

PragFormer: Data-driven Parallel Source Code Classification with Transformers

Re'em Harel

Ben-Gurion University of the Negev

Tal Kadosh

Ben-Gurion University of the Negev

Niranjana Hasabnis

Intel (United States)

Timothy Mattson

Intel (United States)

Yuval Pinter

Ben-Gurion University of the Negev

Gal Oren

`galoren@cs.technion.ac.il`

Technion – Israel Institute of Technology

Research Article

Keywords: parallel programming, artificial intelligence, software development, programming assistance

Posted Date: August 30th, 2023

DOI: <https://doi.org/10.21203/rs.3.rs-3254961/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at International Journal of Parallel Programming on October 28th, 2024. See the published version at <https://doi.org/10.1007/s10766-024-00778-9>.

PRAGFORMER: Data-driven Parallel Source Code Classification with Transformers

Re'em Harel^{1,2}, Tal Kadosh^{1,3}, Niranjan Hasabnis⁴, Timothy Mattson⁴, Yuval Pinter¹, Gal Oren^{2,5}

¹Ben-Gurion University, Israel.

²NRCN, Israel.

³IAEC, Israel.

⁴Intel Labs, United States.

⁵Technion, Israel.

Abstract

Multi-core shared memory architectures have become ubiquitous in computing hardware nowadays. As a result, there is a growing need to fully utilize these architectures by introducing appropriate parallelization schemes, such as OpenMP worksharing-loop constructs, to applications. However, most developers find introducing OpenMP directives to their code hard due to pervasive pitfalls in managing parallel shared memory. To assist developers in this process, many compilers, as well as source-to-source (S2S) translation tools, have been developed over the years, tasked with inserting OpenMP directives into code automatically. In addition to having limited robustness to their input format, these compilers still do not achieve satisfactory coverage and precision in locating parallelizable code and generating appropriate directives. Recently, many data-driven AI-based code completion (CC) tools, such as GitHub CoPilot, have been developed to ease and improve programming productivity. Leveraging the insights from existing AI-based programming-assistance tools, this work presents a novel AI model that can serve as a parallel-programming assistant*. Specifically, our model, named PRAGFORMER, is tasked with identifying `for` loops that can benefit from conversion to parallel worksharing-loop construct (OpenMP directive) and even predict the need for specific data-sharing attributes clauses on the fly. We created a unique database, named OPEN-OMP, specifically for this goal. OPEN-OMP contains over 32,000 unique code snippets from different domains, half of which contain OpenMP directives, while the other half do not. We experimented with different model design parameters for these tasks and showed that our best-performing model outperforms a statistically-trained baseline as well as

*We have published preliminary results of this work in 2022 [1] and 2023 [2]. This paper extends the preliminary results with more comprehensive experimental evaluations.

✉ Corresponding author: Gal Oren, galoren@cs.technion.ac.il

a state-of-the-art S2S compiler. In fact, it even outperforms the popular generative AI model of ChatGPT. In the spirit of advancing research on this topic, we have already released source code for PRAGFORMER as well as OPEN-OMP dataset to public.[†] Moreover, an interactive demo of our tool, as well as a Hugging Face webpage [‡]to experiment with our tool, are already available.

Keywords: parallel programming, artificial intelligence, software development, programming assistance

1 Introduction

The world has shifted towards multi-core, and many-core shared memory architectures in past decades. Due to the constant need for increased performance, computational paradigms needed to be adapted to these architectures by introducing shared memory parallelism.

The most comprehensive parallelization API that implements the shared memory model is the OpenMP API [3]. The OpenMP API consists of a set of compiler directives (*pragmas*), library routines, and environment variables that allow a program to be executed in parallel (multi-threaded) within a shared memory environment and exploit the potential of many-core and multi-core architectures. One of the main advantages of the OpenMP API, which also contributes to its popularity, is its flexible and straightforward interface that is readable and easily interpretable.

Nevertheless, converting a sequential program into a parallel program by introducing *correct* OpenMP pragmas is a complex and tedious process. This is primarily due to ubiquitous pitfalls in managing parallel shared memory, architecture heterogeneity, and the need for human expertise to comprehend many fine details and abstract correlations [4].

That is why automatic parallelization tools that assist programmers in converting sequential programs into parallel were developed [5, 6]. Most, if not all, automatic parallelization tools typically target loops because programs mostly spend their time in loops. Because automatic parallelization is a type of program transformation, popular compilers, such as GCC [7], LLVM [8–10], ICC [11], etc., are naturally the logical approach to perform automatic parallelization as a part of the compilation process¹. These compilers conceptually offer an ideal solution for automatic parallelization also. First, these compilers relieve programmers from manually adding parallelization directives in the code. Additionally, they eliminate the need for a separate source-to-source transformation pass for parallelization, thus simplifying the software development process. Moreover, these compilers can ensure the soundness of the automatic parallelization process by reusing their program analysis infrastructure, such as data-flow

[†]<https://github.com/pragformer/PragFormer>

[‡]<https://huggingface.co/spaces/Pragformer/PragFormer-demo>

¹In fact, these compilers have supported automatic parallelization for a few years now. For example, GCC first supported automatic parallelization in GCC-4.3 in 2012. LLVM’s Polly, which can be accessed from `clang`, is a loop optimizer that can also perform automatic parallelization.

<pre> 1 // Serial reduction 2 for (int i = 0; i < N; i++) { 3 d2 += a[i] * b[i]; 4 } </pre>	<pre> 1 // Input functions such as scanf 2 for (i = 0; i < n; i++) { 3 scanf("%d", &a[i]); 4 } </pre>
--	--

(a) OpenMP advanced features

(b) Handling I/O functions

Fig. 1: Motivating examples highlighting the limitations of S2S compilers

analysis, to reason about potential constraints that may prevent automatic parallelization. Finally, these compilers typically employ heuristics to ensure that parallelized loops do not degrade performance (e.g., the number of iterations of a loop need to be high [7].)

An alternative approach to automatic parallelization is to perform a source-to-source translation. The tools that employ this approach are called source-to-source translation tools (S2S, in short). These tools offer a couple of advantages over parallelizing compilers. First, these tools can keep the source code clean — typically, they do not modify the original source code. More importantly, they open up the possibility of using any standard tool — a standard compiler, a vendor-specific tool — to subsequently compile the parallelized code. Some examples of S2S tools are ComPar [12], Par4all [13, 14], and Cetus [15].

General passes	Parallelization passes
Symbolic analysis	Induction variable recognition/substitution
Points-to/alias analysis	Reduction recognition/transformation
Function inlining	Scalar/array privatization
USE/DEF chain analysis	Data dependence analysis
Miscellaneous	Loop parallelizer

Table 1: Major passes implemented in ComPar (using Cetus)

Although automatic parallelization tools provide a solution for code parallelization, they have many pitfalls. We discuss two key pitfalls below. A more detailed analysis of the limitations of these tools, such as leading to fragile software development workflows, interference with downstream compiler optimizations, etc., is already discussed in detail in literature [5, 16–18].

- *Manual development efforts.* Automatic parallelization tools are typically developed manually as rule-based, pattern-matching tools. Unfortunately, implementing these tools is not a one-time effort; maintaining these tools by supporting the revisions to OpenMP API specification also requires continuous manual effort. It is unsurprising to imagine that these tools have difficulty keeping up with the latest OpenMP API specifications. GCC’s latest version, for instance, does not support all the features of OpenMP v5.0 [19]. As another example, our evaluation of existing S2S tools revealed that none of the existing S2S compilers even suggested an OpenMP API between version 3 and version 5.2 (such as handling

irregular parallelism with *task* or offloading kernels to devices with *target*) — version 3 of OpenMP API was introduced in 2008.

