#pragma omp for
for( k=1 ; k <= grid_points[2]-2 ; k++ )
{
    for( j=1 ; j <= grid_points[1]-2 ; j++ )
    {
        for( i=0 ; I <= isize ; i++ )
        {
            tmp1 = rho_i[k][j][i];
            tmp2 = tmp1 * tmp1;
            ... 287 more lines
```

3.5.1 Analysis of the Correctly Predicted Results

In the evaluation of the NPB benchmarks, multiple for-loops were identified as parallelizable by the proposed method, whereas Clava did not recognize them as such. This discrepancy can be attributed to two possible reasons.

Limitations of Static Analysis Static analyzers often struggle to identify parallelizable loops when runtime information or application-specific knowledge is required. An example is shown in Listing 1, where the data dependency analysis by Clava incorrectly deems a loop as non-parallelizable. In this case, the "`key_buff2`" array is indexed using the value from the "`bucket_ptrs`" array, which is determined by the value of "`k`" shifted right by "`shift`" bits. Despite Clava's assessment, the loop is indeed parallelizable, as evidenced by the presence of the "`schedule(static)`" directive in the corresponding OpenMP annotation.

This example highlights the limitations of static analysis, which can fail when dynamic or context-specific information is not available.

Computational Complexity of Static Analysis Another significant challenge for static analyzers like Clava is the computational complexity involved in analyzing deeply nested loops. Listing 2 illustrates a case where Clava encounters an out-of-memory exception while processing a code snippet with multi-level nested loops, with the loop body spanning approximately 300 lines. Clava's approach involves exhaustively analyzing all dependencies within the loops to determine their parallelizability, leading to excessive memory consumption and eventual failure. In contrast, the proposed model maintains consistent memory consumption, regardless of the input size or complexity, due to its data-driven nature.

Superior Performance of the Proposed Model The superior performance of the proposed model compared to Clava can be attributed to the extensive and diverse source code repositories used to fine-tune the CodeT5 model. While Clava relies on a rigid, rule-based approach, the proposed model leverages the embedded knowledge on parallelization extracted from a vast array of source codes. This flexibility allows the model to dynamically adapt and provide accurate predictions without being constrained by predefined rules.

The effectiveness of the proposed model is intrinsically linked to the quality and quantity of the training data, which encompass a wide variety of parallelization patterns. Unlike predefined rule-based approaches that can only address a limited set of patterns, the data-driven approach of the proposed model enables it to recognize and adapt to a broader range of parallelization scenarios. This adaptability is illustrated in Listing 1, where the model successfully identifies parallelizable loops that do not conform to Clava's rigid rule set.

3.5.2 Analysis of the Incorrectly Predicted Results

The evaluation results, as presented in Table 2, indicate that the recall scores of the proposed model are notably low. Specifically, the model with a batch size of 128 achieves a recall score of only 0.396. In comparison, Clava outperforms

Listing 3: Example of a parallel nested for-loop

```
#pragma omp parallel for
for( i=1 ; i<n ; i++ )
{
    for( j=1 ; j<m ; j++ )
    {
        for( k=1 ; k<o ; k++)
        {
            // Do something
        }
    }
}
```

other models with a margin of 2.8

The current training dataset comprises only code snippets of for-loops, lacking the broader context or inherent characteristics of the surrounding code. For example, the preprocessing approach currently in use decomposes nested for-loop code snippets, as illustrated in Listing 3, into separate data entities. In the original source code, only the outermost loop has the OpenMP directive and is labeled as "parallelizable" in the dataset. The inner loops, however, are labeled as "unparallelizable." This labeling approach fails to consider that if these inner loops appeared independently in the source code, they might be deemed suitable for parallelization and deserve their own OpenMP directives.

This discrepancy highlights a crucial issue: the decision to parallelize a for-loop is often context-dependent, influenced by its relationship with other loops. Ignoring this broader context introduces significant challenges in model training, leading to inaccurate predictions.

One possible remedy is to exclude inner loops from the training data when they are part of parallelized outer loops. While this approach might prevent the inclusion of misleading data, it would also eliminate valuable information about contextual decisions, thereby reducing the training dataset's comprehensiveness.

To address this issue fundamentally, a reconfiguration of the training dataset is essential. The model should be trained not only on isolated code snippets but also within the context of their encompassing code structure. Incorporating information about nesting levels and the parallelizability of outer loops could significantly enhance the model's understanding of the broader context. This would enable more accurate predictions regarding whether a loop should be parallelized with OpenMP directives.

Incorporating these contextual details into the training process could involve several strategies. One approach could be to label nested loops based on their potential for parallelization in isolation and within their broader context. Another strategy might include training the model on entire code segments, highlighting the relationships between nested loops and their parallelization directives. By adopting these methods, the model can better grasp the complexities of loop parallelization decisions, improving its ability to predict accurately.

3.5.3 Benefits of the Proposed Model as a Coding Companion

The proposed model for classifying parallelizable for-loops offers significant benefits to developers by assisting in the identification of potential parallelization opportunities within source code. The high F1 scores achieved in both GitHub projects and NPB evaluations highlight the model's effectiveness and reliability.

One of the primary advantages of this model is its utility as a learning tool for novice developers. By integrating this model into a pair programming setup, beginners can expedite the process of developing parallel programs. The model serves as an intelligent coding companion, highlighting loops that can be parallelized and providing insights into parallelization strategies. This hands-on guidance helps beginners understand the characteristics of parallelizable code, thereby enhancing their coding skills and knowledge.

Additionally, the model aids in the educational process by offering practical examples for users to study and learn from. By practicing with real-world code, developers can gain experience in identifying potential parallelizable for-loops, gradually building their proficiency in parallel programming. This iterative learning process not only accelerates skill acquisition but also instills confidence in applying parallelization techniques independently.

Furthermore, the model's ability to provide consistent and accurate suggestions reduces the likelihood of overlooking parallelization opportunities. In complex codebases where manual identification can be challenging and error-prone, the model ensures thorough analysis and identification of parallelizable loops. This leads to more efficient and optimized code, ultimately enhancing the performance of software applications.

3.6 Conclusion

This research introduced a deep learning-based NLP model designed for the automatic parallelization of source code. By fine-tuning CodeT5, a state-of-the-art transformer model, using source code from GitHub repositories containing OpenMP directives, the model has been trained to effectively identify loops suitable for parallelization.

The evaluation of the proposed model was conducted using source files from GitHub projects and the NPB. The results demonstrate that the proposed model outperforms the baseline method, AutoPar-Clava. Specifically, the model achieved an F1 score of 0.860 in the GitHub projects evaluation and an F1 score of 0.764 in the NPB evaluation. In comparison, Clava attained an F1 score of 0.669 in the NPB evaluation.

These findings highlight the effectiveness of the deep learning-based approach in identifying parallelizable loops within source code. The model's superior performance can be attributed to the comprehensive training dataset, which included diverse parallelization patterns from real-world codebases, enabling the model to generalize well across different contexts. The significant improvement in F1 scores indicates the model's potential to enhance the parallelization process, making it a valuable tool for developers aiming to optimize their code for parallel execution. This research highlights the importance of leveraging advanced NLP techniques and extensive training data to address complex problems in software optimization.

4 OMP-CodeT5+: An OpenMP Directive Generation Model

4.1 Introduction

In the previous chapter, a parallelizable for-loop classification model was developed as the first function of the proposed automatic parallelization tool. The subsequent function of this tool involves transforming the identified parallelizable code segments to achieve parallel execution, specifically through the insertion of OpenMP directives.

Enhancing program performance through parallelization necessitates meticulous modification of the identified code segments. The effectiveness of the performance gain depends on the precision of the parallel annotations introduced by the tool. For OpenMP, this process involves the strategic selection of directives and clauses, which are fundamental for the tool to determine the appropriate method for parallelizing the source code.

Incorrect or incompatible directives and clauses can significantly impact the performance of the parallelized code. Potential issues include lack of speedup, compilation failures due to syntax errors, incorrect outputs, and in the worst-case scenario, unexpected behaviors leading to inaccurate results. Thus, the accuracy of these directives and clauses is critical for achieving proper parallelization.

The selection of suitable OpenMP directives and clauses is not a trivial task. It requires a deep understanding of both the application's computational patterns and the underlying hardware architecture. For instance, the "`#pragma omp parallel for`" directive is commonly used for distributing loop iterations across multiple threads. However, the choice of additional clauses such as "`private`", "`firstprivate`", and "`reduction`" must align with the data dependencies and the specific requirements of the code segment. Misalignment can lead to race conditions, inefficient memory usage, or incorrect results.

Furthermore, the hardware's characteristics, such as the number of available cores and the memory hierarchy, must be taken into account when selecting directives and clauses. For example, the "`schedule`" clause can be used to control the distribution of loop iterations to threads, which can have a significant impact on load balancing and cache utilization. Incorrect scheduling strategies can result in suboptimal performance or even degraded performance compared to serial execution.

The second function of automatic parallelization tool involves generating parallel modifications for the input source code segment. Different tools employ various approaches to achieve this goal, with static analysis techniques being particularly effective. For instance, AutoPar-Clava utilizes a parallelization engine that incorporates dataflow analysis and dependency analysis, both of which are categorized as static analysis techniques. However, these analyzers are limited by the predefined rules encoded by the developers, similar to the limitations discussed in the previous chapter.

This chapter focuses on the second model, OpenMP directive generation, referred to as OMP-CodeT5+. This model leverages a fine-tuned CodeT5+ [30] model to generate appropriate OpenMP directives for given for-loop code snippets, as illustrated in Fig 3. CodeT5+ is a pre-trained Code LLM on multiple programming languages, including C and C++, making it well-suited for this task.

The OMP-CodeT5+ model works in tandem with the parallelizable for-loop classification model to achieve the objectives of the automatic parallelization tool, as depicted in Fig 13. This LLM-based tool aims to generate more efficient parallel programs using OpenMP, enhancing the overall performance and efficiency of the code.

The rest of the chapter is structured as follows: Section 4.2 explains the data collection, labeling, and fine-tuning workflow of the CodeT5+ model. Section 4.3 provides the evaluation setup and evaluation material. Section 4.4 presents the evaluation results. Section 4.5 discusses the performances gained from parallelizing the source code by using the proposed approach in each category of speedup. Section 4.6 concludes this chapter and discusses future work.

4.2 Methodology

4.2.1 Overview

The development process of the proposed OMP-CodeT5+ model is outlined in this section, as depicted in Fig. 12. This figure illustrates the detailed stages involved in the development of OMP-CodeT5+ model, which encompass data

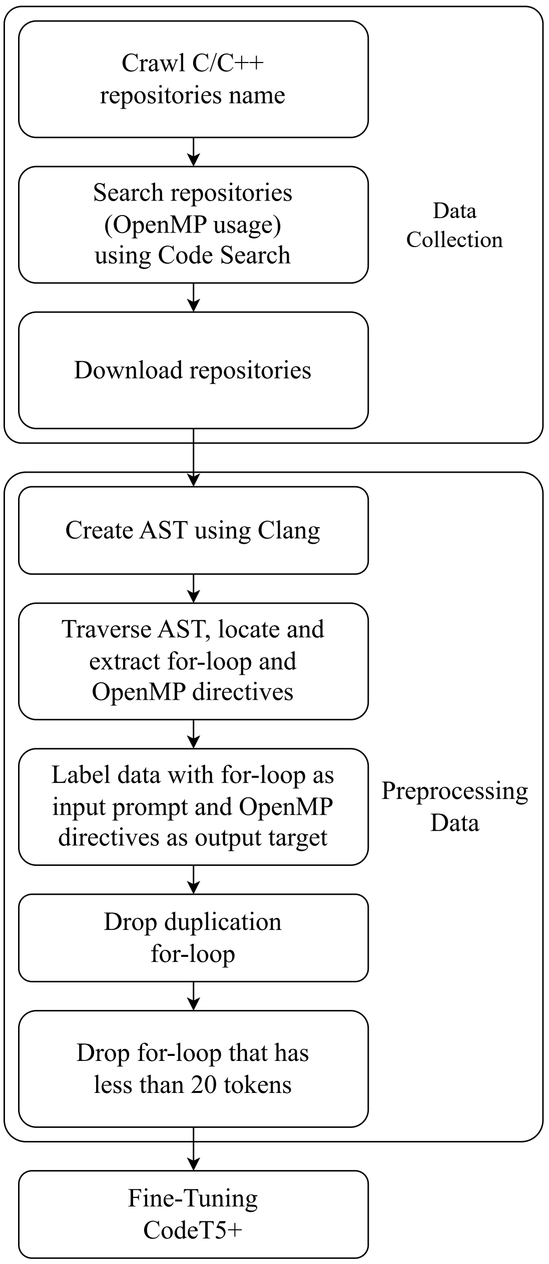
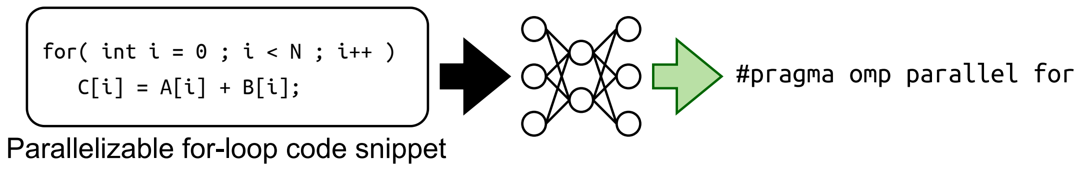

Crawl C/C++ repositories name
Search repositories (OpenMP usage) using Code Search
Download repositories
Data Collection
Create AST using Clang
Traverse AST, locate and extract for-loop and OpenMP directives
Label data with for-loop as input prompt and OpenMP directives as output target
Drop duplication for-loop
Drop for-loop that has less than 20 tokens
Preprocessing Data
Fine-Tuning CodeT5+

Figure 12: Overview of OMP-CodeT5+ model Processes



