J Inf Process Syst, Vol.21, No.3, pp.271~283, June 2025
https://doi.org/10.3745/JIPS.04.0349
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Toward Automatic Parallelization: Detecting Parallelizable Loops Using Large Language Modeling

Soratouch Pornmaneerattanatri[1,*], Keichi Takahashi[2], Yutaro Kashiwa[1], Kohei Ichikawa[1], and Hajimu Iida[1]

Abstract
To fully harness the capabilities of multi-core processors, parallel programming is indispensable, demanding a comprehensive understanding of both software and hardware. Although various tools have been developed to automate parallel programming by leveraging static analysis approaches, manually parallelized code continues to consistently outperform those that are automatically parallelized. Meanwhile, the emergence of transformer-based large language models has facilitated significant breakthroughs in understanding and generating programming languages. This study presents a model tailored to detecting parallelizable for-loops by fine-tuning the transformer-based model, CodeT5. The fine-tuned model assists programmers in identifying independent for-loops that have the potential for parallelization with libraries like OpenMP, leading to performance enhancements in software applications. The model was trained on 500,000 for-loops sourced from public GitHub repositories and demonstrated an F1-score of 0.860 in detecting parallelizable for-loops within a public GitHub dataset, along with an F1-score of 0.764 on the NAS Parallel Benchmark suite.

Keywords
Automatic Parallelization, Deep Learning, Large Language Model, OpenMP, Parallel Computing

1. Introduction

Parallel software development is supported by a large number of programming languages, libraries [1], and frameworks [2] that have been introduced over time. OpenMP [3] stands out as a significant language extension for C/C++ and Fortran, tailored for parallel software development. It offers developers straightforward compiler directives, enabling the explicit annotation of code sections for parallel execution on shared-memory architectures.

However, effective utilization of OpenMP requires an in-depth understanding of parallel programming and a comprehensive grasp of the underlying hardware to fully exploit its potential performance of the hardware. Furthermore, even experienced developers may commit errors in developing parallel applications due to the complexities inherent in both software and hardware [4-7].

In response to these challenges, automatic parallelization tools [8-11] have been introduced. These tools perform an analysis of the source code and automatically modify it into a parallelized version. However, while these tools have undergone consistent enhancements and show promising developments it has been frequently reported that they often generate code whose performance falls short of code that

Manuscript received October 12, 2023; accepted December 23, 2023.
* **Corresponding Author:** Soratouch Pornmaneerattanatri (pornmaneerattanatri.so.pn8@is.naist.jp)
[1] Nara Institute of Science and Technology, Nara, Japan (pornmaneerattanatri.so.pn8@is.naist.jp, yutaro.kashiwa@is.naist.jp, Ichikawa@is.naist.jp, iida@itc.naist.jp)
[2] Cyberscience Center, Tohoku University, Sendai, Japan (keichi@tohoku.ac.jp)

has been manually parallelized [8].

The limitations observed in current automatic parallelization tools primarily stem from their reliance on static analysis, which determines whether particular sections of code, such as loop iterations that can be safely executed in parallel. As static analyzers work only with variables accessible at compile time, they possess intrinsic constraint when compared to manual parallelization. Firstly, these tools are unable to parallelize loops that need runtime information or specific domain knowledge about the application. Secondly, even if a loop is technically parallelizable, the added overhead might compromise its efficiency relative to the non-parallelized version. Additionally, the time required for static analysis can increase significantly as the code's size and complexity rise.

Furthermore, recent advancements in natural language processing (NLP) studies highlight the potential for extracting knowledge from source code through deep learning-based methods [12-14]. This is achieved by training NLP models on a vast collection of source code accessible on open platforms like GitHub. Many studies have shown the effectiveness of NLP techniques across various tasks related to programming languages. Therefore, using deep learning-based NLP approaches, it is possible to extract parallelization patterns in public source code and develop tools to assist in parallel programming.

This research introduces a classification model designed to identify parallelizable for-loops through a fine-tuned transformer model, CodeT5. The model was pre-trained on CodeSearchNet [15] and supplemented with C and C# source code. By leveraging this classification model, programmers can convert serial programs into parallel programs more effectively.

The structure of this article is as follows. Section 2 reviews the relevant studies on automatic parallelization tools and deep learning-based NLP. Section 3 explains the data collection, labeling, and fine-tuning workflow of the CodeT5 model. Section 4 provides the evaluation setup and dataset. Section 5 exhibits the evaluation results. Section 6 discusses the strengths and weaknesses of using our proposed model, and Section 7 summarizes this article and explores directions for future research.