Figure 1 provides two specific examples that highlight the limitations of Cetus² (v2.0). In particular, for example, on the left, which models a reduction pattern, Cetus does not support OpenMP’s SIMD-based parallel reduction feature (via `pragma simd reduction(+:reduction_var)`) and misses a possible performance improvement offered by the vectorization. To be specific, this example can be best parallelized using `#pragma omp parallel for simd reduction(+:d2)` before the `for` loop. And for the example on the right, Cetus performs dynamic analysis on the code during the execution to learn the run-time characteristics. However, this dynamic analysis causes Cetus to fail to terminate when the code contains functions related to the user input, such as `scanf`.

- *Inability to handle uncompileable code or code in-development.* A more fundamental limitation of these tools is their inability to handle incomplete code (or code in-development) as input. This limitation stems from the fact that these tools are designed to operate on complete/compileable programs by relying on standard compiler passes such as parsing, semantic analysis, etc., that reason about the potential of code to be parallelized. ComPar, for instance, relies upon a list of analysis passes mentioned in Table 1.

Although automatic parallelization tools have contributed to the field of parallel programming, their inability to process uncompileable code precludes these tools from being deployed as developer-assistance tools that can provide immediate feedback during programming. Developer-assistance tools (mostly available as a part of code editors such as Visual Studio Code (VS Code, in short) serve a useful purpose of providing immediate feedback to developers to speed up their development process. For instance, by default, VS Code is loaded with several useful features, such as highlighting syntax errors, auto-completing code fragments, etc., that speed up the process of writing code. Recently though, with the rapid advancements in AI, and more specifically, in the sub-fields of natural language processing (NLP) and deep learning, the field of AI-based developer assistance tools [20, 21] is advancing rapidly. Several AI-based tools for purposes such as static analysis, code completion [22], bug finding, documentation, etc., already exist. VS Code, for instance, already provides a suite of AI-based tools that can suggest *context-specific* code completions and automatically generate documentation and tests from code, among others. GitHub CoPilot [23] and ChatGPT are popular AI-based tools that can write code and/or find bugs.

An important advantage offered by AI-based tools is that they also reduce, if not eliminate, the manual efforts required to maintain the tools (the first limitation above). Specifically, *AI-based tools exploit the insight that one can learn characteristics of code, such as syntax [24], translation rules [25], etc., by leveraging massive amounts of existing source code (e.g., using open-source code repositories from code hosting sites such as GitHub).* Learning from data eliminates the need to program these tools explicitly. For instance, ControlFlag [25], a data-driven programming anomaly detector, can automatically learn language-level typing rules without explicit programming.

²<http://icetus.ece.udel.edu/cetusWeb>

Specifically for the example shown on the left in Figure 1, we found its version has appropriate SIMD-based parallel reduction pragma in several GitHub repositories.

Of course, correctness is a concern for AI-based (probabilistic) tools that automatically learn from data/code. This is one key advantage that S2S tools (or those based on deterministic/formal approaches) have over probabilistic tools. Nevertheless, recent AI-based tools have shown that the limitation of the probabilistic tools can be mitigated by applying them to problems where probabilistic output is acceptable and/or to problems where humans can act upon the output of these tools (*human-in-the-loop* approach). We observe that this is one of the popular deployment models of recent AI-based tools, such as GitHub CoPilot, that assist developers in their popular code editors.

In this paper, we propose an alternative to existing auto-parallelization tools by applying the insights — vast availability of open-source code and learning rules from them to reduce manual efforts — exploited by existing AI-based tools for software development. Specifically, *we hypothesize that the rules for automatically converting serial code into (OpenMP-based) parallel code can be learned from existing code repositories by applying the latest advancements in the field of NLP and deep learning.* Towards that end, we propose and develop PRAGFORMER, an AI-based tool for automatic parallelization that learns to suggest appropriate OpenMP pragmas and data-sharing attributes (clauses) for C/C++ code. We believe that PRAGFORMER can be deployed in code editors as a programming assistance tool whose output can be acted upon by the programmer, thus mitigating the need for correctness guarantees. Nevertheless, as with any AI-based tool, we empirically evaluate the correctness of PRAGFORMER. Exploiting the advancements from NLP, PRAGFORMER can also accept the raw text of source code as input, eliminating the need for a parser. More importantly, PRAGFORMER can serve as a code advisor to developers by operating on incomplete/uncompilable serial code. Experimental results obtained from code collected from over 8,000 repositories demonstrate that PRAGFORMER outperforms state-of-the-art auto-parallelization tools in recognizing the potential of parallelizing serial code. We also compare PRAGFORMER with ChatGPT, a popular generative AI model that can answer programming related questions, to find that PRAGFORMER outperforms ChatGPT as well. We also demonstrate the practicality of our tool by deploying it on top of HuggingFace. And we have also released our source code as well as the datasets to the open-source community to advance this field further.

Contributions. This paper makes following contributions:

- To the best of our knowledge, PRAGFORMER is the first effort at applying the latest advancements from the field of NLP and deep learning to the problem of automatic parallelization.
- As a part of this work, we have also synthesized a large corpora of serial and parallel code snippets. To the best of our knowledge, no such dataset already exists. In the spirit of advancing research on this topic, we have already released our source code and dataset to public (links withheld for anonymity).
- Our experimental results on two sub-tasks of auto-parallelization reveal that PRAGFORMER outperforms competing approaches (including a state-of-the-art

S2S compiler), including the popular generative AI model of ChatGPT. We also attempt at explaining PRAGFORMER’s decisions.

2 Related Work

In this section, we first discuss related S2S automatic parallelization tools that typically rely on rule-based approaches (§2.1). Although, to the best of our knowledge, no AI-based programming assistant for code parallelization exists, recent generative AI tools, such as ChatGPT, can be applied to the task. We discuss their limitations with specific examples (§2.2).

2.1 Rule-based Methods

Along with the evolution of the multi-core era, the need for standards for parallel programming was evident. This was when OpenMP became one of the open standards for parallel programming. Although OpenMP offered the ability to programmers to parallelize their code, most, if not all, of the legacy code was serial code. This was when the need for automatic parallelization was evident.

Compiler-based auto-parallelization

Given the central role played by compilers in processing source programs, it was logical and intuitive to introduce automatic parallelization capability in them. GCC [7], LLVM [8, 10], and even commercial compilers such as Intel’s ICC [11] have supported automatic parallelization for a few years now. These compilers leverage their dataflow analysis frameworks to detect loops that can be safely and efficiently executed in parallel. Safety comes from applying program analysis techniques to reason about loop-carried dependencies and other constraints (e.g., pointer aliasing) that may prevent automatic parallelization. Efficiency comes from heuristics that are typically used to estimate the workload of a loop and potential improvement after parallelizing it. Given the need for manual efforts to maintain these compilers, they lag behind in supporting the latest OpenMP features [19].

Auto-parallelization Based on Source-to-source (S2S) Translation

Automatic parallelization pass in compilers modify source code in place and are also compiler-specific. S2S translation-based tools offered to eliminate these limitations of existing compilers. Several S2S compilers, including Cetus [26] and Par4All [13, 14], have emerged in the last decade to insert OpenMP pragmas into code automatically.