```
for( int i = 0 ; i < N ; i++ )
   C[i] = A[i] + B[i];
```

```
#pragma omp parallel for
```

Figure 13: Overview of OMP-CodeT5+ model Processes

```
#pragma omp target teams distribute parallel for simd
for (int i=0; i < 2; ++i)
    a = 2;
```

Figure 14: Testing OpenMP For-loop Found In Microsoft/clang Repository, File: test/OpenMP/target_teams_distribute_parallel_for_simd_ast_print.cpp Line: 113-115

collection, preprocessing, and model fine-tuning. The OMP-CodeT5+ model leverages the same dataset containing OpenMP directives, as detailed in Chapter 3. By fine-tuning the pre-trained CodeT5+ model with this dataset, the model is developed to understand parallelizable source code and generate appropriate OpenMP directives. The detailed process is explained in the following sections.

4.2.2 Data Collection and Preprocessing Process

The dataset utilized in this chapter follows the same collection methodology as outlined in Chapter 3. However, the labeling technique differs from the one employed for the model in Chapter 3. The model in Chapter 3 was designed for binary classification to determine whether a given loop could be parallelized. Consequently, the labels were binary, indicating either parallelizable or non-parallelizable loops.

In contrast, the model addressed in this chapter focuses on generating OpenMP directives. This requires a different labeling approach, where the training data for each loop includes the specific OpenMP directive associated with that loop. The shift from binary labels to detailed OpenMP directive labels demands a comprehensive preprocessing phase to ensure the dataset's integrity and relevance. During preprocessing, each loop is carefully examined to extract its corresponding OpenMP directive, which serves as the target output for the model during training.

The dataset undergoes thorough cleaning and formatting to ensure that the input loops and their associated directives are accurately aligned.

To further enhance the dataset's quality, several measures are implemented to remove duplicates and low-quality code snippets. Duplicates are identified and removed to prevent redundancy. It is common to encounter identical for-loops across different projects, and retaining such redundant data can degrade the dataset's overall quality. Additionally, low-quality code snippets, such as test cases or machine-generated source code, are filtered out. A specific criterion is employed to identify low-quality snippets: the number of tokens within each for-loop snippet is counted, and those with fewer than 20 tokens are discarded. This threshold is based on the minimum number of tokens required for a meaningful for-loop code snippet containing a single operation. Fig. 14 illustrates an example of such a low-quality snippet. By filtering out these snippets, the dataset's quality is improved, ensuring that only relevant and high-quality data is used for model training.

4.2.3 Model Training

To develop the OMP-CodeT5+ model, the pre-trained CodeT5+ was fine-tuned using the dataset developed in this study. There are multiple variations of the CodeT5+ model with different sizes, including CodeT5+ embedding model, CodeT5+ bimodal model, CodeT5+ fine-tuned for Python, and instruct CodeT5+. The bimodal version was selected for this study due to its efficacy in code generation tasks.

The fine-tuning of the CodeT5+ bimodal model, which comprises 220 million parameters (220M) [30], introduces considerable challenges, especially when compared to the previously utilized CodeT5 small model, which has only 60 million parameters (60M) [23]. Initial attempts to fine-tune the bimodal model on the existing same hardware were met with out-of-memory errors, highlighting the need for advanced memory management strategies.

To address these challenges, memory optimization techniques were employed, particularly ZeRO-Infinity [38–40]. ZeRO-Infinity, implemented through the DeepSpeed library [41], enables efficient memory usage by transferring only the essential parts of model computation to the GPU from the CPU main memory.

This approach significantly reduces the GPU memory footprint, facilitating the training of larger models on consumer-grade GPUs that would typically require industrial-grade hardware.

4.3 Hardware and Software Setup

4.3.1 Model Training

The OMP-CodeT5+ model was fine-tuned using a high-performance workstation featuring an Intel Core i9 10900K processor, 64 GB of RAM, and a GeForce RTX 4090 GPU with 24 GB of memory. Clang 16 was utilized for parsing the source codes and generating ASTs. The fine-tuning process was conducted using the Transformers library version 4.34.1, integrated with PyTorch 2.1.1, CUDA 12.2, and DeepSpeed 0.12.3.

4.3.2 Parallelized Code Evaluation

Performance evaluation of both the parallelized and baseline source codes was conducted on a server equipped with two Intel Xeon Gold 6230R CPUs and 384 GB of memory. Intel oneAPI 2022 was utilized to compile and execute the parallelized source code. This setup enabled a comprehensive comparison of the parallelized implementation against existing methods and the baseline implementation.

4.3.3 Fine-tuning

The dataset used for fine-tuning comprised 37,946 for-loops as prompts, paired with their corresponding OpenMP directives as labels. This dataset was divided into a training set and a testing set, containing 33,883 and 4,063 samples, respectively.

The fine-tuning process was configured with a batch size of 32, which represented the maximum feasible size for executing CodeT5+ training in the given environment. An initial learning rate of 5×10^{-5} was set to facilitate steady convergence. Gradient accumulation was set to 4 to effectively manage memory usage during training. Additionally, ZeRO Optimization Stage 2 was employed, utilizing the CPU offload configuration to enhance memory efficiency. The `AdamW` optimizer was used, known for its effectiveness in handling large-scale deep learning tasks.

4.3.4 Evaluation Method for Automatic Parallelization Tools

To assess the performance improvements achieved by various automatic parallelization tools, including the proposed method, the NAS Parallel Benchmarks (NPB) were employed. These benchmarks, previously utilized in the model described in Chapter 3, are essential for evaluating the enhancements in performance that arise from applying parallelization techniques, such as OpenMP, to the code.

The evaluation process involved several key steps. First, the serial versions of the NPB benchmarks were parallelized using each automatic parallelization tool. Subsequently, the runtimes of the parallelized benchmarks were measured. This comparative analysis of runtimes enabled a thorough assessment of the effectiveness of the code or executables generated by each tool. This evaluation methodology is consistent with the approach described in the existing literature on Clava.

It is important to highlight a common challenge encountered during the evaluation: the directives generated by both the proposed model and other tools can occasionally contain syntax errors or fail verification. In such cases, the problematic directives were commented out to ensure that the parallelized code was compilable and could successfully pass benchmark verification.

By following this rigorous methodology, the performance improvements offered by different automatic parallelization tools were systematically evaluated, providing valuable insights into their effectiveness and reliability in enhancing computational performance through parallelization.

4.3.5 Baseline Methods Compared with the Proposed Model

To assess the performance of the proposed model, it was compared against two baseline automatic parallelization tools: the Intel compiler-supported automatic parallelization and Clava, a state-of-the-art source-to-source compiler. The Intel compiler's automatic parallelization feature has been evaluated in existing literature on Clava [29], making it a suitable baseline for comparison.

Intel Compiler-supported Automatic Parallelization The Intel compiler provides an automatic parallelization feature that uses static analysis techniques to identify parallelizable sections of code. For this evaluation, the NPB was utilized with two different par-threshold configurations: 0 and 100. The default par-threshold setting of 100 directs the compiler to parallelize loops only when there is a high likelihood of significant performance gains. Conversely, a par-threshold of 0 instructs the compiler to parallelize all loops deemed safe to parallelize, regardless of the anticipated performance benefits. This configuration aligns with the settings used in the existing evaluation of Clava.

AutoPar-Clava Clava is an advanced source-to-source compiler that also leverages static analysis for automatic parallelization. Specifically, the AutoPar library within Clava was used, which is designed to automatically parallelize source code using OpenMP directives. For this evaluation, a script was developed to extract for-loops parallelized in the original OpenMP NPB benchmarks. AutoPar then assesses the parallelizability of each extracted for-loop and generates the appropriate OpenMP directives.

4.4 Evaluation Results

4.4.1 Performance Evaluation Using Benchmarks

The speedup achieved by each automatic parallelization tool across different benchmarks is illustrated in Fig. 15. The speedup results are categorized into three levels: high, medium, and no speedup. This classification helps in understanding the effectiveness of each tool across different benchmarks.

The benchmarks MG and CG demonstrate notable speedups, with the OMP-CodeT5+ model achieving over 10× improvements compared to their single-threaded versions. These benchmarks fall into the high speedup category. Specifically, the CG benchmark shows a speedup of 24.23× with the OMP-CodeT5+ model. However, this is lower than the speedups achieved by tthe original OpenMP CG implementation, AutoPar-Clava, and Intel compiler with 0 par-threshold, which are 55.96×, 55.53×, and 47.42×, respectively. In contrast, the Intel compiler with a par-threshold of 100 shows no speedup for the CG benchmark.

In the SP benchmark, the OMP-CodeT5+ model, AutoPar-Clava, and Intel compiler (with both 100 and 0 par-thresholds) achieve moderate speedups of 1.67×, 1.69×, 2.00×, and 1.86×, respectively. For the FT and LU benchmarks, the OMP-CodeT5+ model achieves speedups of 1.2× and 1.3×, respectively, which

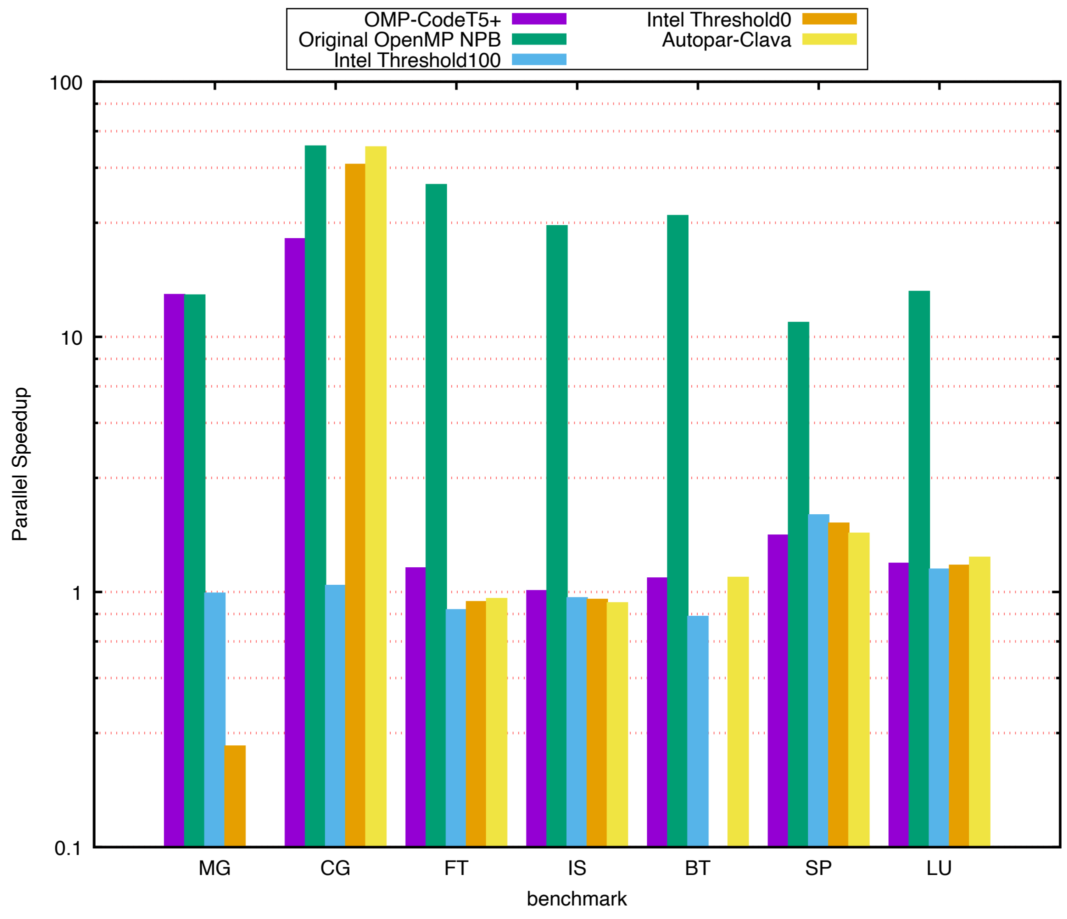


Figure 15: Speedup Over the Original Serial NPB Runtimes Using Various Automatic Parallelization Tools and Manual Parallelization

are categorized as medium speedup. Notably, other automatic parallelization tools result in lower performance than their single-threaded versions for the FT benchmark.

The IS and BT benchmarks do not exhibit significant speedups compared to manually parallelized implementations. Particularly for the BT benchmark, the Intel compiler with a par-threshold of 0 results in a considerable slowdown, achieving only 0.09× the performance of the single-threaded version.

Table 3: Evaluation of Syntax Correctness and Benchmark Verification for Generated OpenMP Directives

Name	**# of Parallel**	**OMP-CodeT5+**		**AutoPar-Clava**	
	For-loops	**Passed**	**Failed**	**Passed**	**Failed**
EP	1	0	1	0	1
MG	11	10	1	-	-
CG	9	9	0	9	0
FT	7	3	4	1	6
IS	3	1	2	0	3
BT	15	12	3	12	3
SP	19	16	3	16	3
LU	13	6	7	7	6
Total	78	57	21	45	22

4.4.2 Evaluation of Syntax Correctness and Benchmark Verification for Generated OpenMP Directives

Table 3 presents an evaluation results in terms of syntax correctness and benchmark verification for NPB. The table categorizes the results into successful (Passed) and unsuccessful (Failed) attempts, with failures arising from either syntax errors or verification errors. The first column lists the specific NPB benchmarks, and the second column indicates the number of for-loops targeted for parallelization within each benchmark. The subsequent columns detail the number of passed and failed attempts for the OMP-CodeT5+ and AutoPar-Clava tools.

The evaluation reveals that both OMP-CodeT5+ and AutoPar-Clava encounter difficulties in parallelizing certain for-loops that were effectively parallelized in the original OpenMP implementations. Notably, AutoPar-Clava failed to analyze and generate directives for the MG benchmark, highlighting a significant limitation in its capability.

Conversely, the OMP-CodeT5+ model demonstrates a higher success rate in producing correct OpenMP directives. Specifically, the OMP-CodeT5+ model successfully parallelized 57 out of 78 targeted loops in the original NPB benchmarks, resulting in a success rate of 73%. In comparison, AutoPar-Clava successfully parallelized only 45 loops, achieving a success rate of 58%.

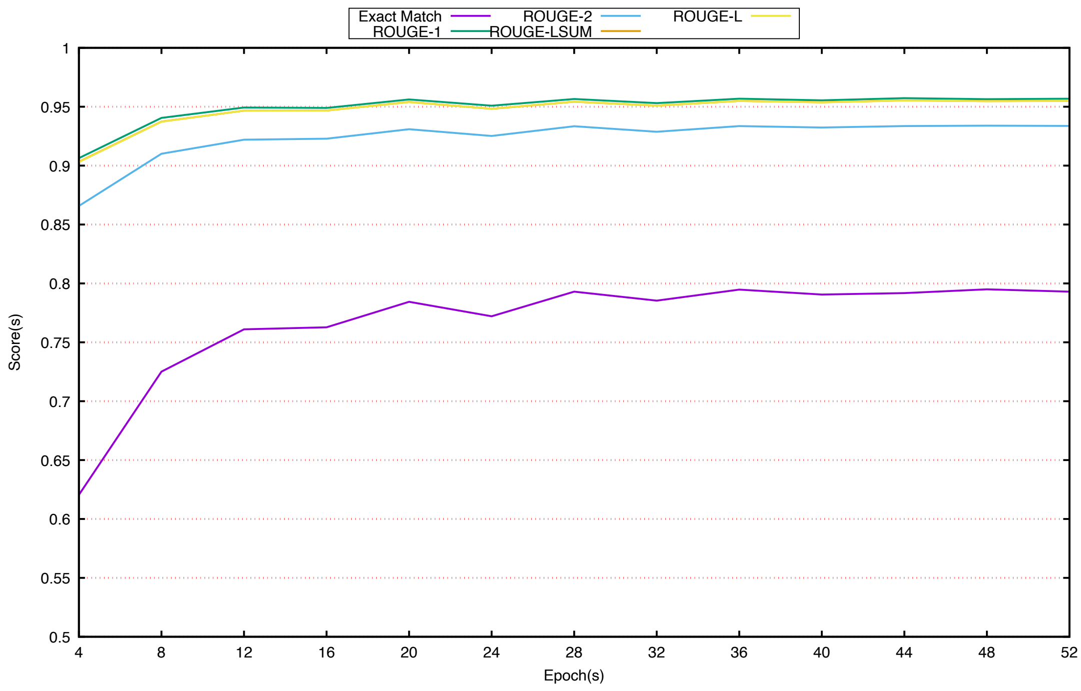


Figure 16: The Exact Match and ROUGE Scores with Split Training Data At Each Epoch

The comparative analysis indicates that while both tools have their strengths and limitations, OMP-CodeT5+ generally outperforms AutoPar-Clava in terms of the number of successful parallelizations. The higher success rate of OMP-CodeT5+ suggests a more robust capability in generating correct OpenMP directives across diverse benchmarks.

4.4.3 The Evaluation Metrics Scores At Each Epoch

Figure 16 presents the evaluation scores, including Exact Match and Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [42], across different epochs during the fine-tuning of the OMP-CodeT5+ model. The Exact Match metric assigns a score of 1 if the generated text matches the labeled string exactly and 0 otherwise.

Another metric, ROUGE is a set of metrics used to evaluate the quality of automatic summarization and machine translation software in natural language processing. It includes four key metrics: ROUGE-1 is unigram (1-gram) based

Listing 4: Generated OpenMP Directive For For-loop in conj_grad() Function in CG Benchmark By The Trained Model