2. Related Work

2.1 Automatic Parallelization Tools

Most modern compilers, ranging from open-source options like the GNU C compiler (GCC) to proprietary compilers like the Intel C compiler (ICC), can perform automatic parallelization through static analysis. Specifically, ICC employs dataflow analysis [16]. Although compilers can significantly improve execution times through code parallelization, the parallelization of sophisticated loops continues to demand hand-operated and experienced direction.

Several source-to-source compilers incorporate automatic parallelization, leveraging a range of static analysis techniques. Notably, Cetus [9,17], Rose [10,18], DawnCC [11], and Clava [8,19] primarily employ dependence and range analyses. Dependence analysis [20] assesses the parallelizability of a for-loop by utilizing methods such as privatization, reduction, or alias analysis. Range analysis identifies parallelizable for-loops by evaluating the interaction of integer variables between the lower and upper bounds throughout the operation.

Source-to-source compilers diverge significantly in their parallelization methods. Cetus focuses solely on the C language and applies privatization, reduction, and alias analysis for dependence determination. Rose addresses dependence by solving linear integer equations related to loop induction variables accessing arrays, utilizing Gaussian elimination. DawnCC can convert C/C++ source code to parallelized code using OpenMP, OpenCL, or CUDA, primarily relying on symbolic range analysis, supplemented by traditional dependence analysis, to identify parallelizable loops. Clava incorporates induction variables, privatization, and scalar and array reductions in its dependence analysis. Scalar and array reductions are employed specifically when the initial two methodologies prove ineffective in identifying parallelizable loops. However, it is noteworthy that the execution performance of code automatically generated by such source-to-source compilers often lags behind meticulously crafted, hand-written source code [8].

2.2 Deep Learning-based Natural Language Processing Models

The rapid evolution of deep learning has ushered in novel possibilities across a multitude of fields, particularly those leveraging human-generated data from the Internet. Text represents a primary and prolific source of data that is not immediately available. One must first annotate the context within sentences. Furthermore, to optimize the performance of language models, extensive collections of labeled sentences are needed.

The development of the Transformer model [12] represented a transformative moment in NLP, establishing new benchmarks in machine translation. Before the advent of the Transformer model, recurrent neural networks (RNNs) were the prevailing choice for deep learning-based NLP. The key contribution of the Transformer model was its self-attention mechanism, which revolutionized the way linguistic relationships were processed. Subsequently, numerous studies have focused on improving the Transformer model using various approaches. A particularly noteworthy enhancement emerged with bidirectional encoder representations from transformers (BERT) [21]. BERT augmented the Transformer architecture by incorporating two additional training methodologies: predicting words within a sentence and predicting the following sentence. BERT's design is particularly valued for its adaptability, enabling fine-tuning for specific downstream tasks through its pre-trained model's output layer.

A notable advancement is the T5 (Text-To-Text Transfer Transformer) [22], derived from the archetypal Transformer model, characterized by its encoder-decoder architecture. In contrast to the encoder-only approach, BERT, T5 reframes NLP tasks into a text-to-text format, representing both input and output in text form. Furthermore, T5 introduces modifications to certain layers of the original Transformer model to align with its methodology. It has proven effective in numerous downstream tasks, including translation, summarization, and question answering.

Intriguingly, the software analysis field has also drawn inspiration from deep learning-based NLP methods. A prominent example is code-understanding BERT (CuBERT) [23], which leverages a BERT-based model pre-trained on Python source code. Recognizing the structural differences between programming and natural languages, the research team developed a Python-specific tokenizer to optimize BERT for Python tokens. CuBERT demonstrated superior performance across five downstream tasks compared to earlier approaches. Another significant model, CodeT5 [24], adheres to the T5 architecture, as its name suggests. Its unique proposition lies in the downstream tasks that target conversion between programming language (PL) and human language or natural language (NL). These tasks from CodeT5 include code explanation (PL to NL), code generation from descriptions (NL to PL), and code translation between different programming languages (PL to PL).

3. Methodology

3.1 Overview

This section details the methodology for developing a deep learning-based model for classifying parallelizable and non-parallelizable for-loops. Fig. 1 shows the flow of this study. Initially, C/C++ source files with OpenMP directives were gathered from publicly available GitHub repositories. Afterward, for-loops were extracted from these files and labeled based on the existence or non-existence of OpenMP directives. Lastly, these labeled for-loops were utilized to fine-tune a pre-trained CodeT5 model, empowering it to assess the parallelization potential of a given for-loop.