Both types of auto-parallelizing tools — compilers as well as S2S translation-based tools — are technically implemented as static, rule-based methods, which often have limited capabilities and robustness. At a high level, these tools rely on program analysis-based techniques to analyze and identify potential constraints (e.g., loop-carried dependencies) that may restrict the code from being parallelized. Their workflow can be summarized as follows: 1. Create an abstract syntax tree (AST) [27], which is a tree representation of the code’s syntactic structure, 2. Apply data dependence algorithms [28] to ASTs, 3. Produce appropriate OpenMP directives based on the data dependence graph.

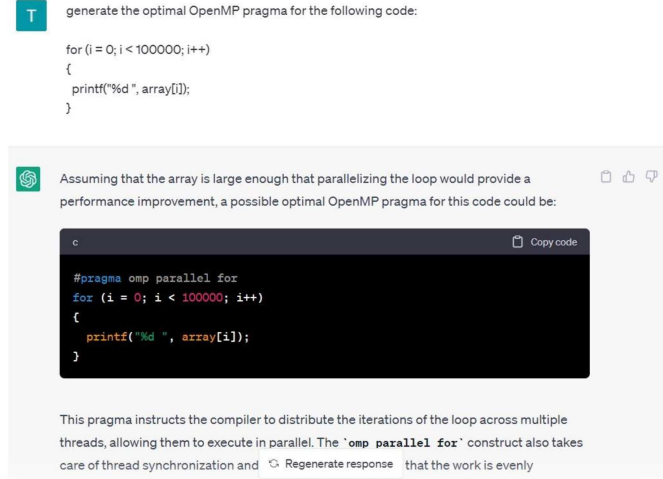


Fig. 2: ChatGPT example of OpenMP Parallelization Task: “generate the optimal OpenMP pragma if possible”.

There are multiple drawbacks associated with the approach of generating ASTs and applying data dependence algorithms. First, creating an AST with a parser can be a challenging task with limited robustness to input due to each programming language’s unique syntactic structures that have evolved over the years. Thus, many S2S compilers cannot handle the diverse syntax of programming languages. Moreover, not all parsers are publicly available. As a result, some S2S compilers may fail to produce an AST and analyze the input code. Secondly, data-dependence algorithms can be time-consuming, particularly for large-scale code, since these algorithms are strongly dependent on the size of the AST, which in turn is influenced by the length of the code. Additionally, studies by Harel et al. [16] and Prema et al. [18] have shown that existing rule-based tools may produce sub-optimal results and even degrade program performance in some cases.

2.2 AI-based Programming Assistants

One of the important limitations of rule-based tools is that they require manual programming efforts to add new rules to maintain and update them. However, recent AI-based programming assistance tools have shown that it is possible to reduce manual efforts by instead learning them from data.

Although, to the best of our knowledge, no AI-based tool exists to auto-parallelization using OpenMP pragmas, recent large language models such as Codex [29] (which powers GitHub CoPilot [23]), ChatGPT [30], etc., can be applied to this task by asking appropriate questions (*prompt*) — after all, these models are

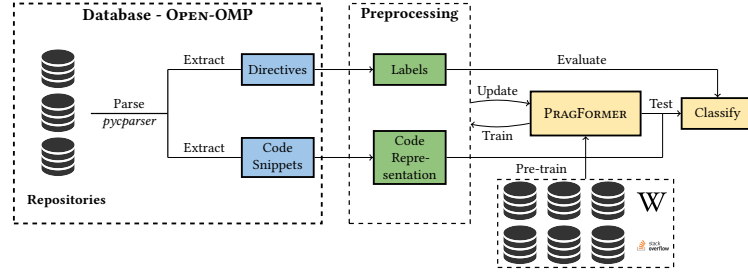


Fig. 3: Overview of the workflow for classifying OpenMP directives and clauses. PRAGFORMER is our proposed model.

huge³ (and trained on large corpora of open-source code⁴ (which would also include parallel code containing OpenMP pragmas). Moreover, ChatGPT has been trained (using reinforcement learning with human feedback [32]) to follow instructions in a prompt and provide detailed responses. It is natural, then, to wonder if these models can parallelize a piece of serial code. In our evaluation of ChatGPT, however, we found that its suggestions were overly optimistic, frequently suggesting parallelization when it was impossible or counter-productive. For instance, we used the prompt “*generate the optimal OpenMP pragma if possible*” for a `for` loop containing `printf` function from C language (Figure 2) to find that ChatGPT suggested an OpenMP pragma for it (which is incorrect as it would lead to a race condition). The lack of details of ChatGPT makes it difficult to analyze its performance on this task. Nevertheless, later in the evaluation, we show that PRAGFORMER outperforms ChatGPT on this task and can recognize key code characteristics that can lead to code parallelization decisions.

3 PragFormer: A Transformer-based Parallel Programming Assistant

Given the limitations of existing approaches for auto-parallelization, we aim to answer the question of *if it is possible to parallelize a `for` loop, without having access to rest of the code, and then determine appropriate OpenMP pragma*. Towards that end, we propose a novel model, named PRAGFORMER, that given a piece of serial `for` loop, determines if a loop can be converted into a parallel worksharing loop and, if so, its appropriate data-sharing attributes clauses (*private* and *reduction*). Given the recent advancements in NLP and deep-learning-based code comprehension techniques, we base PRAGFORMER on the transformer architecture [33]. The workflow of applying it is presented in Figure 3. This pipeline resembles the workflow of program comprehension models [21] and of S2S compilers: (1) tokenize the source code in order to allow its ingestion by the transformer model (§3.2); (2) train the transformer-based model (§3.1) to classify OpenMP directives from the data segments in the corpus; (3) and evaluate

³Codex was produced by OpenAI researchers by fine-tuning GPT models containing up to 12B parameters.

⁴Quoting from OpenAI’s Codex webpage [31]: *its training data contains both natural language and billions of lines of source code from publicly available sources, including code in public GitHub repositories. Moreover, it is also proficient in over a dozen languages*

the predicted OpenMP directives, as described in §5. In this work, we train models for two tasks:

1. **Given a serial loop, can it be parallelized?** Given a (serial) `for` loop, this task determines the possibility of parallelizing a loop using OpenMP (i.e., converting into a parallel work-sharing loop using `pragma omp parallel for`).
2. **Need a private or reduction clause?** One of the problems in executing a program concurrently is possible data races, which under some circumstances, cause the code to be inconsistent across separate runs. In some cases, these problems can be eliminated by adding specific data-sharing attributes clauses [34], such as the `private` clause and the `reduction` clause. Thus, after the first task determines that a loop can be parallelized as a work-sharing loop construct, this task determines: Whether the data-sharing attributes of variables should be changed to a thread-local copy, i.e., a `private` or a `reduction` clause.

3.1 Model

PRAGFORMER comprises a transformer model followed by a fully-connected (FC) layer for classification.