```
#pragma omp parallel for
for(j = 0; j < lastrow - firstrow + 1; j++) {
  sum = 0.0;
  for(k = rowstr[j]; k < rowstr[j+1]; k++){
    sum = sum + a[k]*p[colidx[k]];
  }
q[j] = sum;
}
```

scoring, ROUGE-2 is bigram (2-gram) based scoring, ROUGE-L is longest common subsequence-based scoring, and ROUGE-LSUM is longest sentences common subsequences-based scoring, splits text using "`\n`".

The results indicate that the OMP-CodeT5+ model achieves its best performance at epoch 28, as reflected by both Exact Match and ROUGE scores. After this point, further fine-tuning does not significantly improve the evaluation metrics, suggesting that the model has reached a point of stability.

The total training duration from the first epoch to the 52nd epoch is approximately six hours and 45 minutes. Each fine-tuning epoch takes around 7 minutes and 50 seconds. The evaluation dataset comprises 4,063 for-loops, which were split from the overall dataset for the purpose of these evaluations.

4.5 Discussion

4.5.1 Analysis Through Speedup Categories

This section analyzes the evaluation results based on the previously observed speedup categories: High, Medium, and No Speedup. To provide a detailed analysis, the Intel VTune profiler is employed to examine specific benchmarks within each category.

High Speedup (MG and CG) The OMP-CodeT5+ model demonstrates significant speedups for two benchmarks, MG and CG. Specifically, in the case of MG, the directives generated by OMP-CodeT5+ are highly similar to those in the original OpenMP MG implementation. This alignment enables the parallelized

Listing 5: Original OpenMP CG Source Code in conj_grad() Function

```
#pragma omp parallel private(it,i,j,k)
{
  ...
  conj_grad(...)
  ...
}

static void conj_grad(...){
...
#pragma omp for nowait
for(j = 0; j < lastrow - firstrow + 1; j++) {
  suml = 0.0;
  for(k = rowstr[j]; k < rowstr[j+1]; k++){
    suml += a[k]*p[colidx[k]];
  }
  q[j] = suml;
}
...
}
```

implementation by the OMP-CodeT5+ model to gain speedups comparable to those obtained from manually parallelized code.

Similarly, the speedup of CG parallelized by the OMP-CodeT5+ model is also high. However, it remains lower than that achieved by the original OpenMP implementation, despite the generated directives being largely similar to the original. Profiling revealed a function with significant CPU usage. In the original OpenMP version, as shown in Listing 5, a parallel region is created around the `conj_grad()` function. Conversely, the OMP-CodeT5+ model generates a parallel region for each individual for-loop. This approach introduces additional overhead, thereby degrading the performance of the parallelized code compared to the original OpenMP version.

The primary factor contributing to the high speedup results of the parallelized MG source code is the number of successfully parallelized for-loops. For-loops from the MG source code are mostly short and medium in length, which the OMP-CodeT5+ can parallelize effectively. For the long sequence for-loops, the average token count is 592, and the maximum token count is 641. These long sequence for-loops fall within the range that OMP-CodeT5+ can interpret and

Listing 6: Generated Directive in x_solve Function of SP Benchmark by OMP-CodeT5+ Model

```
void x_solve(){
  int i, j, k i1, i2, m;
  double ru1, fac1, fac2;
  #pragma omp parallel for
  for(k=1; k<nz2, k++){
  ...
}
```

generate OpenMP directives successfully.

In the case of the CG benchmark, the primary factor contributing to the high speedup results of the parallelized CG source code is the number of successfully parallelized for-loops. The for-loops in the CG source code consist solely of short sequences, which allows OMP-CodeT5+ to effectively parallelize all for-loops from the CG benchmark.

Medium Speedup (SP, LU and FT) The proposed model demonstrates less than 2× speedup in three benchmarks: SP, LU, and FT. To analyze the underlying causes of this limited performance enhancement, a detailed analysis was conducted using the SP benchmark as a representative case.

From the profiling of the original OpenMP SP source code, the bottleneck functions are `z_solve` followed by `y_solve` and `x_solve`. The generated directives for `z_solve`, `y_solve` and `x_solve` cause the benchmark to fail the verification. This is due to a race condition because the loop index `k` is not privatized as shown in Listing 6. On the other hand, in the original source code shown in Listing 7, the loop indices are declared in a parallel region and thus automatically privatized. The directives generated by AutoPar-Clava also caused the benchmark to fail verification. These directives, which caused the benchmark to fail verification, had to be removed for performance evaluation, resulting in limited speedup.

The primary factor contributing to the medium speedup result of the parallelized FT source code is the limited number of successfully parallelized for-loops, which accounts for less than half. The OMP-CodeT5+ struggled to parallelize the main computational segments of this benchmark, specifically the medium sequence for-loops that were not successfully parallelized. This limitation impeded

Listing 7: Original x_solve Function of SP Benchmark

```
#pragma omp parallel
{
  adi();
}
...
void adi(){
  compute_rhs();
  txinvr();
  x_solve();
  y_solve();
  z_solve();
  add();
}
...
void x_solve(){
  int i, j, k i1, i2, m;
  double ru1, fac1, fac2;
  #pragma omp for
  for(k=1; k<nz2, k++){
  ...
}
```

the parallelized source code by OMP-CodeT5+ from achieving a higher speedup.

In the case of the LU benchmark, the primary reason for the medium speedup result of the parallelized LU source code is the lower success rate of parallelized for-loops, which is less than half. Most of the for-loops in the LU benchmark are long sequence for-loops, with an average token count of 2,760. The OMP-CodeT5+ is less effective in generating successful OpenMP directives for higher token counts, and half of the failed parallelized for-loops are long sequence for-loops. This factor hinders the speedup of the LU benchmark.

In the case of the SP benchmark, the primary factor contributing to the medium speedup result of the parallelized SP source code, despite the higher success rate of parallelized for-loops, is that the three failed parallelized for-loops are the main computational segments of the benchmark. These three main computational for-loops are long sequences for-loops with 4,081 tokens, 4,151 tokens, and 4,052 tokens. The OMP-CodeT5+ does not effectively generate successful parallelized OpenMP directives for these long sequence for-loops. The speedup of the parallelized SP source code by OMP-CodeT5+ is higher than other medium speedup results due

Listing 8: Parallelized For-loops by The OMP-CodeT5+ Model in the rank Function of IS Benchmark

```
#pragma omp parallel for schedule(static) // failed
for( i=0; i<NUM_KEYS; i++ )
  work_buff[key_array[i] >> shift]++;
...
#pragma omp parallel for schedule(static) // failed
for( i=0; i<NUMKEYS; i++ ){
  k = key_array[i];
  key_buff2[bucket_ptrs[k >> shift]++] = k;
}
...
#pragma omp parallel for schedule(dynamic) // passed
for( i=0; i< NUM_BUCKETS; i++ ) {
  k1 = i * num_bucket_keys;
  k2 = k1 + num_bucket_keys;
  for ( k = k1; k < k2; k++ )
    key_buff_ptr[k] = 0;
  m = (i > 0)? bucket_ptrs[i-1] : 0;
  for ( k = m; k < bucket_ptrs[i]; k++ )
    key_buff_ptr[key_buff_ptr2[k]]++;
  key_buff_ptr[k1] += m;
  for ( k = k1+1; k < k2; k++ )
    key_buff_ptr[k] += key_buff_ptr[k-1];
}
```

to the higher number of successfully parallelized for-loops.

No Speedup (IS and BT) The proposed model was not able to offer any performance improvements for the IS and BT benchmarks. To identify the causes of the lack of speedup, a detailed analysis of the IS benchmark was conducted.

The IS benchmark uses a bucket sort algorithm to sort integers. This is implemented in the `rank` function that contains three loops. The first loop counts the number of keys in each bucket. The second loop stores each key in the appropriate bucket. The last loop sorts the keys within each bucket.

The profiling results from the original OpenMP IS source code showed that the second loop is most time-consuming in the `rank` function, followed by the third and the first loop. The proposed model was able to generate a correct directive for the third loop only. The two generated directives cause race conditions because the loop indices `i` and `k` are not privatized as shown in Listing 8. The original

Listing 9: Parallel For-loops in rank Function from Original OpenMP IS Benchmark