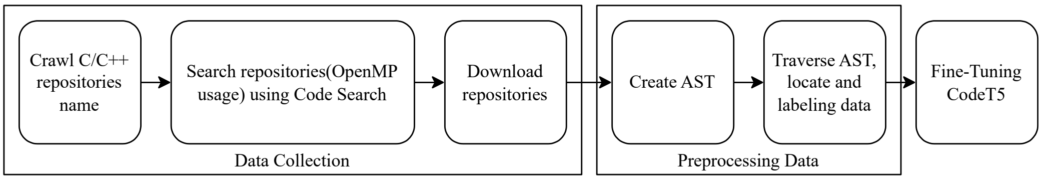

Crawl C/C++ repositories name
Search repositories(OpenMP usage) using Code Search
Download repositories
Data Collection
Create AST
Traverse AST, locate and labeling data
Preprocessing Data
Fine-Tuning CodeT5

Fig. 1. Model training workflow.

3.2 Data Collection

Currently, there is no publicly available dataset that is dedicated to building models capable of predicting parallelizable for-loops, such as a collection of labeled for-loops. We implemented an extensive data collection process using the GitHub code search API, focusing on files that utilize OpenMP directives. This section explains our data collection methodology in detail.

While the GitHub API code search function offers a wealth of data, it does not support searching for code 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 the direct extraction of source files containing OpenMP directives challenging. Therefore, we decided to collect data in the following steps: 1) collect C/C++ repositories; 2) identify repositories containing OpenMP directives; and 3) extract files containing OpenMP directives.

Our first step was to identify suitable C/C++ repositories. Using the GitHub API, we searched for all C/C++ repositories on GitHub, bypassing the output limit constraint. To ensure a certain level of code quality [25], we excluded repositories with 10 or fewer stars. After securing this subset of repositories, we performed another search to identify repositories showing traces of OpenMP usage by searching for the keyword "#pragma omp" using the GitHub API code search. This process resulted in the identification of 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, our next step was to extract the files relevant to our research. We searched for files containing the string "#pragma omp", 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, ensuring a level of quality in parallelization; 2) while technically we could gather non-parallelizable for-loops from files without the "#pragma omp" keyword, these files might have been written by developers not well-acquainted with OpenMP and may contain quality issues. For example, they might contain many loops that should have been parallelized but were not. For these reasons, we focused only on files where OpenMP was used. After filtering, we then collected approximately 140,000 files.

3.3 Data Labeling

For this study, both parallel and serial for-loops were extracted from the gathered files and categorized into two categories: parallelizable and non-parallelizable. An abstract syntax tree (AST) was generated using Clang to accurately identify the for-loops and OpenMP annotations in the source files.

The AST is traversed to identify the nodes corresponding to for-loops and OpenMP directives. We collected the information, including start line, end line, and syntax types, from the identified OpenMP and for-loop nodes. Additionally, while traversing the AST, we annotated each record node with the level of nested loops. This data will not be used when fine-tuning the parallelizable for-loop classification model but will be used when deciding which level of nested loop should be parallelized.

Each for-loop is labeled as either serial or parallel depending on its preceding node within the data collected from the AST. If the prior node corresponds to an OpenMP directive syntax, the for-loop is labeled as parallelizable. If not, it is labeled as non-parallelizable.

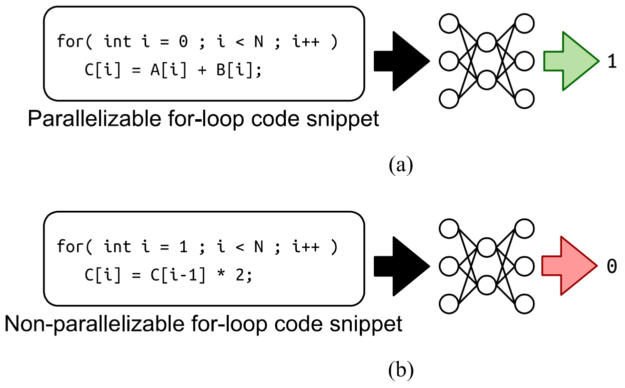
For-loops that do not adhere to correct syntax or contain invalid OpenMP directives are excluded from the dataset. This measure ensures that the model is trained only on valid instances.

3.4 Fine-Tuning the CodeT5 Model

To assess the parallelizability of a for-loop, a pre-trained transfer model was fine-tuned on the dataset we built. The model employed is CodeT5 [24], developed by Salesforce and pre-trained on several programming languages, where it achieves state-of-the-art performance in code-related tasks. From the different available sizes, we selected CodeT5-small to minimize the computational cost during fine-tuning. While this decision may slightly reduce effectiveness relative to larger models, it demonstrates the minimum level of performance in line with other studies utilizing CodeT5 [26].