A brief background on Transformer architecture. The transformer architecture [33] has gained popularity due to its impressive ability to leverage data from vast unlabeled sequence resources in a *pre-training* phase and apply it to a large range of end tasks, a property known as *transfer learning*. At a high level, the transformer is a model which receives a sequence of vectors as its input and passes it through a massively-parameterized *encoder* module, which outputs a sequence of *contextualized* vectors of the same length. In theory, the output vectors become more and more context-dependent after each layer’s treatment. The main mechanism allowing this behavior is the *self-attention* component within the blocks that calculates a numerical score. The score then determines the level of consideration in preparing this position’s output for ingestion by the next layer. In the realm of source code, attention scores may represent how much influence each of the other variables or statements have over a given input variable’s contextualized vector. This enables even elements that are far away, position-wise, to affect any output; a typical transformer model’s sequence length is capped at hundreds of tokens.

We implement PRAGFORMER based on the pre-trained state-of-the-art DeepSCC [35] model. Similarly to our task, which is identifying and advising the need for an OpenMP directive, DeepSCC was trained with the aim of classifying the programming language of a source code. In DeepSCC, the self-attention mechanism helps focus on specific scopes (of variables or operations) to determine the programming language. This mechanism is crucial in our task, as the identification of the need for an OpenMP directive is hinted at long-range dependencies in the source code, which the self-attention mechanism of the transformer model is most suited to. As such, it can be seen as an instance of transfer learning into a *low-resource* scenario, a technique shown to be effective in many applications of transformers in NLP [36–38]. DeepSCC was created by fine-tuning a pre-trained RoBERTa [39] model on a large corpus of source code files (including C and C++, which approximate our corpus) via the masked language

model (MLM) objective [40]. Ultimately, DeepSCC manages to fine-tune the parameters of the RoBERTa model, which were originally trained to process English text, to be better suited for source code as well as suppressing the results of other models tasked with the same objective (achieving 87% accuracy). Thus, we believe DeepSCC provides adequate initialization and providing an apt starting point for PRAGFORMER. Following the encoder’s operation over source code input, the output vectors are fed into an FC layer which predicts a binary label based on the individual task (identification of a need for a directive or a specific clause) through a softmax layer that transforms the real-valued scores output from the FC layer into probabilities.

3.2 Code Representation

Raw source code cannot serve as input to an ML model; instead, transforming the code to a sequence of tokens from a pre-set vocabulary is needed. This transformation process, known as *tokenization*, is then followed by associating each keyed token with a numerical vector (*embedding*) which is used as input to the model, to be modified along with the rest of the model’s parameters during training.

One possible way to tokenize source code is as if it were text. This way, each keyword, operator, identifier, and symbol would be assigned its own token [21]. However, due to the structured nature of code, previous work [41, 42] suggests that tokenizing the abstract representation of the code (such as AST or CDG) rather than the raw code itself can be helpful in order to perform source code analysis. Representing the code this way has been shown to improve results in various code understanding tasks compared to a lexical representation [43, 44]. Another line of work proposes a combination of AST and raw code, either by embedding them separately and combining their vectors inside the model or by mixing them directly at the textual level, finding it to be beneficial over using each kind separately [45, 46].

Although other source code representations achieve proper word embeddings, such as *word2vec* [47], *IR2vec* [48] and *Programl* [49], the tokenizer and the pre-trained model (DeepSCC and RoBERTa) were trained exclusively on natural language text. Therefore, exploiting the transfer learning property with different representations other than text will likely produce poor results (as exemplified by the AST representation in this work). Overcoming this limit can be achieved by training the tokenizer and the base model (i.e., RoBERTa) from scratch with a large corpus of the appropriate representation. Therefore, we believe that it might be easier for PRAGFORMER to understand the dependencies from raw source code rather than from its AST representation. We nevertheless evaluate this decision via experiments.

A frequent problem in applying pre-set vocabulary tokenizers to any textual input is the representation of strings not present in the vocabulary, known as out-of-vocabulary items, or *OOVs* [50]. In the code domain, the OOV problem can manifest itself mainly through identifier naming conventions, which vary greatly across organizations and communities, which harms the model performance [43, 45, 51]. While most previous work [43, 45, 51, 52] has opted for the *UNK* solution, classifying OpenMP directives does not require knowledge of individual identifier names, and we thus replace them during a pre-processing step with predefined indexed words such as *v1*, *v2*.

Repr	Example
<i>Text</i>	<code>for (i = 0; i < len; i++) a[i] = i;</code>
<i>R-Text</i>	<code>for (v0 = 0; v0 < v1; v0++) arr0[v0] = v0;</code>
<i>AST</i>	For: Assign: = ID: i Constant: int, 0...
<i>R-AST</i>	For: Assign: = ID: v0 Constant: int, 0...

Table 2: Different code representations of `for (i = 0; i < len; i++) a[i] = i;`

Altogether, we experimented with four different code representations for our model: text (*Text*), text with replaced identifiers (*R-Text*), AST (*AST*), and AST with replaced identifiers (*R-AST*). An example of each of the four representations is shown in Table 2. Although, we mentioned that ASTs (using parsers) would restrict the scope of the approach to complete programs, we decided to consider them to compare against text based representations.

The *Text* representation contains 6,427 tokens with 398 OOVs in the test set; the *R-Text* contains 2,424 tokens with 220 OOVs in the test set; the *AST* representation contains 5,261 tokens with 348 OOVs in the test set; and the *R-AST* with 3,409 with 309 OOVs in the test set. In terms of current transformer models, such vocabulary sizes and OOV token (or type) counts as the ones present in all representations are relatively small, suggesting that the raw textual representation might be sufficient for obtaining good results. In addition, the average number of tokens per code snippet is lower for text representation as the AST representation adds words used to describe operational logic.

3.3 Implementation

In PRAGFORMER, we use the tokenizer from DeepSCC-RoBERTa. The largest token length in a code snippet was 110, and so it was set as the maximum length for the input to the model. We fine-tuned the DeepSCC model in PRAGFORMER to our dataset, meaning the internal parameters already pre-trained on the language modeling task were also updated during our model’s training phase. The FC layer in PRAGFORMER contains two dense layers with a ReLU activation function between them. We implemented dropout as a regularization strategy to avoid overfitting the training set [53]. We updated model parameters using backpropagation, implemented via the AdamW gradient descent optimizer [54].

4 Data

Although, there exists a number of publicly-available source code datasets, we could not find a dataset of `for` loops (serial code) and their corresponding OpenMP pragmas. That is why we decided to create a dataset, or corpus, of code files, which we call OPEN-OMP⁵. In the spirit of advancing research on this topic further, we have released the dataset publicly (link withheld for anonymity).

⁵https://github.com/pragformer/PragFormer/blob/main/Open_OMP.tar.gz

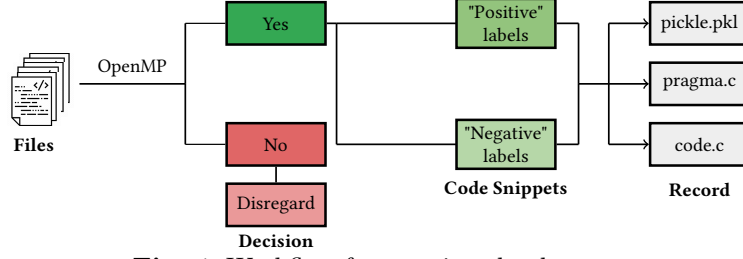


Fig. 4: Workflow for creating the dataset.

4.1 Dataset Creation

We employed a simple idea for synthesizing a dataset of (serial) `for` loops and their applicable OpenMP pragmas: mine (parallel) `for` loops having an OpenMP pragma before them, and remove those pragmas to turn them into serial loops and store the pragma and its respective serial loop in the dataset. Towards that end, we queried github.com, the most well-known code repository hosting website, via a script named *github-clone-all*⁶, which allows searching for source files by a repository. We extracted only C files from repositories, containing the phrase OpenMP in the title, description, or *README* file. This query resulted in more than 20,000 files from approximately 8,000 different repositories.