```
#pragma omp parallel private(i,k)
{
...
#pragma omp for schedule(static)
for( i=0; i<NUM_KEYS; i++ )
  work_buff[key_array[i] >> shift]++;
...
#pragma omp for schedule(static)
for( i=0; i<NUMKEYS; i++ ){
  k = key_array[i];
  key_buff2[bucket_ptrs[k >> shift]++] = k;
}
...
#pragma omp for schedule(dynamic)
for( i=0; i< NUM_BUCKETS; i++ ) {
  k1 = i * num_bucket_keys;
  k2 = k1 + num_bucket_keys;
  for ( k = k1; k < k2; k++ )
    key_buff_ptr[k] = 0;
  m = (i > 0)? bucket_ptrs[i-1] : 0;
  for ( k = m; k < bucket_ptrs[i]; k++ )
    key_buff_ptr[key_buff_ptr2[k]]++;
  key_buff_ptr[k1] += m;
  for ( k = k1+1; k < k2; k++ )
    key_buff_ptr[k] += key_buff_ptr[k-1];
}
...
}
```

OpenMP IS source code shown in Listing 9 privatizes `i` and `k` in the outer parallel directive that covers all three loops and thus does not cause race conditions.

AutoPar-Clava also fails to parallelize loops in the `rank` function and offers no speedup. Specifically, Clava refuses to parallelize the first loop with the message, "Array access `work_buff[key_array[i] >> shift]` which is used for writing has a subscript of arrayType `key_array[i] >> shift`." This is because the content of `key_array` cannot be statically determined at compile time and thus Clava cannot verify if the writes to `work_buff` can be performed in parallel. Similarly, the second and third loops employ indirect indexing, which also prevents their parallelization.

These issues highlight a limitation of the proposed approach when applied to codes where loop indices and other variables requiring privatization need to be declared in a parallel region directive separate from the loops themselves. Consequently, the generated directives fail to appropriately privatize these variables, resulting in race conditions and verification errors.

The primary factor limiting the speedup of the parallelized BT source code, despite achieving 80% successfully parallelized for-loops, is that the failed parallelized for-loops reside in separate functions that constitute the main computational segments of the benchmark. These functions execute calculations along each axis in 3D dimensions: the x-axis, y-axis, and z-axis. Each function contains a long sequence for-loops with 4,868 tokens, 4,844 tokens, and 4,855 tokens. The OpenMP directives generated by the OMP-CodeT5+ do not effectively identify the necessary variables and clauses for these long sequence for-loops.

4.5.2 Analysis of the Impact of Training Dataset Characteristics

Analysis Across Different For-Loop Complexities This section explains the generation capability of OMP-CodeT5+ from the input in different complexities in detail. From the NPB source code, this study divides the for-loops sequence into three categories, starting with zero to 100 tokens categories as short sequence, 101 to 500 tokens categories as medium sequence, and more than 500 tokens categories as long sequence as shown in Table 4. Table 4 shows the results of passed and failed generated OpenMP directives from the OMP-CodeT5+ for each benchmark divided into for-loops sequence categories. Fig. 17 shows the overall

Table 4: Number of Passed and Failed Generated OpenMP Directives in Each For-loop Size Category

Name	# of Parallel For-loops	**Short**		**Medium**		**Large**	
		Passed	**Failed**	**Passed**	**Failed**	**Passed**	**Failed**
MG	11	4	0	3	1	4	0
CG	9	9	0	-	-	-	-
FT	7	1	1	2	3	-	-
SP	19	1	0	9	0	6	3
LU	13	1	0	3	1	2	6
IS	3	0	2	1	0	-	-
BT	15	-	-	9	0	3	3
EP	1	-	-	-	-	0	1
Total	78	16	3	26	5	15	13

passed generated OpenMP directives in the percentage of the for-loops sequence.

The speedup results for the MG and CG benchmarks, as illustrated in Fig. 15, correspond to the number of parallelized for-loops listed in Table 4. A detailed examination of the for-loop types in the MG and CG source code reveals that OMP-CodeT5+ effectively parallelizes short and medium-sized loops, which are predominant in these benchmarks. The average token of the long for-loop sequence from the MG benchmark is 555.5 tokens and the longest token is 641 tokens. This is the shortest average token from the long for-loop sequence type.

In the case of medium speedup results, the generated OpenMP directives for the SP, LU, and FT benchmarks fail to fully exploit the potential for increased speedup. Table 4 shows two scenarios contributing to the moderate speedup in these benchmarks. The first scenario involves the FT benchmark, where OMP-CodeT5+ failed to parallelize the medium for-loop sequences, despite the generated OpenMP directives being similar to the original FT benchmark source code. The unsuccessful parallelization occurs in three core functions of the FT computation: "`cffts1`", "`cffts2`", and "`cffts3`". Listing 11 shows parallel for-loop in `cffts1` function from the original OpenMP FT benchmark. This function, called from the parallel directive, allows the variables declaration inside this

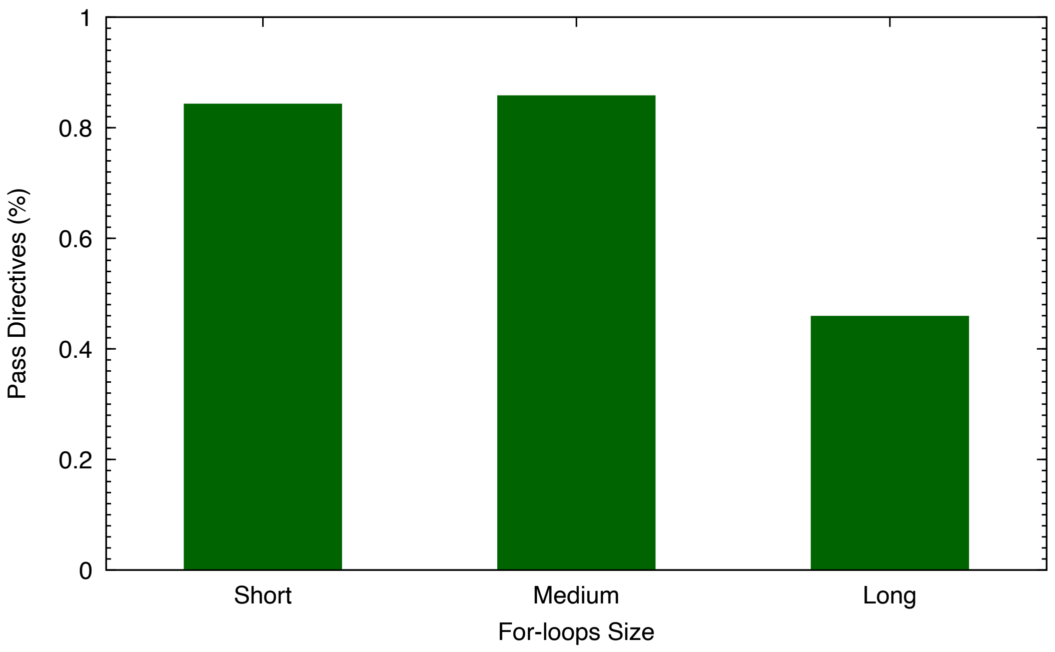


Figure 17: Overall Pass Generated OpenMP Directives in Each Size of For-loop

function to be automatically privatized. However, OMP-CodeT5+'s limitations in code modification resulted in its failure to identify the variables that need to be privatized and shared across threads to ensure the FT benchmark operates correctly. The parallel code in the other two functions, "`cffts2`", and "`cffts3`", are very similar and have the same problems as the "`cffts1`" function.

In the second case involving the SP and LU benchmark, the OMP-CodeT5+ failed to parallelize the long sequence for-loops within these two benchmarks. The average number of tokens in the long for-loop sequences is 1,976 for the SP benchmark and 2,759 for the LU benchmark, with the longest for-loops consisting of 4,151 tokens in SP and 5,282 tokens in LU. For the SP benchmark, the failed parallelizations occurred in the functions, "`x_solve`", "`y_solve`", and "`z_solve`". Similarly, for the LU benchmark, the functions, "`blts`", "`buts`", "`l2norm`", and "`rhs`", experienced failed parallelizations. The OMP-CodeT5+ model cannot recognize the necessity to privatize and share variables within these long for-loop sequences.

The no-speedup result of the BT benchmark shares the same problem with the

Listing 10: Parallel For-loops by The OMP-CodeT5+ Model in the cffts1 Function FT Benchmark

```
#pragma omp parallel for
for(k=0; k<d3; k++){
    for(jj=0; jj<=d2-FFTBLOCK; jj+=FFTBLOCK){
        for(j=0; j<FFTBLOCK; j++){
            for(i=0; i<d1; i++){
                y1[i][j] = x[k][j+jj][i];
            }
        }
        cfftz(is, logd1, d1, y1, y2);
        for(j=0; j<FFTBLOCK; j++){
            for(i=0; i<d1; i++){
                xout[k][j+jj][i] = y1[i][j];
            }
        }
    }
}
```

Listing 11: Parallel For-loops in the cffts1 Function from Original OpenMP FT Benchmark