The training data structure consists of two parts: a for-loop code snippet and label data. A for-loop code snippet includes only the for-loop section in string format. Each for-loop code snippet is associated with label data, either “0” or “1,” where “0” indicates that the loop is a non-parallelizable for-loop and “1” indicates a parallelizable one, as shown in Fig. 2. Before fine-tuning the model with the training data, each word (code) in the code snippet needs to be tokenized into tokens. Once all the snippets are tokenized, we begin the fine-tuning process.

The CodeT5, though trained on 1,000,000 C source files, has not been trained on C++ source files [24]. Given that C++ is a superset of C with a comparable syntax, the decision was made not to further pre-train the CodeT5 model with C++ source files. To accommodate memory constraints, gradient accumulation is employed to facilitate training with larger batch sizes, and early stopping is implemented to minimize overfitting risks.



```
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;
```

1
Parallelizable for-loop code snippet
(a)
0
Non-parallelizable for-loop code snippet
(b)

Fig. 2. (a) Parallelizable code snippets are labeled as “1.” (b) Non-parallelizable code snippets are labeled as “0.”

4. Evaluation Setup

4.1 Hardware and Software

For the fine-tuning process of our model, we employed a server equipped with dual Intel Xeon Gold 6230R CPUs, 256 GB of memory, and two NVIDIA A100 40 GB PCIe cards. Using Clang version 16, the source code was parsed, and ASTs were extracted. To fine-tune the CodeT5 model, Transformers 4.26 was used along with PyTorch 1.13 and CUDA 11.6.

4.2 Fine-Tuning

The dataset curated for fine-tuning comprises 2,017,111 parallel for-loops and 1,117,493 serial for-loops. However, a model trained on such an imbalanced dataset could potentially suffer in terms of performance. To mitigate this, we opted for under-sampling, equalizing the counts of parallel and serial for-loops to 500,000 each, which also had the beneficial effect of reducing training time.

The dataset was divided into training and testing sets at a proportion of 80 to 20, while ensuring the proportionality between parallel and serial for-loops remained consistent. The CodeT5 model was fine-tuned with varying batch sizes (128, 256, 512, and 1,024) at a learning rate set to 2×10^{-4}. Following the fine-tuning process, the optimal number of epochs was determined as 12 epochs for batches of 128 and 256, 25 epochs for 512, and 10 epochs for 1,024, respectively.

4.3 Evaluation Benchmark

To evaluate the effectiveness of the proposed models, we employed two datasets:

GitHub Projects dataset: The constructed dataset was used as the testing data. Duplicate for-loops were identified and removed, potentially introducing bias into the results. The final refined testing dataset comprised 50,000 parallelizable for-loops and 50,000 serial for-loops. This dataset was employed to assess the effectiveness of the proposed model in identifying parallelizable for-loops using OpenMP directives.

NAS Parallel Benchmarks: The NAS Parallel Benchmarks (NPB) [27,28], one of the well-known performance benchmark suites, were used, chosen primarily due to their availability in both serial and parallel implementations with OpenMP in C++. The NPB benchmark suite comprises two divisions of benchmarks, kernel applications and pseudo-applications, as outlined in Table 1.

Table 1. NPB benchmarks list

Kernel applications	Pseudo-applications
Embarrassingly parallel (EP)	Block tri-diagonal solver (BT)
Multi-grid (MG)	Scalar pentadiagonal solver (SP)
Conjugate gradient (CG)	Lower-upper Gauss-Seidel solver (LU)
Discrete 3D fast Fourier transform (FT)	
Integer sort (IS)	

4.4 Baseline

In our comparisons, Clava [8] was employed as the baseline. As a source-to-source compiler, Clava incorporates the AutoPar library, which automates parallelization through static analysis. Clava consistently demonstrates state-of-the-art effectiveness in this domain. To imitate Clava's performance, a custom program was developed to extract for-loops from the NPB Benchmarks. This program identifies all loops, determines which are 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.

4.5 Performance Metrics

This evaluation assessed the effectiveness of the proposed model in categorizing each for-loop as either parallelizable or non-parallelizable within a binary classification framework. To quantify the prediction quality, we employed four metrics: accuracy, F1-score, precision, and recall.