Inclusion criteria. Among the 20,000 files, we considered only files written in C and containing an OpenMP directive. We used *pycparser* [55], a Python-based C code parser, to identify these files. The parser creates an AST of the source code, and by traversing the tree, one can easily identify worksharing-loop constructs. As our first task needs to answer the question of whether or not a code segment is even suitable for parallelization, the code corpus must also include code examples that are not explicitly parallelizable (negative examples to learn from). We limited our negative examples to `for` loops without OpenMP directives appearing in files where elsewhere such directives do exist. We believe that this rules out cases where code amenable to directives was not annotated due to developers unfamiliar with parallelization schemes or those who work with incompatible hardware.

Exclusion criteria. Some OpenMP directives are nearly impossible to predict automatically from code alone; for example, the *task* construct requires extensive background knowledge regarding the logic of the code. Thus, in accordance with S2S compilers, we only include worksharing-loop constructs defined over a `for`-loop segment (directives containing `#pragma omp parallel for`) in the corpus.

In addition, we found numerous repositories containing examples of OpenMP directives created solely for testing compiler compatibility. These examples, which contained an empty `for` loop, were also excluded from the corpus. In the developer community, it is common to copy source code from other existing projects. Therefore, to verify that each entry is unique, it was scanned for similar entries, and replicas were removed. The corpus thus contains two types of records: those that contain examples

⁶[http://github.com/rhysd/github-clone-all](https://github.com/rhysd/github-clone-all)

Description	Num	# Line	Num	Domain	%
Total code snippets	32,099	< 10	17,836	Benchmark	16.5%
OpenMP directives	14,906	11–50	13,046	Testing	7%
reduction	2,147	51–100	782	Generic	43%
private	5,568	> 100	435	Unknown	33.5%

(a) Code snippet types (b) Code snippet lengths (c) Code snippet domains

Table 3: Statistics related to OPEN-OMP.

with OpenMP directives and those that do not. Figure 4 summarizes the creation process of the corpus.

4.2 Dataset Statistics

We present statistics of our database by: (1) the different OpenMP directives and clauses count (Table 3a), (2) the number of lines count of the code snippets (Table 3b). As can be seen, most of the database contains small to medium code snippets with diverse OpenMP directives and mostly from real applications ⁷

To determine the domain distribution of the code snippets, we searched the README file attached to the snippet’s repository for specific keywords. 16.5% of the README files contained the keyword “benchmark”, 7% had keywords about testing, 43% without any of these keywords, while 33.5% did not had any README file.

We then created two different datasets, one for each task described earlier. We obtained the dataset for the first task by sampling 31984 code samples from our base database, with 50% of 31984 code samples amenable for parallelization (positive examples with OpenMP pragma), while the remaining 50% otherwise. The dataset for the second task of clause determination contained 14800 code samples, obtained by sampling from 14906 samples known to have OpenMP directives. Of 14800 code samples, 50% had either **private** or **reduction** class, while the remaining had none of these clauses. The train/validation/test splits in both the datasets were performed using the standard ratio of 80:10:10 while maintaining a balanced positive-negative label distribution.

Dataset	Task 1	Task 2
Training	25,588	11,840
Validation	3,198	1,480
Test	3,198	1,480
Total	31984	14800

Table 4: Dataset splits for the tasks

5 Results

PRAGFORMER in its current mode acts as a programmer-assistant tool and not as a parallel code generator tool. In other words, it does not produce OpenMP pragmas for a serial code, but rather predicts if a serial code can be parallelized and if so, if any data-sharing clause would apply. As such, the premise of this work is that a

⁷Full GitHub repositories by domain distribution: https://github.com/pragformer/PragFormer/blob/main/log_repositories_readme.txt

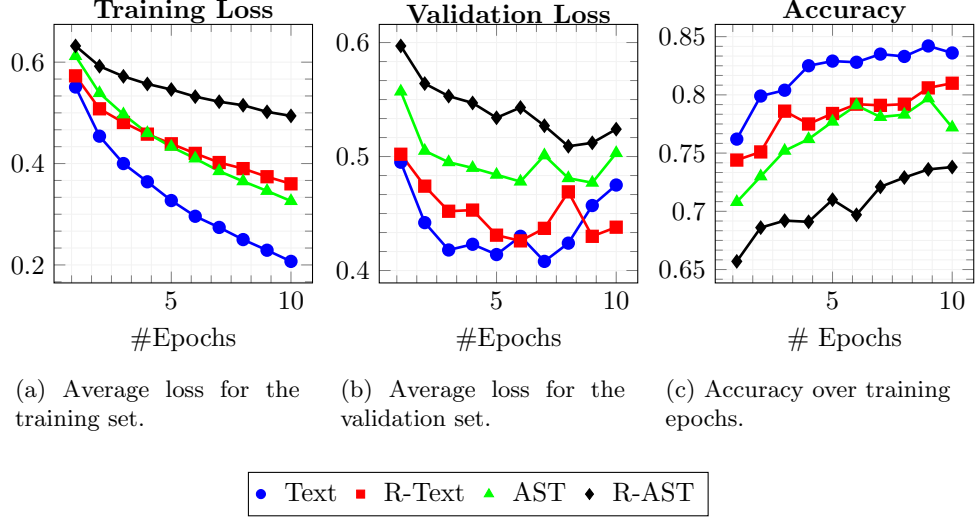


Fig. 5: The training loss, validation loss and accuracy of the four source code representations of PRAGFORMER.

human-expert has produced the annotated positive labels and that they correspond with the best performance achieved by the directive. Therefore, the evaluation of PRAGFORMER is performed by measuring the number of correct predictions against the actual label and not by measuring performance.

To evaluate the effectiveness of PRAGFORMER, several experiments are conducted over the datasets synthesized for two tasks. We first determine the best code representation for the tasks (cf. §3.2), then follow up by comparing the best-represented PRAGFORMER variant against other models, namely a state-of-the-art S2S compiler and a standard machine learning (ML) model.

5.1 Code Representation

A crucial step in performing source code analysis with ML models is representing the code so that the model can extract the maximum information from it and infer upon it. We compare the performance of models trained on each of the four code representations presented in §3.2, namely: text (*Text*), replaced text (*R-Text*), AST (*AST*), and replaced AST (*R-AST*). The training process was conducted on the OpenMP directive classification dataset presented in Table 4, and the accuracy was calculated for each trained model on the validation set.

In Figure 5c the accuracy scores are presented as a function of the number of completed training epochs (passes over the full training set). Both versions of raw text representation obtained better results than their corresponding AST representations, inconsistent with findings for other code-related tasks mentioned in §3.2. Moreover, a slight advantage (of about 2%) is obtained by the simple text representation over the replaced text representation. We believe this can be explained by an implicit pattern present in parallelizable loops: they tend to have the same unique naming convention.

For example, iteration variables tend to be named i , j , k , and A , B , C , vec , arr as matrices and vectors. In addition, consider a file that contains multiple parallel worksharing-loop constructs; in such a case, the parallelized loops are likely operating on the same variables. Therefore, the model obtains better results by recognizing these unique names. Note that these results also appeared in the basic bag-of-words (BoW) setup (which will be described in §5.2), suggesting our hypothesis extends beyond the specific transformer architecture. Nevertheless, we hypothesize that AST representations may be more useful for models whose pre-training step includes the introduction of this syntax and leave this experiment to future work.