```
static void cffts1( ... )
{
    int logd1;
    int i, j, k, jj;
    ...
    #pragma omp for
    for(k=0; k<d3; k++){
        for(jj=0; jj<=d2-FFTBLOCK; jj+=FFTBLOCK){
            for(j=0; j<FFTBLOCK; j++){
                for(i=0; i<d1; i++){
                    y1[i][j] = x[k][j+jj][i];
                }
            }
            cfftz(is, logd1, d1, y1, y2);
            for(j=0; j<FFTBLOCK; j++){
                for(i=0; i<d1; i++){
                    xout[k][j+jj][i] = y1[i][j];
                }
            }
        }
    }
}
```

SP benchmark. OMP-CodeT5+ failed to parallelize the long for-loops sequence in the three functions, "`x_solve`", "`y_solve`", and "`z_solve`". They shared the same function name but the algorithm inside is different.

The reason for the poor performance in generating the OpenMP directive for the long for-loops sequence is the original configuration used in creating the tokenized data for the training process, "`max_length`" configuration. This configuration will dictate the maximum size of the generated tokenized training data and the tokenized labeled data. The default setting is 512 tokens long. It impacts the performance of OMP-CodeT5+ from the loss of information in training data that has longer tokens and generating accuracy from the long input prompt. However, the benefit of this configuration is to control memory usage during the fine-tuning phase. Increasing the sequence size of the training data can lead to a longer training time.

Impact of Data Imbalance on Model Accuracy The training dataset, the ratio of the three for-loop sequences shown in Fig. 18. The ratio of the training data is not balanced toward the longer for-loop sequence data, nearly half of the dataset consists of short size for-loops. This indicates that in parallel programs in the real-world scenario, simpler short size for-loops is frequently utilized in the codebase. The medium size for-loops also constitute a significant portion, suggesting that they are also quite common in typical codebases. The reason for a small number of large size for-loops might be their relatively less practical in general scenarios because of the large size for-loops complexity and low reusable in other codebases due to the specific requirement of their functions.

The result shows in Fig. 17, the accuracy of the generated OpenMP directive for large size for-loops is lower than the OpenMP directives generated for the other two sizes. There are three reasons for the lower accuracy. First, large size for-loops constitute only 9.98% of the training dataset. This limited exposure may not be sufficient for the model to learn the complex patterns and intricacies associated with larger loops. Second, given the dominance of short and medium size loops in the training data, the model might be overfitting to these sizes, optimizing its parameters in a way that is less effective for large loops. Last, the tokenized parameter "`max_length`" that limited the large size for-loops to be fully trained,

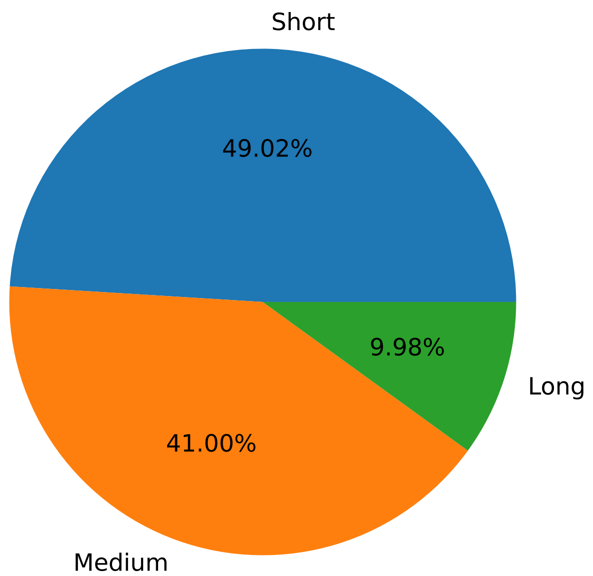


Figure 18: Training Data Ratio divided by Tokens

the default parameter is set at 512 tokens. Most of the failed generated OpenMP directives for large size for-loops have more than 1,500 tokens.

Considering there is a practical use of the large size for-loops, the accuracy for OpenMP directives generated by the OMP-CodeT5+ needs to be increased. There are a few approaches that can be applied to address this problem. Increasing the training data is the easiest approach. However, the current dataset is gathered from public GitHub repositories that have more than 10 stars. The decision to filter only the repositories that have more than 10 stars is the quality and reliability of the code, but this decision led to abandoning the newer code from the newly created repositories. To build the next dataset, the gathering method needs a new approach, to increase more quality and reliability of data from multiple sources.

The second approaches, tuning the parameter "`max_length`" increase the the

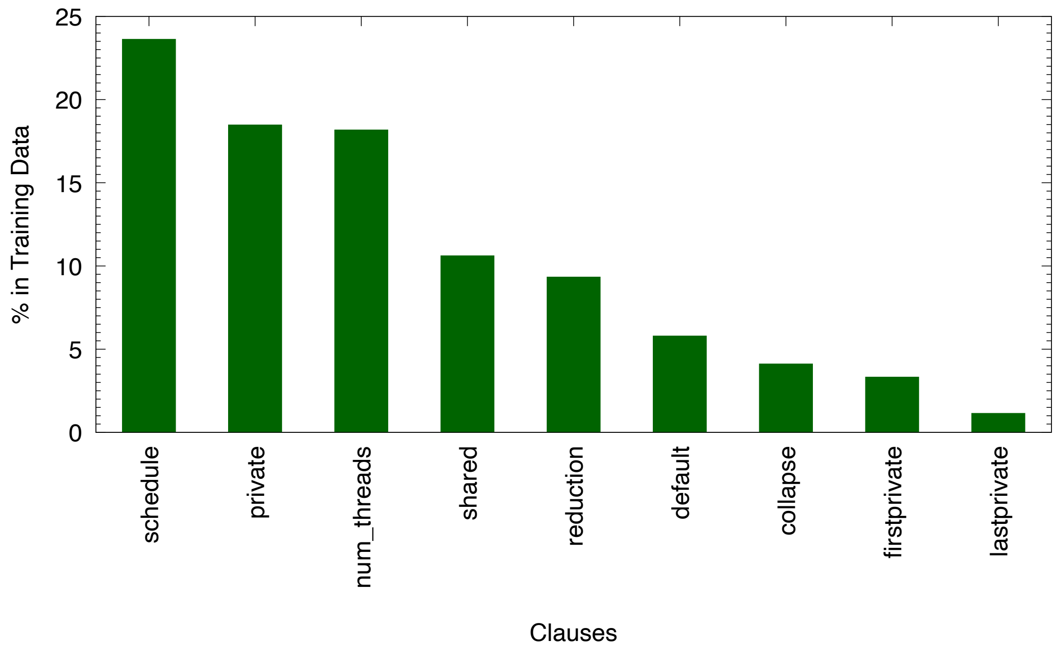


Figure 19: Number of Clauses in Training Data

input tokens for training the OMP-CodeT5+ using the specialized training sessions. Increasing the tokens of training input will impact memory usage during the training and the training time. By conducting targeted training sessions specifically on large for-loops with the increased input token parameter, "`max_length`". This can involve fine-tuning the model using a dataset predominantly composed of large loop examples.

Number of Generated Clauses Fig. 19 shows the number of OpenMP clauses in the training data. The most clauses in the dataset are "`schedule`". This indicates a strong focus on controlling the iteration scheduling of loops in parallel regions. The second and third most used clauses in the, for "`private`" is necessary for parallelizing the source code, it will prevent the data race problem among the thread. For the "`num_threads`", significant presence in the dataset reflects the importance of managing variable scopes and thread counts in parallel programming. The presence of the "`shared`" clause lower than the "`private`" clause, indicates the need to share the data across threads is lower than privatizing the data. The low presence of the "`reduction`" and "`default`" leads to the OMP-CodeT5+

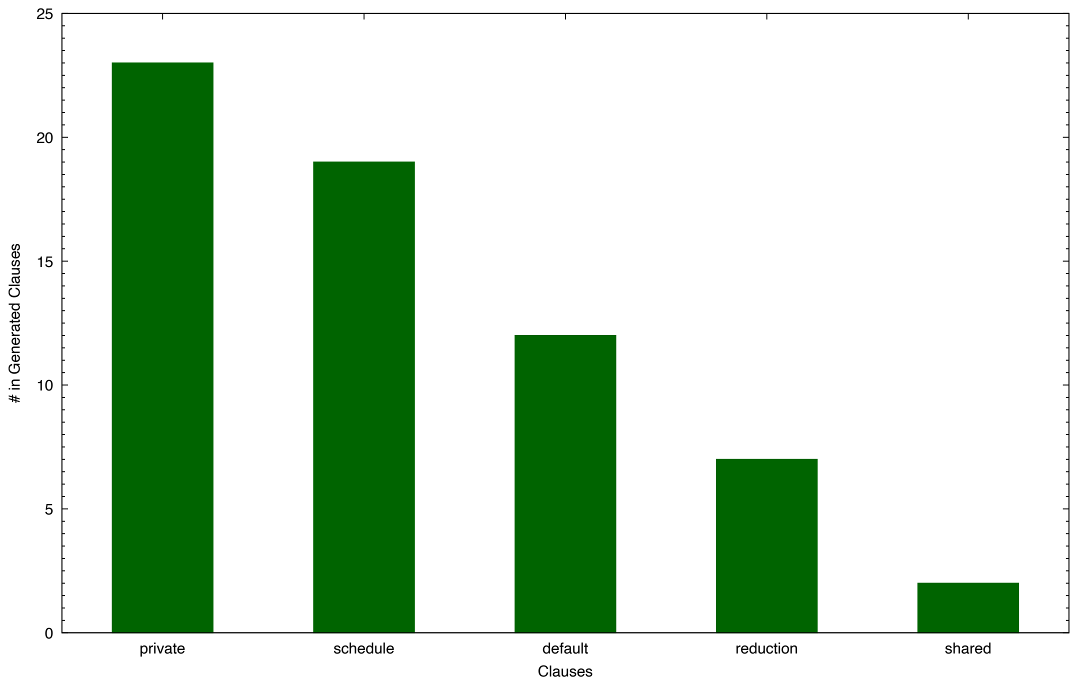


Figure 20: Number of Clauses in Generated Directives

harder to capture the code pattern to identify the need to use this clause. In the rest of the clauses, "`collapse`", "`firstprivate`", and "`lastprivate`", the impact of the very low presence of these three clauses, especially "`lastprivate`" at 1.1% in the training data, cause the OMP-CodeT5+ to cannot capture the code pattern of these three clauses and it get bias from the other clauses like "`private`" and "`schedule`".

Fig. 20 shows the number of generated OpenMP clauses. The high appearance of the "`private`" clause ensures that variables are treated as private to each thread, which is crucial for avoiding data races and ensuring correct parallel execution. This aligns with the number of "`private`" in the training data that reflect the common need in parallel programming to control the scope of variables and prevent unintended sharing among threads. The high number of the generated "`schedule`" indicates a strong emphasis on controlling how loop iterations are divided among threads and align with the number of clauses in training data. The "`default`" clause that appears in the generated results suggests the model recognizes the need to define default data-sharing attributes for variables within parallel regions,

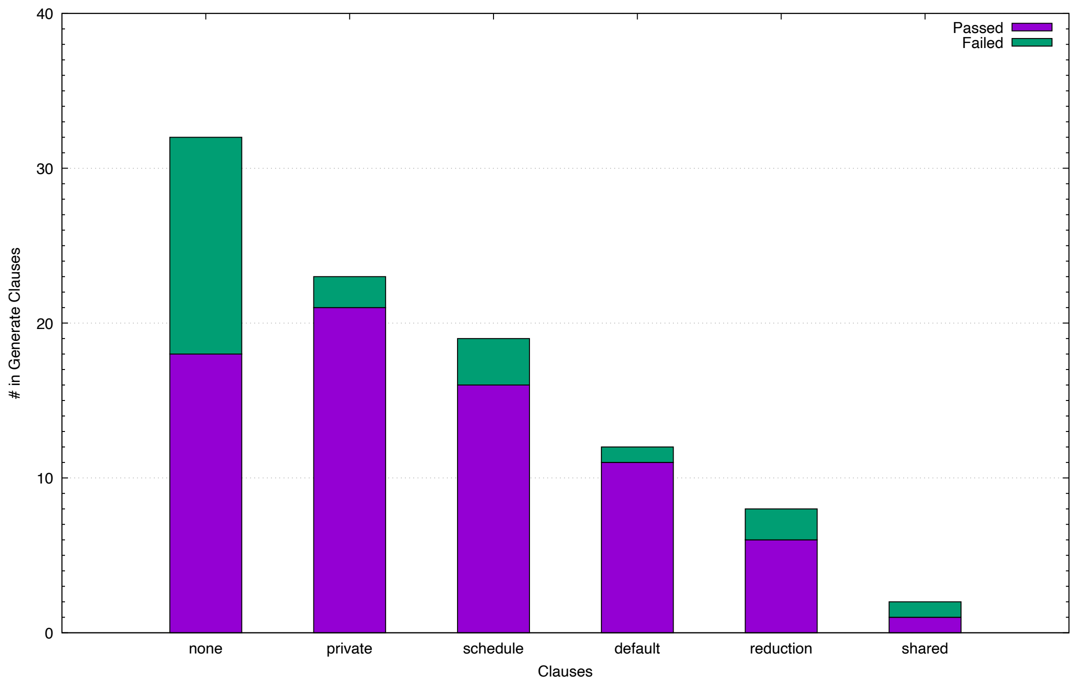


Figure 21: Number of Passed and Failed Clauses in Generated Directives

which can simplify code and reduce the need for explicit declarations. The number of the "`reduction`" clause indicates the model includes functionality to handle reduction operations. For the "`shared`" clause, there are two generated clauses. The main reason is the "`shared`" keyword is the same as the setting used with the "`default`" clause. The model may not acknowledge the difference with the small training data, "`shared`" at 10.6% and "`default`" at 5.7%.

Fig. 21 shows the number of the parallel for-loops that decorate with the OpenMP directives generated by the OMP-CodeT5+ model. The results show that most of the generated clauses, "`private`", "`scheule`", "`default`", and "`reduction`", can parallelize the for-loops without problems. For the "`shared`", this result cannot determine the effectiveness of the generated "`shared`" clause by OMP-CodeT5+ with two cases and 50% accuracy.

4.5.3 Analysis of the Generated Clauses

Generated Scheduling Types For Schedule Clause Fig. 22 shows the ratio of the scheduling type used with the "`schedule`" clause. There are five

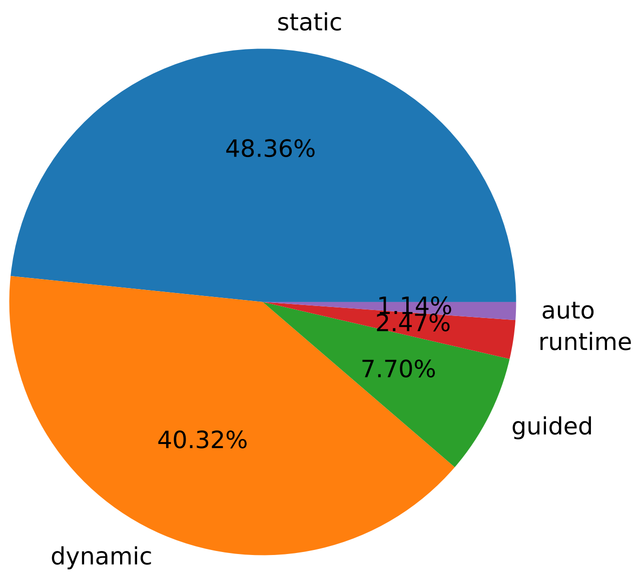


Figure 22: Ratio of Schedule Kind in Training Data

scheduling types for the "`schedule`", "`static`","`dynamic`", "`guided`", "`runtime`", and "`auto`". The number of static and dynamic scheduling keywords data present in the training data is balanced. The other scheduling keywords have low utilization in real-world scenarios. The imbalance of the scheduling keywords in the dataset will cause the model to generate bias toward static and dynamic scheduling. The reason static and dynamic have more usage due that these two scheduling are well-suited for each scenario. The static scheduling is good where a workload of iterations is uniform, meaning each iteration takes approximately the same amount of time to execute. On the other hand, dynamic keyword scheduling is ideal for scenarios with non-uniform workloads where some iterations might take significantly longer to execute than others.

Fig. 20 shows 19 "`schedule`" clause had been generated. In the 19 of the generated "`schedule`" clause, there are two scheduling types that have been gen-

Listing 12: Parallel For-loops Utilize private Clause by The OMP-CodeT5+ Model in the compute_rhs Function BT Benchmark