Accuracy measures the model's effectiveness in distinguishing between parallelizable and non-parallelizable for-loops and is represented by the following formulation:

$$Accuracy = \frac{\#\ of\ correct\ predictions}{\#\ of\ predictions}. \tag{1}$$

F1-score represents the harmonic mean of precision and recall. Precision assesses the correctness of the model's parallel for-loop predictions, while recall assesses the comprehensiveness of the model's parallel for-loop identification. Given the inherent compromise between precision and recall, the F1-score serves as a balanced metric between the two. The definitions for precision, recall, and the F1-score are represented by the following formulations:

$$F1 - score = \frac{2 \cdot precision \cdot recall}{precision + recall}, \tag{2}$$

$$Precision = \frac{\#\ of\ correctly\ predicted\ parallelizable\ for - loops}{\#\ of\ predicted\ parallelizable\ for - loops}, \tag{3}$$

$$Recall = \frac{\#\ of\ correctly\ predicted\ parallelizable\ for - loops}{\#\ of\ parallel\ for - loops}. \tag{4}$$

Clava performs source code analysis and recommends OpenMP directives and clauses. Nevertheless, this assessment focuses only on determining if a loop is parallelizable. Thus, even if Clava proposes a potentially incorrect OpenMP directive, the output is considered correct if the loop is parallelizable.

5. Evaluation Results

5.1 GitHub Projects

Table 2 provides the performance evaluation of the proposed models trained on GitHub projects with varying batch sizes. It is noteworthy that the baseline approach demands compilable source code. However, a considerable portion of the dataset files were not compilable, leading to the absence of baseline performance in this evaluation.

All models showed strong performance, with accuracy values centered around 0.84. The model trained with a batch size of 1,024 accomplished the highest F1-score at 0.860 and a recall of 0.973. Although the disparity between the highest and lowest scores is marginal (less than 2%), models trained with larger batch sizes (512 and 1,024) yielded slightly superior results in both accuracy and F1-score.

Table 2. GitHub project evaluation results

Model	Accuracy	F1-score	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**

The best results for each metric are highlighted in bold.

5.2 NAS Parallel Benchmarks

Table 3 details the performance evaluation on the NPB benchmark, comparing our proposed models against the baseline approach, Clava. Consistent with the results from the GitHub project evaluation, the proposed models consistently displayed high accuracy scores (approximately 0.9) across all batch sizes. However, for metrics like the F1-score, precision, and recall, the variance between the highest and lowest scores was more pronounced than in the GitHub dataset. Specifically, the 128-batch size model showed a difference of 21.5% compared to the best model, while the other models exhibited a gap of around 3% in their F1-scores. Clava's recall was superior by 2.8% compared to our best-performing model.

In addition, while the 256-batch size model registered the highest accuracy and F1-score, the 128-batch size model stood out in terms of precision. However, it exhibited substantially diminished recall scores, impacting its overall F1 values.

In comparison to Clava, the baseline tool, most of the proposed models outperformed it across all metrics, with the exception of the recall score. Notably, even the lowest-performing model, excluding the 128-batch size model, yielded slightly better results than the baseline tool. Specifically, differences of 2.9% in accuracy and 6.8% in the F1-score were observed.

Table 3. NPB benchmark evaluation results

Model	Accuracy	F1-score	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**

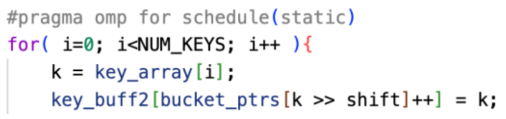
The best results for each metric are highlighted in bold.

6. Discussion

6.1 Analysis of the Correct Predictions

In the NPB benchmarks evaluation, the proposed model successfully identified multiple for-loops as parallelizable, a task at which Clava struggled. This discrepancy can be attributed to two potential reasons.

The first reason pertains to the challenges faced by static analyzers in identifying parallelizable for-loops when runtime information or domain-specific knowledge is necessary. As highlighted in the introduction, static analysis alone can be insufficient in such cases. An illustrative example is shown in Fig. 3. Clava's data dependency analysis marks the for-loop as non-parallelizable because key_buff2 uses bucket_ptrs as an index and is assigned the k variable. Nonetheless, the for-loop is in fact parallelizable, as confirmed by the schedule (static) directive in the OpenMP annotation provided in Fig. 3.