We provide the model’s objective loss over the training set and the validation set in Figure 5a and Figure 5b, respectively. This analysis assists us in determining the extent to which the model is *overfitting*—tuning its parameters to fit spurious relations in the training set, which do not generalize to the untuned-upon validation set. Since the validation loss curve converges after 5–6 epochs, we choose to use the models trained up to those points and report final accuracy scores of 84% for raw text (*Text*), 81% for replaced text (*R-Text*), 77% for AST (*AST*), and 71% for replaced AST (*R-AST*). As the *Text* representation of code produced the best accuracy for PRAGFORMER, we used it as our representation for all the subsequent experiments.

5.2 Parallel Work-sharing Loop Construct Classification

Baselines. In this experiment, we compare PRAGFORMER with two alternative methods on the task of classifying a parallel worksharing-loop construct: given a `for` loop, is it possible to add a parallel worksharing-loop construct? Our first comparison is a statistical trained Bag-of-Words (BoW) [56] model with a logistic regression classifier. In BoW, which has also been used in computer vision models [57, 58], individual tokens in the input text (code) are counted and populated in a count vector containing only non-negative integers. The order and structure of the text are not taken into account. The classifier learns a weight vector, where each entry corresponds to a word in the vocabulary, multiplied to produce a score for each instance, traversing over a training set using gradient descent, as is standard in ML.

The second system we evaluate against is the state-of-the-art S2S compiler named COMPAR [12]. COMPAR is the most suitable for this task, as it incorporates several S2S compilers under the hood — Par4All [14], AutoPar [59] and Cetus [60] — and combines their results to produce the best OpenMP directive. In practice, when we applied the system to our code segments, it was found that only Cetus managed to compile the examples successfully. While COMPAR generates the directives as a whole by executing the three S2S compilers and choosing the best directive, for our current experiment, we considered the outcome of whether or not it managed to insert an OpenMP directive into the code.

As the *Text* representation produced the best results for PRAGFORMER during the validation setup (§5.1), it was chosen as the representation for PRAGFORMER and BoW. We trained PRAGFORMER and the BoW model over the training set (Table 4). Being a deterministic compiler, COMPAR doesn’t need a training step; thus, it was executed on the test set directly. However, out of the 3,198 examples in the test set, COMPAR failed completely to compile 221 code instances due to complex structure

Test	Model	P	R	Acc
OPEN-OMP	Our	0.84	0.84	0.84
	BoW	0.71	0.71	0.71
	COMPAR	0.51	0.56	0.5
PolyBench	Our	0.95	0.95	0.95
	COMPAR	0.43	0.43	0.43
SPEC-OMP	Our	0.83	0.84	0.83
	COMPAR	0.76	0.75	0.75

(a) The results on the task of predicting parallelization potential of serial code (using OpenMP directive) on OPEN-OMP, PolyBench, and the SPEC-OMP suite.

Clause	Model	P	R	Acc
<i>private</i>	Our	0.86	0.85	0.85
	BoW	0.79	0.78	0.79
	COMPAR	0.56	0.51	0.56
<i>reduction</i>	Our	0.89	0.87	0.87
	BoW	0.78	0.78	0.78
	COMPAR	0.92	0.52	0.79

(b) The results on the task of identifying the need for a specific OpenMP clause – *private* and *reduction* — on OPEN-OMP.

Table 5: Comparison between PRAGFORMER (**our** in the table) and the competing systems on the tasks. (P=Precision, R=Recall, Acc=Accuracy)

definitions and operations unrecognized by its internal parser, emphasizing its limited robustness. We thus followed a fall-back strategy and consider these cases as having *negative* outcome. Moreover, we drop these 221 code examples from the test dataset for this experiment. Nevertheless, we observed that similar results in all three metrics are obtained by conducting the same experiment with the 2,977 examples for PRAGFORMER, BoW, and COMPAR.

Results. For evaluation, we report precision, recall, and accuracy on the test set as the performance measurements. These are standard metrics used by ML models for classification tasks. We present the performance of PRAGFORMER, BoW, and COMPAR on this classification task in Table 5a.

Result analysis. The best results are achieved by PRAGFORMER according to all measurements. We find the main reason behind COMPAR’s poor performance in detecting code examples that can be parallelized is the lack of association of functions, macros, and structure definitions and implementations in the code examples, a common situation for large code projects. In addition, we found that for loops with a low iteration count, Cetus didn’t insert an OpenMP directive (possibly a performance-oriented heuristic), although the examples did contain OpenMP directives. Furthermore, better results achieved by PRAGFORMER and BoW answers our research question in the affirmative.

By examining the incorrect predictions of PRAGFORMER, we found that more than 80% of the incorrect predictions occurred for code snippets with a length less than 20 lines of code. This might indicate that the length of the code does not necessarily affect the model’s decision. Moreover, The number of positive and negative examples for each specified length are approximately the same. However, the number of failed predictions for the negative examples was slightly more than for the positive ones. This indicates that not all of the negative examples are, in fact, negative and could possibly be wrongfully annotated by the developer.

5.3 Data-sharing Attribute Clause Classification

Similar to the parallel worksharing-loop construct classification task, we compared PRAGFORMER to BoW and COMPAR on the task of classifying the need of `private` and `reduction` clauses by testing them on the clause dataset. We again used the same metrics of precision, recall, F1 score, and accuracy for this experiment. We note that in a real-world application of the compared systems, directive classification will take place before clause identification, cascading errors from the previous step to this one, making our separated evaluation setup beneficial to the models which performed worse than PRAGFORMER in the directive task.

Results. We present the results of this experiment in Table 5b. COMPAR does not perform well on precision in predicting *private* clauses, mostly due to the default behavior of this clause: calling for its application to the iteration variable while applying the *shared* clause to the rest of the variables. As a result, most developers do not explicitly insert a *private(i)* statement, while COMPAR does. In addition, COMPAR also scores low on recall as that the model misses many cases where a *private* clause is possible.

On the *reduction* clause classification task, COMPAR manages to obtain a high score on the precision measurement as most of its positive predictions were correct. This indicates that the conservative, deterministic nature of COMPAR produces correct directives, *if* it manages to generate a directive at all. In contrast to the high precision score, and similar to its recall performance in the *private* task, COMPAR scores low on recall, as the model misses many cases where a *reduction* clause is possible. In contrast, PRAGFORMER produces both higher recall and precision scores for both the clauses, demonstrating a good balance between finding many true cases, without allowing many false predictions.

5.4 Benchmarks

In order to test the generality of PRAGFORMER, we apply it to two standard OpenMP benchmarks that do not appear in OPEN-OMP: PolyBench [61] and the Standard Performance Evaluation Corporation (SPEC) [62, 63]. In contrast to the evaluation over the test set, evaluating over OpenMP benchmarks provides an evaluation of the ability to predict best performance label, as these benchmarks contain the best performance directives.