```
#pragma omp parallel for private(i,j,k,m)
for(k=1; k<=grid_points[2]-2; k++){
    for(j=1; j<=grid_points[1]-2; j++){
        for(i=1; i<=grid_points[0]-2; i++){
            for(m=0; m<5; m++){
                rhs[k][j][i][m]=rhs[k][j][i][m]*dt;
            }
        }
    }
}
```

Listing 13: Parallel For-loops Utilize private Clause by The OMP-CodeT5+ Model in the zero3 Function MG Benchmark

```
#pragma omp parallel for private(i2,i1)
for(i3 = 0;i3 < n3; i3++){
    for(i2 = 0; i2 < n2; i2++){
        for(i1 = 0; i1 < n1; i1++){
            z[i3][i2][i1] = 0.0;
        }
    }
}
```

erated by the OMP-CodeT5+ model, with 18 "`static`", and with one "`dynamic`". However, there are three failed generated OpenMP directives that utilize the "`schedule`" clause. All failed "`schedule`" clauses utilize the static scheduling is appear only 16.67% from the NPB source code. In conclusion, the OMP-CodeT5+ captures the code pattern for utilizing static and dynamic scheduling and can generate the correct scheduling types for the "`scheule`" clause. However, the capability of the OMP-CodeT5+ to generate the less of the scheduling types, guided, runtime, and auto, cannot be determined at this moment because of the generated OpenMP directives from the NPB benchmark do not include those keywords.

Generated Variables For private Clause Listing. 12 shows the parallelized for-loop by the OMP-CodeT5+ that recognizes the variables to privatize. In this listing, OMP-CodeT5+ recognized the variables "`i`", "`j`", "`k`", and "`m`" in the

Listing 14: Passed Parallel For-loops Utilize shared Clause by The OMP-CodeT5+ Model in the ssor Function LU Benchmark

```
#pragma omp parallel for private(k,j,i,m) shared(u,rsd)
for(k=1; k<nz-1; k++){
    for(j=jst; j<jend; j++){
        for(i=ist; i<iend; i++){
            for(m=0; m<5; m++){
                u[k][j][i][m]=u[k][j][i][m]+tmp*rsd[k][j][i][m];
            }
        }
    }
}
```

nested for-loops to prevent other threads access these variables and create the race condition between threads. For-loop that shows in Listing 12 is one of the parallel for-loops that utilizes the "`private`" clause perfectly with the four nested for-loops. Compared to the Listing 13, the for-loop that shows in this listing has three nested for-loops but the OMP-CodeT5+ model miss recognizes the variable from one nested for-loops, "`i3`". This parallel for-loops by the OMP-CodeT5+ didn't show the error sign during the evaluation run. However, an error is still present in the code. Listing 4 shows the normal for-loop, not nested, without OpenMP "`private`" clause. The code in this Listing also needs to privatize the variable "j" to prevent the race condition among the multi threads. Also, OMP-CodeT5+ didn't recognize variable "j" that needs to be privatized.

In conclusion for the generated "`private`" clause, the OpenMP directives with the "`private`" clause that was correctly generated by the OMP-CodeT5+ model must have more than or equal to four nested for-loops. The input nested for-loop that has less than four nested for-loops, OMP-CodeT5+ can generate some of the variables for the "`private`" clause.

Generated Variables For shared Clause Listing 14 shows the passed parallelized for-loop by the OMP-CodeT5+ model that recognizes the variable correctly and utilizes the "`shared`" clause. The purpose of this nested for-loop is to update the item in the array "u" by summing the previous value of items in array "u" with the multiplication of variable "tmp" and item in array "rsd". Compared to the Listing 15, the parallelized for-loop by the OMP-CodeT5+ failed to run this

Listing 15: Failed Parallel For-loops Utilize shared Clause by The OMP-CodeT5+ Model in the l2norm Function LU Benchmark

```
#pragma omp parallel for private(k,j,i,m) shared(v,sum)
for(k=1; k<nz0-1; k++){
    for(j=jst; j<jend; j++){
        for(i=ist; i<iend; i++){
            for(m=0; m<5; m++){
                sum[m]=sum[m]+v[k][j][i][m]*v[k][j][i][m];
            }
        }
    }
}
```

for-loop correctly. The purpose of this nested for-loop is to update the item in the array "sum" by summing the previous value of items in array "sum" with the power of two of values from items in array "v". The generated clauses still do not iron out the possibility of the race condition when the variables "k", "j", and "i" are different but variable "m" is the same in the parallel region across the threads, the retrieved the same value from array "sum" across threads and update the incorrect value.

In conclusion for the generated "`shared`" clause, the OpenMP directives with the "`shared`" can generate the correct variables for the "`shared`" clause in some cases that the OMP-CodeT5+ models have learned previously with the training data. Another reason for the incorrectly generated variables for "`shared`" is the variable's names. The self-attention mechanism function that calculates the relationship weight among the sentence, the same variable name that is used with one clause in one source code can appear with other clauses in other source codes. The intention of the variables can be different in each scenario but it can share the same variable names. The training data may need to normalize the variable names to indicate the usage purpose in the pre-processes data phase, for instance, the variable to use with "`private`" should change to "p1", "p2", "p3" and so on, to isolate the variable use with different clauses when training the model.

Generated Variables For Reduction Clause Listing 16 shows the passed parallelized for-loop by the OMP-CodeT5+ model that recognizes the variable correctly and utilizes the "`reduction`" clause. The purpose of this for-loop is to

Listing 16: Passed Parallel For-loops Utilize reduction Clause by The OMP-CodeT5+ Model in the conj_grad Function CG Benchmark

```
#pragma omp parallel for reduction(+:rho)
for(j = 0; j < lastcol - firstcol + 1; j++){
    rho = rho + r[j]*r[j];
}
```

calculate the summation from all the power of two values of each item in array "r". The OMP-CodeT5+ model can generate the "`reduction`" clause and identify the variable "rho" as a reduction-identifier and the reduction-modifier, "+", from the input source code. Compared to the Listing 4, the OMP-CodeT5+ cannot generated the "`reduction`" clause with variables, even though, the input source code has a similar pattern. In conclusion for the generated "`reduction`" clause, the training data for the "`reduction`" clause is not enough for the training data to recognize the pattern for utilizing this clause as shown in Fig. 19.

4.5.4 Failures of OMP-CodeT5+ in OpenMP Directive Generation

Failure in Parallelizing EP Benchmark The OMP-CodeT5+ failed to parallelize the EP benchmark completely and could not produce any speedup result because the EP benchmark has only one parallel for-loop as shown in Table 3. The reason generated OpenMP directives and clauses failed to parallelize EP benchmark source code even if it has similar directives and clauses is the OMP-CodeT5+ cannot identify which variables need to be privatized and which variables need to be shared. In the original EP benchmark source code shown in Listing 18 privatize the variable by declaring the variable after the thread has been forked, by declaring after forking the thread the variable will be isolated from other threads and can be used in this thread only, preventing it from getting read and write from other threads. However, the OMP-CodeT5+ scope does not include modifying the code to suit the parallel environment and only focuses on generating OpenMP directives and clauses from raw for-loops source code as shown in Listing 17. Many variables are not privatized and allow other threads to read and write the variable address, resulting in EP benchmark source code parallelized by OMP-CodeT5+ failing the benchmark verification steps.

However, privatizing the variables using the original OpenMP EP source code

Listing 17: Parallelized For-loops by The OMP-CodeT5+ Model from EP Benchmark

```
#pragma omp parallel for reduction(+:sx,sy)
for(k=1; k<=np; k++){
    kk = k_offset + k;
    t1 = S;
    t2 = an;
    for(i=1; i<=100; i++){
        ik = kk / 2;
        if((2*ik)!=kk){t3=randlc(&t1,t2);}
        if(ik==0){break;}
        t3=randlc(&t2,t2);
    kk=ik;
    }
    vranlc(2*NK, &t1, A, x);
    for(i=0; i<NK; i++){
        x1 = 2.0 * x[2*i] - 1.0;
        x2 = 2.0 * x[2*i+1] - 1.0;
        t1 = pow2(x1) + pow2(x2);
        if(t1 <= 1.0){
            t2 = sqrt(-2.0 * log(t1) / t1);
            t3 = (x1 * t2);
            t4 = (x2 * t2);
            l = max(fabs(t3), fabs(t4));
            q[l] += 1.0;
            sx = sx + t3;
            sy = sy + t4;
        }
    }
}
```

Listing 18: Parallel For-loops from Original OpenMP EP Benchmark

```
#pragma omp parallel
{
    double t1, t2, t3, t4, x1, x2;
    int kk, i, ik, l;
    double qq[NQ];
    double x[NK_PLUS];

    #pragma omp for reduction(+:sx,sy)
    for(k=1; k<=np; k++){
        ...
        #pragma omp critical
        {
            for( i = 0; i <= NQ - 1; i++ ) q[i] += qq[i];
        }
    }
}
```

approach is more convenient and it ensures that the variables are isolated from other threads. If the OMP-CodeT5+ user modifies the generated OpenMP directives and clauses by adding the private and shared clauses, the EP benchmark can run in multi-thread and pass the verification step. The modify OpenMP directive is "`#pragma omp parallel for reduction(+:sx,sy,q) private(ik,kk,i,l ,t1,t2,t3,t4,x1,x2,x,timers_enabled) shared( an, np, k_offset )`"

Failure to Recognize Pointers in OpenMP Directive Generation Listing 19 shows the syntax error of the generated directive by the trained model. The cause of the error is the reduction clause cannot be used with the pointer. In this case, the trained model failed to recognize the "`rnmu`" as a pointer and used this pointer with the reduction clause. Due to the small training data that led to the trained model not recognizing yet that the reduction clause cannot be used with the pointer. Efforts were made to increase the training data, but the availability of high-quality OpenMP source code is very limited. Consequently, it was not possible to create a comprehensive training dataset that would enable CodeT5+ to fully learn the correct and incorrect OpenMP syntax.

High Failed Rated of OpenMP Directives Without Additional Clauses Fig. 21 shows the high percentage of the failed parallelized for-loop with the direc-

Listing 19: The Generated Directive Has Syntax Error, For-loop in norm2u3 function From MG Benchmark

```
...
*rnmu = 0.0;
#pragma omp parallel for default(shared) private(i1,i2,i3,a) reduction(+:s)
    reduction(max:rnmu)
for(i3 = 1; i3 < n3-1; i3++){
    for(i2 = 1; i2 < n2-1; i2++){
        for(i1 = 1; i1 < n1-1; i1++){
            s = s + r[i3][i2][i1] * r[i3][i2][i1];
            a = fabs(r[i3][i2][i1]);
            if(a > *rnmu){*rnmu = a;}
        }
    }
}
...
```

tives that do not have additional any OpenMP clauses, "`#pragma omp parallel for`". The reason for this high percentage is the training data don't have the full context of the for-loops. There are two identified root causes. First, the label data in the dataset didn't include the clauses that were declared outside the "`#pragma omp parallel for`" and "`#pragma omp for`" scope, for instance, Listing 5 shows the CG benchmark source code, the "`private(it,i,j,k)`" clause that declared with "`#pragma omp parallel`" directives will not include with the label data. The second cause, the variables that are declared inside the parallel region are automatically privatizing, for instance, Listing 7 shows the "`x_solve`" function call within the parallel region and the variables that declared in the "`x_solve`" function will be automatically privatized for each thread, the pre-processing process does not include these data that declared with this style into the label data. The OMP-CodeT5+ is training with some data that don't have full context because of these two root causes.

To increase the percentage of the passed parallelized for-loop by the OMP-CodeT5+ model, the pre-preprocessing processes need to be modified. The pre-processing phases need to include the whole context of the for-loops, the trace of the variables used in the for-loop scope. First, the labeling of the data process needs to include the clauses that are declared outside the current for-loop scopes. Second, the training data needs to utilize the dataflow of each variable to identify

Listing 20: Parallel For-loops by The OMP-CodeT5+ Model in the main Function from CG Benchmark

```
#pragma omp parallel for default(shared) private(j) reduction(+:norm_temp1,
    norm_temp2)
for(j = 0; j < lastcol - firstcol + 1; j++){
    norm_temp1 = norm_temp1 + x[j] * z[j];
    norm_temp2 = norm_temp2 + z[j] * z[j];
}
```

the procedure to manage the variable and include it in the training data.

4.5.5 Nested For-loops and incorporating Contextual Information in Training Data

The current capability of the OMP-CodeT5+ is focused on handling individual for-loop or whole nested for-loops. In future work on the parallelizable for-loop classification model, the training dataset will include the nesting level of loops as one of the contextual features to assist in determining whether the nested for-loops are parallelizable or not. The output of the model will be streamlined at the end to select only the outermost for-loop in the nested loops for parallelization.

Furthermore, incorporating the additional contextual information about the variable and source code can increase the accuracy of the model in identifying variables and matching them with suitable OpenMP clauses. The contextual information of the variables can be helpful in revealing the data type of each variable, which plays an important role in this process. As shown in Fig. 19, if the training dataset includes information identifying the data types of variables, such as "`rnmu`" as a pointer, the model can learn that pointer variables cannot be used with the reduction clauses.

4.5.6 Evaluation of the Practicality of OMP-CodeT5+ in Parallel Programming

Listing 21 and Listing 20 show the parallelized for-loop by the OMP-CodeT5+ from FT and CG benchmark that have private clause and reduction clause. The OMP-CodeT5+ model will be a good companion to the new parallel developers by following the OMP-CodeT5+ generated OpenMP directives as a guide. As

Listing 21: Parallel For-loops by The OMP-CodeT5+ Model in the compute_indexmap Function from FT Benchmark