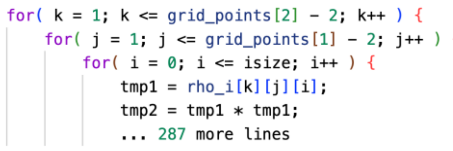


```
#pragma omp for schedule(static)
for( i=0; i<NUM_KEYS; i++ ){
    k = key_array[i];
    key_buff2[bucket_ptrs[k >> shift]++] = k;
```

Fig. 3. A case of parallelizable code successfully identified by the proposed method, which Clava fails to identify due to dependency analysis issues, is found at line 616 in the rank function within the IS benchmark OpenMP variant.

The second reason pertains to the computational characteristics inherent in static analysis. As shown in Fig. 4, Clava is confronted with an out-of-memory exception when analyzing a snippet with multi-level nested for-loops, where the for-loop body spans about 300 lines of code. Clava attempts to inspect all dependencies within the for-loops to ascertain their parallelizability, which increases memory consumption and ultimately triggers an out-of-memory exception. In contrast, the memory consumption of the proposed model remains invariant, regardless of input size or complexity.

The superior performance of our model compared to Clava is attributed to the extensive source code repositories we utilized for fine-tuning the CodeT5 model. While Clava relies on a predefined rigid rule-based approach, our model leverages the embedded knowledge on parallelization extracted from these source codes. As shown in Fig. 3, the rigidity of Clava may prevent accurate predictions if the loop does not match the established rule set. In contrast, our model, which is data-driven, provides a more dynamic and adaptive solution without being bound by strict rule constraints. The effectiveness of our model is intrinsically characterized by both the richness and volume of the training data, which in turn determines the range of parallelization patterns that the model can address. However, the training data essentially contains a wide variety of parallelization patterns, unlike predefined rule-based approaches that can only address a small number of patterns.



```
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
```

Fig. 4. A case of parallelizable code successfully identified by the proposed method, which Clava fails to identify caused by an out-of-memory exception, is found at line 2323 in the x_solve function within the BT benchmark.

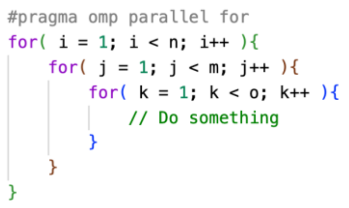
6.2 Analysis of the Mispredictions

As shown in Table 3, the recall scores of our proposed model are notably low. In particular, the 128-batch size model results in a score as low as 0.396. In contrast, Clava manages to surpass other models by a margin of 2.8%. A primary cause behind this poor performance may be the manner in which our input data is designed.

Currently, our training data consists only of code snippets of for-loops, and there is no broad context or inherent characteristics within which these for-loops exist. For instance, our current preprocessing approach decomposes three nested for-loop code snippets, as displayed in Fig. 5, into three separate data entities. In the original source code, only the outermost for-loop has the OpenMP directive and is therefore labeled "parallelizable" in our dataset. The inner for-loops, conversely, are labeled as "non-parallelizable." However, if these inner for-loops appear alone in the source code, they are usually considered suitable for parallelization and deserve their own OpenMP directives. This means that the decision to parallelize a for-loop may depend on its broader context, such as its position in relation to other parallelizable for-loops. These complexities make model training challenging.

A simple approach to remedy this situation might be to exclude for-loops located within parallelized for-loops from the training data. While this strategy would prevent incorrect data from being incorporated into the training dataset, it would deprive the model of the opportunity to capture contextual decisions and simultaneously reduce the amount of training data.

To fundamentally address this issue, a reconfiguration of our training data is imperative. It is important to train the model not only on isolated code snippets but also in conjunction with their encompassing context. A potential solution could involve integrating information about nesting levels within for-loops and whether the outer for-loops are parallelizable. Such an approach is expected to allow the model to better understand the placement of each for-loop in its surrounding context and facilitate accurate predictions regarding whether it should be parallelized by adding OpenMP directives.



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

Fig. 5. Example of a parallel nested for-loops.

7. Conclusion and Future Work

In this research, we introduced a deep learning-based NLP model aimed at automating the parallelization of source code. Through the fine-tuning of CodeT5, an encoder-decoder transformer model, using a custom dataset derived from GitHub repositories containing OpenMP directives, we enabled the model to identify for-loops suitable for parallelization.

The evaluations, conducted on for-loops extracted from both GitHub and the NPB dataset, demonstrated that the proposed models surpassed AutoPar-Clava, the baseline tool used in this study. Distinctively, the new approach accomplished an F1-score of 0.860 in the GitHub dataset evaluation and an F1-score of 0.764 in the NPB evaluation. Contrastingly, Clava accomplished an F1-score of 0.669 in the NPB evaluation.