Benchmark descriptions. The PolyBench test suite contains a collection of 30 compute-intensive benchmarks and is used, among other applications, as a shared-memory environment benchmark [64, 65]. PolyBench [64] contains 64 snippets of code with parallel worksharing-loop constructs and 83 without. SPEC contains several benchmarks designed to evaluate the performance of shared-memory architectures. Among the SPEC-ACCEL and SPEC-OMP benchmarks in SPEC, we used later for our evaluation; we opted not to evaluate SPEC-ACCEL as it shares many similarities with the NAS parallel benchmark, which is already included in OPEN-OMP. SPEC-OMP contains 113 snippets of code with OpenMP worksharing-loop construct and 174 without ⁸. However, the unique syntax of this benchmark produces 230 OOVs

⁸Sources: https://github.com/pragformer/PragFormer/blob/main/DB_TEST.tar.gz

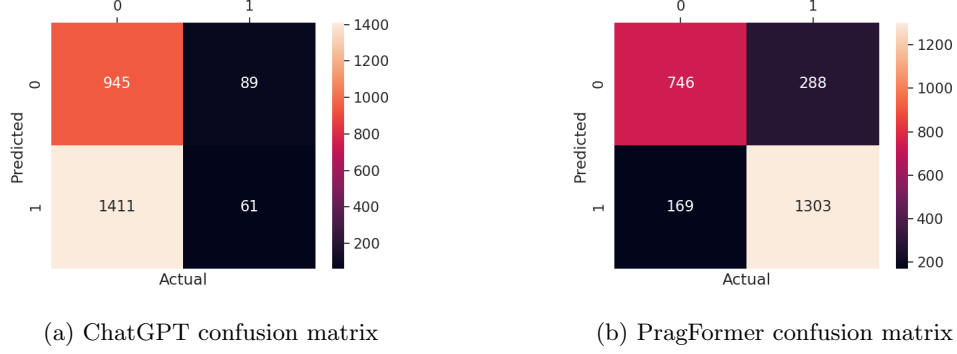


Fig. 6: Comparison of confusion matrix

tokens with respect to the training data, which is five times relatively higher than the number of OOVs in the test set. As mentioned, a high number of OOVs might produce poor results. Therefore, we added approximately 25% of the code snippets from SPEC-OMP to the training set, thus reducing the number of OOVs to 60, which is the same number of OOVs as PolyBench.

Results. The lower rows in Table 5a present the results of PRAGFORMER and COMPAR on PolyBench and SPEC-OMP. The results of PRAGFORMER are comparable to, and even slightly better than, the those over the OPEN-OMP test set. However, COMPAR failed to parse 287 snippets from the SPEC-OMP benchmark, mainly due to unrecognized keywords, such as *register*. Thus, it is excluded from COMPAR’s results. These results highlight the fact that PRAGFORMER can indeed generalize beyond training sets by learning the rules for parallelizing serial **for** loops without explicit programming.

5.5 Comparison with ChatGPT

As general-purpose large language models (LLMs) are becoming increasingly popular, we compared PRAGFORMER with ChatGPT. To perform the comparison, we utilized a subset of our test dataset consisting of approximately 2,500 examples to query ChatGPT and ask if it can be parallelized. Specifically, for every example, we provided the prompt “*generate the optimal OpenMP pragma if possible*” to ChatGPT. As shown in Figure 6, ChatGPT’s predictions were overly optimistic, frequently suggesting parallelization when it was not possible. PRAGFORMER outperformed ChatGPT on both precision and accuracy.

5.6 Explaining PragFormer’s Predictions

Explaining and understanding the reasons behind model’s predictions is a difficult task, especially for source codes [66]. The field of explainable AI (XAI) is evolving to address the exact same requirement. Nonetheless, many algorithms, such as LIME [67], attempt to give an explanation or intuition for classifiers’ decisions [68]. LIME studies the connections between keywords (tokens) of an input and the change in the

prediction of a model once they are removed. Finally, LIME presents the probability that a given keyword has affected the prediction. Towards that end, we applied LIME on specific examples and their outcomes predicted by PRAGFORMER to analyze which keywords and statements are deemed important by PRAGFORMER. However, for lack of space, we present four representative examples from the benchmarks in Appendix A.

6 Conclusion & Future Work

Automatic parallelization of serial code is a well-researched problem in the fields of high-performance computing and parallel programming. A number of well-known, deterministic approaches and implementations already exist, including in tools such as compilers. Our evaluation of existing tools, however, revealed that they suffer from various limitations, some of which restrict their applicability as a programming-assistance tool in code editors. Such editor-based programming-assistance tools have been gaining popularity and adoption, thanks to combination of recent advancements in the field of deep-learning for source code processing. For the auto-parallelization problem, we observed that no programming assistance tool for auto-parallelization exists. Towards that end, this paper presented a novel data-driven model, named PRAGFORMER, that learns to predict the possibility of parallelizing a loop (as a parallel worksharing-loop construct) and applying OpenMP data-sharing clauses. As a part of this efforts, we have also created a corpus of OpenMP code snippets which is, as far as we know, the first of its nature. We trained variants of PRAGFORMER model with different representations of code to find that representing the source code as raw text obtains the best results. More importantly, we found that PRAGFORMER manages to outperform two competing approaches (a state-of-the-art deterministic S2S compiler and a lightweight text-aware ML model) on tasks related to auto-parallelization. With these results, we believe that PRAGFORMER can indeed be deployed as a programming-assistant tool (which we will implement soon) and that the programmers would find it useful. Moreover, in the spirit of contributing to advancement, we have released our source code as well as code corpus publicly (links withheld for anonymity). We invite the community to use our corpus for analyzing parallelization use cases and as a development bed for ML models that can learn to parallelize code.

Although, PRAGFORMER outperformed competing tools on our tasks, we believe there are several avenues for future exploration. In most cases, the context of the loop itself is enough to generate an OpenMP directive. However, in order to push performance further, it is effective to include *context* of code. Thus, for future work, we will expand the database to contain these extra lines of code. In addition, in order to improve the effectiveness of PRAGFORMER, we plan to explore additional code representations such as the structured-based traversal (SBT) representation [43, 46], the combination of text and AST, *IR2vec*, *Programl* and more. Furthermore, explaining the model’s decision with additional explainability methods is crucial in creating a code advisor [66, 69]. We also plan to expand the corpus by searching other keywords on GitHub as well as including code snippets from C++ and Fortran, in addition to C. Most importantly, PRAGFORMER provides a crucial step towards a more robust framework based on NLP techniques that will eventually be able to insert

OpenMP directives automatically with high fidelity, contributing to make the (serial) code more efficient.

Acknowledgement

This research was supported by the Israeli Council for Higher Education (CHE) via the Data Science Research Center, Ben-Gurion University of the Negev, Israel; Intel Corporation (oneAPI CoE program); and the Lynn and William Frankel Center for Computer Science. Computational support was provided by Intel Developer Cloud [70], and the NegevHPC project [71].