```c
#pragma omp parallel for default(shared) private(i,j,k,kk,kk2,jj,kj2,ii)
for(k=0; k<d3; k++){
    kk = ((k+NZ/2) % NZ) - NZ/2;
    kk2 = kk*kk;
    for(j=0; j<d2; j++){
        jj = ((j+NY/2) % NY) - NY/2;
        kj2 = jj*jj+kk2;
        for(i=0; i<d1; i++){
            ii = ((i+NX/2) % NX) - NX/2;
            twiddle[k][j][i] = exp(ap*(double)(ii*ii+kj2));
        }
    }
}
```

shown in Fig. 17, the high accuracy of the passed parallelized source code by OMP-CodeT5+ for the small and medium complexity source code will help developers reduce the manual effort required to write boilerplate parallel constructs, allowing programmers to focus on higher-level logic. The OMP-CodeT5+ can consistently apply parallelization patterns and best practices, ensuring uniformity across the codebase.

The advantage of fine-tuning the CodeT5+ model based on the transformer models, these models excel at recognizing patterns in the datasets. By training on diverse for-loop patterns of well-parallelized code, the OMP-CodeT5+ can learn and replicate effective parallelization strategies, potentially surpassing less experienced programmers in some aspects. The model can capture and utilize the context of the code. This contextual understanding enables the generation of code that is not only syntactically correct but also semantically meaningful.

At the current state of the OMP-CodeT5+ to parallelized source code, the developers can reduce the developing time by parallelizing the common source code and focusing on the specific function of the programs. The OMP-CodeT5+ shows the capability to recognize the common parallel source code patterns that had been trained with the training data and generate the correct parallelized source code that can increase the performance of the program as shown in Fig. 15.

4.6 Conclusion

In conclusion, the OMP-CodeT5+ model shows a promising capability to parallelize C/C++ source codes and can be positioned as a viable alternative to existing automatic parallelization tools based on static analysis. The model effectively generates OpenMP directives for source codes with low to intermediate complexity. However, for highly complex source codes, some generated directives contain syntax errors and do not enhance performance. OMP-CodeT5+ successfully generates compilable OpenMP directives for 73% of the parallel for-loops in the NPB benchmarks, surpassing AutoPar-Clava's success rate of 58%. The higher success rate of OMP-CodeT5+ indicates its ability to parallelize a broader range of loop patterns. In certain cases, it achieves speedups comparable to those obtained by human developers, showing its effectiveness in parallel code generation.

Future work will focus on integrating additional features of source codes, including dataflow graphs, to enhance the model's ability to accurately parallelize source code and manage a wider range of parallel patterns and algorithms. Dataflow analysis is considered useful for identifying variables declared outside the loops that require privatization. Additionally, the current dataset is constrained by its size and the limited variety of parallel patterns it covers. To address this limitation, the training dataset will be expanded by collecting OpenMP source codes from a wider range of public repositories beyond GitHub.

5 Automatic Parallelization Tool based on LLM

This chapter will show the expected outcomes of the automatic parallelization with OpenMP using LLM as shown in Fig. 3. The overall workflow incorporates two models: the parallelizable for-loop classification model detailed in Chapter 3, and the OpenMP directive generation model, OMP-CodeT5+, described in Chapter 4. Additionally, the advantages of employing this LLM-based automatic parallelization tool for parallel program development will be examined, along with an overview of the target users of this study.

5.1 The Expected Results of Automatic Parallelization Tool based on LLM

Table 5 shows the number of successfully parallelized for-loops by the proposed LLM-based automatic parallelization tool on NPB. The benchmarks are categorized based on their speedup results from Chapter 4 into three groups: high speedup, medium speedup, and no speedup. The table is divided into three main sections. The first section lists the number of parallel for-loops in the original NPB. The second section details the classification results from the parallelizable for-loop classification model, indicating the number of for-loops identified as parallelizable versus those deemed not parallelizable. The third section reports the outcomes from the OMP-CodeT5+ model, showing the number of successful (passed) and unsuccessful (failed) OpenMP directive generations for the loops identified as parallelizable in the second section.

The proposed LLM-based automatic parallelization tool achieved parallelization for 53% of the for-loops that are parallelized in the original NPB. Upon closer examination, the parallelizable for-loop classification model identified 75% of the for-loops that were parallelized in the original NPB as parallelizable. Subsequently, the OMP-CodeT5+ model successfully generated correct OpenMP directives for 71% of these identified parallelizable for-loops. Consequently, the overall success rate of the proposed parallelization method stands at 53%.

Figure 23 provides a visual representation of the results summarized in Table 5. The figure distinguishes between the for-loops in the original NPB that were identified as parallelizable and successfully generated with correct OpenMP

Table 5: Detailed Results of Parallelization in Parallel NPB For-loops: Identification and Generation Success Rates

Name	**# of Parallel For-loops**	**Parallelizable**		**Generation**	
		Yes	**No**	**Passed**	**Failed**
MG	11	10	1	10	0
CG	9	9	0	9	0
FT	7	7	0	3	4
SP	19	11	8	9	2
LU	13	8	5	3	5
IS	3	2	1	0	2
BT	15	11	4	8	3
EP	1	1	0	0	1
Total	78	59	19	42	17

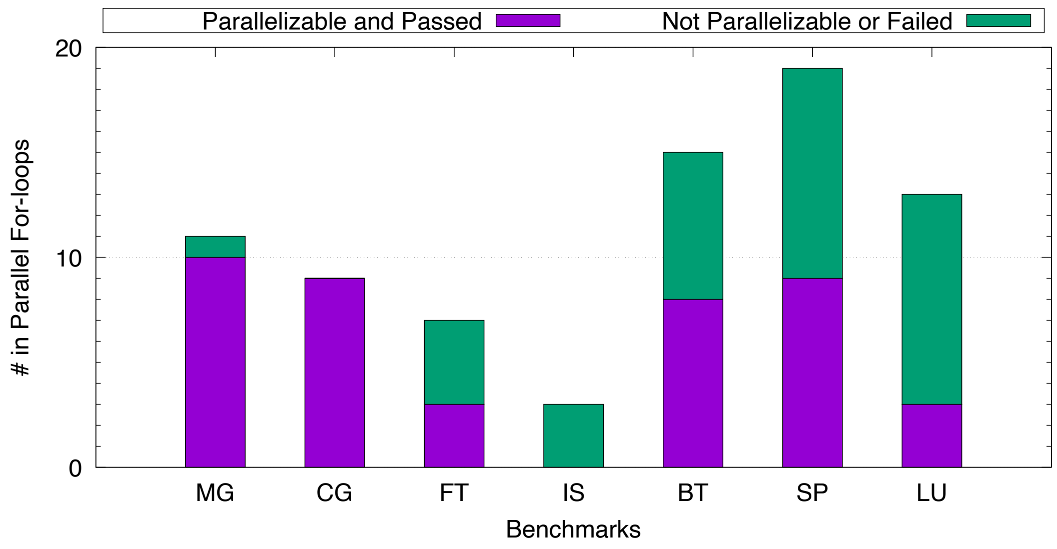


Figure 23: Parallelization Success and Failure Rates in Parallel NPB For-loops by Parallelizable For-loops Classification Model and OMP-CodeT5+ Model

directives by the OMP-CodeT5+ model (indicated in purple), and those that were either not identified as parallelizable or for which the OMP-CodeT5+ model failed to generate correct OpenMP directives (indicated in green).

High Speedup The results of the proposed automatic parallelization tool applied to the MG and CG source codes illustrate its effectiveness in identifying and parallelizing for-loops. Specifically, the tool identified a significant number of parallelizable for-loops and successfully generated the appropriate OpenMP directives for both MG and CG benchmarks. This capability is further substantiated by the observed speedup, which exceeds twice the performance of the original single-threaded implementations for these benchmarks.

Medium Speedup The speedup results for the parallelized FT source code using the proposed automatic parallelization tool remain unchanged after incorporating the classification results from the parallelizable for-loop classification model. This is because the classification model successfully identifies all parallel for-loops in the FT source code. Consequently, the parallelized FT source code exhibits the same performance improvements as depicted in Figure 15.

In contrast, when the parallelizable for-loop classification model and the OMP-CodeT5+ model are applied to the SP source code, the number of successfully parallelized for-loops is reduced by seven. This reduction in parallelized loops is estimated to decrease the speedup by approximately 43%.

Similarly, for the LU source code, after integrating the parallelizable for-loop classification model and the OMP-CodeT5+ model, only three for-loops are successfully parallelized. This significant reduction in parallelized loops is projected to decrease the speedup to less than 1.2×, potentially resulting in no significant performance improvement.

No Speedup The proposed automatic parallelization tool was unsuccessful in parallelizing the IS source code. Despite the parallelizable for-loop classification model identifying two out of three for-loops as parallelizable, the OMP-CodeT5+ model failed to generate valid OpenMP directives for these loops. As a result, the parallelized IS source code did not achieve any speedup.

Similarly, the BT source code did not exhibit significant speedup when parallelized using the proposed tool. As illustrated in Figure 15 and Table 3, the parallelization effort did not yield notable performance improvements. Even after incorporating the parallelizable for-loop classification model, which further reduced the number of parallelized for-loops, the speedup results remained unchanged.

5.2 Advantages of using the Proposed LLM-based Automatic Parallelization Tool

The proposed automatic parallelization tool offers advantages by allowing developers to concentrate on high-complexity for-loops while delegating the parallelization of lower-complexity for-loops to the tool. This approach leverages the tool's capabilities to efficiently manage small to medium complexity loops, achieving performance improvements comparable to manually parallelized code. This effectiveness is evidenced in Fig. 15 and Fig. 23, where the tool's performance closely matches that of human-optimized parallelized code, such as the parallelized MG source code.

The distribution of for-loop sizes, depicted in Fig. 18, highlights that small and medium complexity for-loops constitute 90% of all for-loops in public GitHub repositories. Consequently, the automatic parallelization tool can substantially reduce the time developers spend on parallelization tasks. Furthermore, the tool demonstrates a high accuracy rate, correctly parallelizing over 80% of small and medium complexity for-loops, as shown in Fig. 17.

5.3 Target Users

The proposed LLM-based automatic parallelization tool is designed to cater to both intermediate or expert parallel programmers and beginners. For intermediate and expert users, the tool offers the advantage of automating the parallelization of small and medium complexity for-loops, thereby allowing these programmers to focus on more complex aspects of their code. This automation streamlines the development process, as these users can rely on the tool to handle less complex parallelization tasks efficiently.

For beginner parallel programmers, the tool serves an educational function by guiding them in identifying parallelizable for-loops and generating the appropriate OpenMP directives. By learning from the tool's parallelization of small and medium complexity for-loops, beginners can gain a deeper understanding of the principles of parallel programming. The tool incorporates both the parallelizable for-loop classification model and the OMP-CodeT5+ model, providing a comprehensive framework for learning and applying parallelization techniques.

6 Conclusion and Future Work

This dissertation leveraged Code LLMs to replicate and enhance the capabilities of conventional static analysis-based automatic parallelization tools. I analyze that an automatic parallelization tool fundamentally consists of the following two tasks: (1) identifying independent loops in the input source code and (2) generating functionally identical parallel versions of independent loops. In this dissertation, I proposed and evaluated an LLM-based model to perform each task.

For the first task, a parallelizable for-loop classification model based on the CodeT5 model is proposed. This model takes advantage of the code understanding capabilities of Code LLMs and classifies whether a given code snippet containing a for-loop is parallelizable or not. For the second task, a generative model named OMP-CodeT5+ based on the CodeT5+ model is proposed. This model generates an appropriate OpenMP directive including clauses such as privatization clauses for a given parallelizable for-loop.

Both models were fine-tuned with CodeT5/CodeT5+ models based on the transformer model using C/C++ OpenMP source code from public GitHub repositories. To ensure the training data quality, the gathered C/C++ OpenMP source code was pre-processed, including filtering repositories with stars, dropping duplicated data, and filtering out low-quality for-loops.

Both models were evaluated with the NAS parallel benchmark suite, which contains eight benchmarks. The baseline model compared with the parallelizable for-loops classification model was a state-of-the-art automatic parallelization using static analysis techniques named AutoPar-Clava. For OMP-CodeT5+, the baselines were handed parallelized OpenMP source code and AutoPar-Clava.

The parallelizable for-loops classification model was evaluated by the F1 score of correctly identified parallel for-loops. The parallelizable for-loops classification model achieved an F1 score of 0.764 in the NPB source code, whereas AutoPar-Clava achieved an F1 score of 0.669. The parallelized source code generated by OMP-CodeT5+ was classified into three categories by their speedup: high speedup, medium speedup, and no speedup, and the number of parallel loops each approach can be parallelized. The results speedup of parallelized NPB source code using OMP-CodeT5+ were as follows. Three of the eight benchmarks achieved high speedup, two achieved medium speedup, and three benchmarks did not gain any speedup. OMP-CodeT5+ successfully generated compilable OpenMP directives for 73% of the parallel for-loops in the NPB benchmarks, whereas the success rate was only 58% for AutoPar-Clava.

The parallelizable for-loops classification model could identify parallel for-loops up to a certain level of complexity. The main reason was the amount of training data for classifying each parallel pattern present in the dataset. At the current state, this model can only identify parallel patterns of low to medium complexity because there are many low to medium-complexity source codes scattered across public GitHub repositories. However, the amount of highly complex parallel source code is rare and thus limits the model performance in identifying the highly complex parallel source code. Another reason that the model mispredicts is because of the scope of the pre-processing step. The scope of each variable is important in determining the potential of the parallel for-loops but the code snippets in the training data often do not include the variable definitions that appear outside the for-loops. This is another point where the parallelizable for-loops classification model performance could be increased.