Future research will focus on developing deep learning models capable of recommending OpenMP directives based on the parallelizable for-loops identified by the proposed model. We also plan to assess the effectiveness improvements delivered by the automatic parallelization tool introduced in this study.

Conflict of Interest

The authors declare that they have no competing interests.

Funding

This work was partially supported by JSPS KAKENHI (Grant No. JP21H03416, JP21K17725, and JP23K16890).

Acknowledgements

This paper is the extended version of “Parallelizable Loop Detection using Pre-trained Transformer Models for Code Understanding,” in the 24th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 23) held in Jeju, Republic of Korea, dated August 16-18, 2023.

References

[1] J. Szuppe, “Boost.Compute: a parallel computing library for C++ based on OpenCL,” in *Proceedings of the 4th International Workshop on OpenCL*, Vienna, Austria, 2016, pp. 1-39. https://doi.org/10.1145/2909437.2909454

[2] L. Tierney, A. J. Rossini, and N. Li, “Snow: a parallel computing framework for the R system,” *International Journal of Parallel Programming*, vol. 37, pp. 78-90, 2009. https://doi.org/10.1007/s10766-008-0077-2

[3] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46-55, 1998. https://doi.org/10.1109/99.660313

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, “HPCToolkit: tools for performance analysis of optimized parallel programs,” *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685-701, 2010. https://doi.org/10.1002/cpe.1553

[5] A. Danner, T. Newhall, and K. C. Webb, “ParaVis: a library for visualizing and debugging parallel applications,” in *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Rio de Janeiro, Brazil, 2019, pp. 326-333. https://doi.org/10.1109/IPDPSW.2019.00062

[6] 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,” *Human-centric Computing and Information Sciences*, vol. 2, article no. 8, 2012. https://doi.org/10.1186/2192-1962-2-8

[7] M. I. Malkawi, “The art of software systems development: reliability, availability, maintainability, performance (RAMP),” *Human-Centric Computing and Information Sciences*, vol. 3, article no. 22, 2013. https://doi.org/10.1186/2192-1962-3-22

[8] H. Arabnejad, J. Bispo, J. G. Barbosa, and J. M. Cardoso, “AutoPar-Clava: an automatic parallelization source-to-source tool for C code applications,” in *Proceedings of the 9th Workshop and 7th workshop on Parallel Programming and Runtime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, Manchester, UK, 2018, pp. 13-19. https://doi.org/10.1145/3183767.3183770

[9] C. Dave, H. Bae, S. J. Min, S. Lee, R. Eigenmann, and S. Midkiff, “Cetus: a source-to-source compiler infrastructure for multicores,” *Computer*, vol. 42, no. 12, pp. 36-42, 2009. https://doi.org/10.1109/MC.2009.385

[10] C. Liao, D. J. Quinlan, J. J. Willcock, and T. Panas, “Extending automatic parallelization to optimize high-level abstractions for multicore,” in *Evolving OpenMP in an Age of Extreme Parallelism.* Heidelberg, Germany: Springer, 2009, pp. 28-41. https://doi.org/10.1007/978-3-642-02303-3_3

[11] G. Mendonca, B. Guimaraes, P. Alves, M. Pereira, G. Araujo, and F. M. Q. Pereira, “DawnCC: automatic annotation for data parallelism and offloading,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, article no. 13, 2017. https://doi.org/10.1145/3084540

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in Neural Information Processing Systems*, vol. 30, pp. 5998-6008, 2017.

[13] J. Shin and J. Nam, “A survey of automatic code generation from natural language,” *Journal of Information Processing Systems*, vol. 17, no. 3, pp. 537-555, 2021. https://doi.org/10.3745/JIPS.04.0216

[14] D. G. Lee and Y. S. Seo, “Improving bug report triage performance using artificial intelligence based document generation model,” *Human-centric Computing and Information Sciences*, vol. 10, article no. 26, 2020. https://doi.org/10.1186/s13673-020-00229-7

[15] H. Husain, H. H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet challenge: evaluating the state of semantic code search,” 2019 [Online]. Available: https://arxiv.org/abs/1909.09436v1.

[16] K. B. Smith, A. J. C. Bik, and X. Tian, “Support for the Intel Pentium 4 Processor with hyper-threading technology in Intel 8.0 Compilers,” *Intel Technology Journal*, vol. 6, no. 1, pp. 19-31, 2002.