In conclusion, both the parallelizable for-loop classification model and OMP-CodeT5+ demonstrate their potentials to replace the existing approach of automatic parallelization tools utilizing static analysis techniques. The results from both models show signs that the Code LLMs could replace static analysis techniques and have a large room for improvement to increase the model performance in identifying the parallelizable for-loops and generating the OpenMP directives. At the moment, the main problem is the lack of the highly complex parallel code to increase the dataset. Because of this reason, the performance of automatic parallelization tools based on LLMs is still limited in identifying and generating highly complex source codes.

In the future, the pre-processed training data needs to incorporate more contextual information of the input source code to increase the model capability for identifying and generating the parallel source code. Additionally, the training dataset needs to be significantly enlarged to handle highly complex source codes.

Acknowledgments

I wish to thank the following people for their guidance, support, and assistance. Without these people, this work would not have been possible.

Firstly, I would like to express my appreciation to my research supervisor, Professor Hajimu Iida. He accepted me at his Software Design and Analysis Laboratory (SDLab) during one of the hardest crises in the world for humanity, the COVID-19 pandemic. Without his support and advice, this work would not have been possible.

Secondly, I would like to thank Professor Kazutoshi Fujikawa for taking the time to review, discuss, and provide feedback on my work. His feedback helped steer this work in the right direction.

I wish to thank my advisors, Associate Professor Kohei Ichikawa, Assistant Professor Keichi Takahashi, and Assistant Professor Yutaro Kachiwa for their continued support and guidance in my research work as well as my life in Japan, Their valuable suggestions and comments brought this research to fruition. Without them, I would not have successfully accomplished the doctoral course.

I want to express my gratitude to late Assistant Professor Putchong Uthayopas at the High Performance Computing and Networking Center for his guidance from my bachelor year until the last day of his life. He was my bachelor's senior project and master's thesis advisor. His knowledge and advice both in the classroom and outside the classroom encouraged me to become a wise and good person. He introduced me to the world of high-performance computing, offered me many opportunities to learn various technologies, and nurtured me for multiple years. Lastly, once he told me one of his personal goals was that "Right now, I hold the torch I received from my supervisor and will pass this torch to the next generation". I also will pass on his knowledge and advice to the next generations.

I want to express my thanks to my master thesis co-advisor Professor Chantana Chantraporнchai who gave me many advices, opportunities to express my creation in the High Performance Computing and Networking Center, and sharpened my skill. She helped me secure some financial support during my master's course and during the Covid-19 pandemic before I started my PhD course.

I want to express my thanks to the SDLab secretary Akiko Ogawa who helped me process my scholarship documents and gave me advice on how to handle the

Japanese documents.

I also want to thank the Nara Institute of Science and Technology (NAIST) for giving me the opportunity to explore my ability in the PhD course with the scholarship for all three years of my study. This includes all personnel from the International Student Affairs, Personal Section, and Career Service Office who helped me start living in Japan and solve many problems to live comfortably in Japan during my three years of study. I wish to express my appreciation to my Japanese teachers Yukino Iwade and Masako Hashimoto who taught me the Japanese language and allowed me to communicate with Japanese people in the Japan society. Also, I would like to thank all of the cafeteria staff who gave me the energy to finish my work.

To Friendship, I also express my thanks to all members during my time in the SDLab, especially my tutor, Daisuke Fukumoto, who actively provided assistance in many situations during his time as my tutor, as well as Akihito Ihara, Kyoya Murakami, Shuhei Kayawari, Xingyuan Kang, Junya Hishikawa, Takaha Mino, Shan Gao, Zheyuan Wei, Yasuhito Morikawa, Miki Yonekura, Wataru Mabuchi, Papon Choonhaklai, and M1 students in 2024.

Last but not least, I wish to express my highest gratitude to my parents raising and educating me with great care, and instilling the values of education and ethics since my youth. No amount of words would sufficiently express my gratitude. I thank both my parents for everything.

Reference

References

[1] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. M. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurr. Comput. Pract. Exp.*, vol. 22, no. 6, pp. 685–701, 2010.

[2] A. Danner, T. Newhall, and K. C. Webb, "Paravis: A library for visualizing and debugging parallel applications," in *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops*, pp. 326–333, 2019.

[3] X. Xie, H. Jiang, H. Jin, W. Cao, P. Yuan, and L. T. Yang, "Metis: a profiling toolkit based on the virtualization of hardware performance counters," *Hum. centric Comput. Inf. Sci.*, vol. 2, p. 8, 2012.

[4] M. I. Malkawi, "The art of software systems development: Reliability, availability, maintainability, performance (RAMP)," *Hum. centric Comput. Inf. Sci.*, vol. 3, p. 22, 2013.

[5] J. Szuppe, "Boost.compute: A parallel computing library for C++ based on opencl," in *Proceedings of the 4th International Workshop on OpenCL*, pp. 15:1–15:39, 2016.

[6] L. Tierney, A. J. Rossini, and N. Li, "Snow : A parallel computing framework for the R system," *Int. J. Parallel Program.*, vol. 37, no. 1, pp. 78–90, 2009.

[7] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, p. 46–55, jan 1998.

[8] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.1*, Nov. 2023.

[9] H. Arabnejad, J. Bispo, J. G. Barbosa, and J. M. P. Cardoso, "Autopar-clava: An automatic parallelization source-to-source tool for C code applications,"

in *Proceedings of the 9th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and 7th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms*, pp. 13–19, 2018.

[10] "Automatic Parallelization — intel.com." `https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2023-1/automatic-parallelization.html`. [Accessed 07-Jun-2023].

[11] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su, "Intel® openmp c++/fortran compiler for hyper-threading technology: Implementation and performance.," *Intel Technology Journal*, vol. 6, no. 1, 2002.

[12] Q. Zhang, "The accuracy of the non-continuous I test for one- dimensional arrays with references created by induction variables," *J. Inf. Process. Syst.*, vol. 10, no. 4, pp. 523–542, 2014.

[13] J. Brandt and K. Schneider, "Static data-flow analysis of synchronous programs," in *7th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2009), July 13-15, 2009, Cambridge, Massachusetts, USA*, pp. 161–170, IEEE, 2009.

[14] P. Petersen and D. A. Padua, "Static and dynamic evaluation of data dependence analysis techniques," *IEEE Trans. Parallel Distributed Syst.*, vol. 7, no. 11, pp. 1121–1132, 1996.

[15] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998* (J. W. Davidson, K. D. Cooper, and A. M. Berman, eds.), pp. 106–117, ACM, 1998.

[16] B. Burgstaller, B. Scholz, and J. Blieberger, "A symbolic analysis framework for static analysis of imperative programming languages," *J. Syst. Softw.*, vol. 85, no. 6, pp. 1418–1439, 2012.

[17] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *CoRR*, vol. abs/1808.03314, 2018.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pp. 5998–6008, 2017.

[19] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4171–4186, 2019.

[20] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020.

[21] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.

[22] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proceedings of the 37th International Conference on Machine Learning*, vol. 119 of *Proceedings of Machine Learning Research*, pp. 5110–5121, 2020.

[23] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and

generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021.

[24] H. Bae, D. Mustafa, J. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff, "The cetus source-to-source compiler infrastructure: Overview and evaluation," *Int. J. Parallel Program.*, vol. 41, no. 6, pp. 753–767, 2013.

[25] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. P. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 11, pp. 36–42, 2009.

[26] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011, p. 1, 2011.

[27] C. Liao, D. J. Quinlan, J. Willcock, and T. Panas, "Extending automatic parallelization to optimize high-level abstractions for multicore," in *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, vol. 5568, pp. 28–41, 2009.

[28] G. S. D. Mendonca, B. C. F. Guimarães, P. Alves, M. M. Pereira, G. Araujo, and F. M. Q. Pereira, "Dawncc: Automatic annotation for data parallelism and offloading," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 13:1–13:25, 2017.

[29] J. Bispo and J. M. P. Cardoso, "Clava: C/C++ source-to-source compilation using LARA," *SoftwareX*, vol. 12, p. 100565, 2020.

[30] Y. Wang, H. Le, A. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023* (H. Bouamor, J. Pino, and K. Bali, eds.), pp. 1069–1088, Association for Computational Linguistics, 2023.

[31] L. Chen, A. Bhattacharjee, N. K. Ahmed, N. Hasabnis, G. Oren, V. A. Vo, and A. Jannesari, "OMPGPT: A generative pre-trained transformer model for openmp," *CoRR*, vol. abs/2401.16445, 2024.

[32] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), 2022.

[33] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, “Hpc-coder: Modeling parallel programs using large language models,” in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pp. 1–12, 2024.

[34] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, “An empirical study of code smells in transformer-based code generation techniques,” in *22nd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Limassol, Cyprus, October 3, 2022*, pp. 71–82, IEEE, 2022.

[35] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader-Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*, pp. 336–347, 2021.

[36] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. Weeratunga, “The nas parallel benchmarks,” *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, pp. 63–73, 1991.

[37] J. Löff, D. Griebler, G. Mencagli, G. A. de Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, “The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures,” *Future Gener. Comput. Syst.*, vol. 125, pp. 743–757, 2021.

[38] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, “Zero-infinity: breaking the GPU memory wall for extreme scale deep learning,” in *International Conference for High Performance Computing, Networking, Storage*

and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021 (B. R. de Supinski, M. W. Hall, and T. Gamblin, eds.), p. 59, ACM, 2021.

[39] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (I. Calciu and G. Kuenning, eds.), pp. 551–564, USENIX Association, 2021.

[40] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: memory optimizations toward training trillion parameter models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020* (C. Cuicchi, I. Qualters, and W. T. Kramer, eds.), p. 20, IEEE/ACM, 2020.

[41] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020* (R. Gupta, Y. Liu, J. Tang, and B. A. Prakash, eds.), pp. 3505–3506, ACM, 2020.

[42] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, (Barcelona, Spain), pp. 74–81, Association for Computational Linguistics, July 2004.

List of publications

Journal Article

1. Soratouch Pornmaneerattanatri, Keichi Takahashi, Yutaro Kashiwa, Kohei Ichikawa, Hajimu Iida, Toward Automatic Parallelization: Detecting Parallelizable Loop using Large Language Modeling, *Journal of Information Processing Systems*, Status: Accepted and in print.

Conference Paper

1. Soratouch Pornmaneerattanatri, Keichi Takahashi, Yutaro Kashiwa, Kohei Ichikawa, Hajimu Iida, Automatic Parallelization with CodeT5+: A Model for Generating OpenMP Directives, *2024 International Workshop on Large Language Models (LLMs) and HPC*, status: Accecpted and in print.

2. Soratouch Pornmaneerattanatri, Keichi Takahashi, Yutaro Kashiwa, Kohei Ichikawa, Hajimu Iida, Parallelizable Loop Detection using Pre-trained Transformer Models for Code Understanding, *International Conference on Parallel and Distributed Computing: Applications and Technologies*, pages 32-42, August 16-18, 2023.

3. Nabhan Suwanachote, Soratouch Pornmaneerattanatri, Yutaro Kashiwa, Kohei Ichikawa, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, Hajimu Iida, A Pilot Study of Testing Infrastructure as Code for Cloud Systems, *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 584-588, December 4-7, 2023.

4. Junya Hishikawa, Daisuke Fukumoto, Soratouch Pornmaneerattanatri, Yutaro Kashiwa, Toshiki Hirao, Kenji Fujiwara, Hajimu Iida, Toward Analyzing OSS Developers Contributing to the Removal of Technical Debt in Multiple Projects, *Technical Committee on Knowledge-Based Software Engineering (KBSE)*, pages 52-57, January 19-20, 2023.

Poster Presentation

1. Soratouch Pornmaneerattanatri, Keichi Takahashi, Yutaro Kashiwa, Kohei Ichikawa, Hajimu Iida, A Proposal of Automatic Parallelization using Transformer-based Large Language Models, *The International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2024)*, poster presentation, January 25-27, 2024.