[17] H. Bae, D. Mustafa, J. W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff, “The Cetus source-to-source compiler infrastructure: overview and evaluation,” *International Journal of Parallel Programming*, vol. 41, pp. 753-767, 2013. https://doi.org/10.1007/s10766-012-0211-z

[18] D. Quinlan and C. Liao, “The ROSE source-to-source compiler infrastructure,” in *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, Galveston Island, TX, USA, 2011, pp. 1-3.

[19] J. Bispo and J. M. Cardoso, “Clava: C/C++ source-to-source compilation using LARA,” *SoftwareX*, vol. 12, article no. 100565, 2020. https://doi.org/10.1016/j.softx.2020.100565

[20] Q. Zhang, “The accuracy of the non-continuous I test for one-dimensional arrays with references created by induction variables,” *Journal of Information Processing Systems*, vol. 10, no. 4, pp. 523-542, 2014. https://doi.org/10.3745/JIPS.01.0005

[21] J. Devlin, M. W. 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, Volume 1 (Long and Short Papers)*, Minneapolis, MN, 2019, pp. 4171-4186. https://doi.org/10.18653/v1/N19-1423

[22] 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,” *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1-67, 2020.

[23] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” *Proceedings of Machine Learning Research*, vol. 119, pp. 5110-5121, 2020. https://proceedings.mlr.press/v119/kanade20a.html

[24] Y. Wang, W. Wang, S. Joty, and S. C. 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*, Punta Cana, Dominican Republic, 2021, pp. 8696-8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[25] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. Santos, “An empirical study of code smells in transformer-based code generation techniques,” in *Proceedings of 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Limassol, Cyprus, 2022, pp. 71-82. https://doi.org/10.1109/SCAM55253.2022.00014

[26] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. 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 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Madrid, Spain, 2021, pp. 336-347. https://doi.org/10.1109/ICSE43902.2021.00041

[27] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, et al., “The NAS parallel benchmarks,” *International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63-73, 1991. https://doi.org/10.1177/109434209100500306

[28] J. Loff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, “The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures,” *Future Generation Computer Systems*, vol. 125, pp. 743-757, 2021. https://doi.org/10.1016/j.future.2021.07.021

Soratouch Pornmaneerattanatri **https://orcid.org/0009-0005-5029-1437**

He received his B.S. degree in Software and Knowledge Engineering and his M.S. degrees in Computer Engineering from Kasetsart University, Bangkok, Thailand. He is currently pursuing a Ph.D. degree and working as a research assistant in Software Design and Analysis Laboratory at Nara Institute of Science and Technology, Nara, Japan. His research interests include high-performance computing and infrastructure technology, and cloud technology.

Keichi Takahashi **https://orcid.org/0000-0002-1607-5694**

He is an assistant professor at Tohoku University, Sendai, Japan. He received his M.S. and Ph.D. degrees in Information Science from Osaka University, Osaka, Japan in 2016 and 2019, respectively. In 2018, he was a visiting scholar at the Oak Ridge National Laboratory, Oak Ridge, TN, USA. He was an assistant professor at the Nara Institute of Science and Technology, Nara, Japan from 2019 to 2021. His research interests include high performance computing and parallel distributed computing.

Yutaro Kashiwa **https://orcid.org/0000-0002-9633-7577**

He is an assistant professor at Nara Institute of Science and Technology, Japan. He worked for Hitachi Ltd., as a full-time software engineer for 2 years before spending 3 years as a research fellow of the Japan Society for the Promotion of Science. He received his Ph.D. degree in engineering from Wakayama University in 2020. His research interests include empirical software engineering, specifically the analysis of software bugs, testing, refactoring, and release.

Kohei Ichikawa **https://orcid.org/0000-0003-0094-3984**

He received the B.E., M.S., and Ph.D. degrees from Osaka University, in 2003, 2005, and 2008, respectively. He was a Postdoctoral Fellow at the Research Center of Socionetwork Strategies, Kansai University, from 2008 to 2009. He has also worked as an assistant professor at the Central Office for Information Infrastructure, Osaka University, from 2009 to 2012. He is currently an Associate Professor with the Division of Information Science, Nara Institute of Science and Technology (NAIST), Japan. His current research interests include distributed systems, virtualization technologies, and software defined networking.

Hajimu Iida **https://orcid.org/0000-0002-2919-6620**

He received his B.E., M.E., and Dr. of Eng. degrees from Osaka University in 1988, 1990, and 1993, respectively. From 1991 to 1995, he worked for the Department of Information and Computer Science, Faculty of Engineering Science, Osaka University as a research associate. Since 1995 he has been with the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. His current position is a professor of the Laboratory of Software Design and Analysis. His research interests include modeling and analysis of software and development process.