References

- [1] Harel, R., Pinter, Y., Oren, G.: Learning to parallelize in a shared-memory environment with transformers. In: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (2022). IEEE
- [2] Harel, R., Pinter, Y., Oren, G.: Learning to parallelize in a shared-memory environment with transformers. In: Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, pp. 450–452 (2023)
- [3] Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
- [4] Unknown: Automatic Parallelism and Data Dependency. <https://web.archive.org/web/20140714111836/http://blitzprog.org/posts/automatic-parallelism-and-data-dependency>. [Online]
- [5] Prema, S., *et al.*: A study on popular auto-parallelization frameworks. Concurrency and Computation: Practice and Experience **31**(17), 5168 (2019)
- [6] Prema, S., Jehadeesan, R.: Analysis of parallelization techniques and tools. Int’l J. Information Computation Tech **3**(5), 471–478 (2013)
- [7] Project, G.: Automatic Parallelization in GCC. <https://gcc.gnu.org/wiki/AutoParInGCC> (2012)
- [8] Grosser, T., *et al.*: Polly-polyhedral optimization in llvm. In: Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT), vol. 2011, p. 1 (2011)
- [9] Lattner, C.: Llm and clang: Next generation compiler technology. In: The BSD Conference, vol. 5 (2008)
- [10] Foundation, L.: Polly - A Loop Optimizer in LLVM. <https://polly.llvm.org/docs/Architecture.html> (2023)

- [11] corp, I.: Automatic Parallelization with Intel Compilers. <https://www.intel.com/content/www/us/en/developer/articles/technical/automatic-parallelization-with-intel-compilers.html> (2018)
- [12] Mosseri, I., *et al.*: Compar: Optimized multi-compiler for automatic openmp S2S parallelization. In: Milfeld, K.F., Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22-24, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12295, pp. 247–262. Springer, ??? (2020). https://doi.org/10.1007/978-3-030-58144-2_16 . https://doi.org/10.1007/978-3-030-58144-2_16
- [13] Amini, M., *et al.*: Par4all: From convex array regions to heterogeneous computing. In: IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012 (2012)
- [14] Team, T.P.: Par4All homepage. <http://par4all.github.io/>. [Online]
- [15] Bae, H., Mustafa, D., Lee, J.-W., Lin, H., Dave, C., Eigenmann, R., Midkiff, S.P.: The cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming* **41**, 753–767 (2013)
- [16] Harel, R., *et al.*: Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential. *International Journal of Parallel Programming* **48**(1), 1–31 (2020)
- [17] Milewicz, R., Pirkelbauer, P., Soundararajan, P., Ahmed, H., Skjellum, T.: Negative perceptions about the applicability of source-to-source compilers in hpc: A literature review. In: International Conference on High Performance Computing, pp. 233–246 (2021). Springer
- [18] Prema, S., Jehadeesan, R., Panigrahi, B.: Identifying pitfalls in automatic parallelization of nas parallel benchmarks. In: Parallel Computing Technologies (PARCOMPTECH), 2017 National Conference On, pp. 1–6 (2017). IEEE
- [19] Project, G.: GNU Offloading and Multi-Processing Project (GOMP). <https://gcc.gnu.org/projects/gomp/#omp5.0> (2023)
- [20] Hindle, A., *et al.*: On the naturalness of software. *Communications of the ACM* **59**(5), 122–131 (2016)
- [21] Sharma, T., *et al.*: A survey on machine learning techniques for source code analysis. arXiv preprint arXiv:2110.09610 (2021)
- [22] Google: ML-Enhanced Code Completion Improves Developer Productivity. <https://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves>.

[html](#). [Online]

- [23] GitHub: GitHub CoPilot. <https://github.com/features/copilot>. [Online]
- [24] Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback. In: III, H.D., Singh, A. (eds.) Proceedings of the 37th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 119, pp. 10799–10808. PMLR, ??? (2020). <https://proceedings.mlr.press/v119/yasunaga20a.html>
- [25] Hasabnis, N., Gottschlich, J.: ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures. In: Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming. MAPS 2021, pp. 32–42. Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3460945.3464954> . <https://doi.org/10.1145/3460945.3464954>
- [26] Dave, C., et al.: Cetus: A source-to-source compiler infrastructure for multicores. Computer **42**(12) (2009)
- [27] Neamtiu, I., *et al.*: Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Software Engineering Notes **30**(4), 1–5 (2005)
- [28] Fagin, R., Vardi, M.Y.: The Theory of Data Dependencies: a Survey. IBM Thomas J. Watson Research Division, ??? (1984)
- [29] Chen, M., Tworek, J., Jun, H., Yuan, Q., Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code. CoRR **abs/2107.03374** (2021) [2107.03374](#)
- [30] OpenAI: OpenAI ChatGPT. <https://openai.com/blog/chatgpt>. [Online]
- [31] OpenAI: OpenAI Codex. <https://openai.com/blog/openai-codex>. [Online]
- [32] Stiennon, N., Ouyang, L., Wu, J., Ziegler, D., Lowe, R., Voss, C., Radford, A., Amodei, D., Christiano, P.F.: Learning to summarize with human feedback. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems (2020). https://proceedings.neurips.cc/paper_files/paper/2020/file/1f89885d556929e98d3ef9b86448f951-Paper.pdf

- [33] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS’17 (2017)
- [34] Oren, G., et al.: AutOMP: An Automatic OpenMP Parallelization Generator for Variable-Oriented High-Performance Scientific Codes. arXiv preprint arXiv:1707.07137 (2017)
- [35] Yang, G., et al.: DeepSCC: Source Code Classification Based on Fine-Tuned RoBERTa. CoRR **abs/2110.00914** (2021) [2110.00914](#)
- [36] Zoph, B., *et al.*: Transfer learning for low-resource neural machine translation. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pp. 1568–1575. Association for Computational Linguistics, Austin, Texas (2016). <https://doi.org/10.18653/v1/D16-1163> . <https://aclanthology.org/D16-1163>
- [37] Han, X., Eisenstein, J.: Unsupervised domain adaptation of contextualized embeddings for sequence labeling. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pp. 4238–4248. Association for Computational Linguistics, Hong Kong, China (2019). <https://doi.org/10.18653/v1/D19-1433> . <https://aclanthology.org/D19-1433>
- [38] Imankulova, A., *et al.*: Exploiting out-of-domain parallel data through multilingual transfer learning for low-resource neural machine translation. In: Proceedings of Machine Translation Summit XVII: Research Track, pp. 128–139. European Association for Machine Translation, Dublin, Ireland (2019). <https://aclanthology.org/W19-6613>
- [39] Liu, Y., et al.: Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 (2019)
- [40] Taylor, W.L.: “cloze procedure”: A new tool for measuring readability. Journalism quarterly **30**(4), 415–433 (1953)
- [41] Sun, Z., *et al.*: A grammar-based structural cnn decoder for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 7055–7062 (2019)
- [42] Jayasundara, V., Bui, N.D.Q., Jiang, L., Lo, D.: TreeCaps: Tree-Structured Capsule Networks for Program Source Code Processing. arXiv preprint arXiv:1910.12306 (2019)
- [43] Hu, X., *et al.*: Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), pp. 200–20010 (2018).

- [44] Li, K., *et al.*: An empirical study of transformer-based neural language model adaptation. In: ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 7934–7938 (2020). IEEE
- [45] Hu, X., Li, *et al.*: Deep code comment generation with hybrid lexical and syntactical information. Empirical Software Engineering **25**(3), 2179–2217 (2020)
- [46] Li, Z., *et al.*: SeTransformer: A Transformer-Based Code Semantic Parser for Code Comment Generation. IEEE Transactions on Reliability (2022)
- [47] Church, K.W.: Word2vec. Natural Language Engineering **23**(1), 155–162 (2017)
- [48] VenkataKeerthy, S., *et al.*: Ir2vec: Llm ir based scalable program embeddings. ACM Transactions on Architecture and Code Optimization (TACO) **17**(4), 1–27 (2020)
- [49] Cummins, C., *et al.*: Programl: Graph-based deep learning for program optimization and analysis. arXiv preprint arXiv:2003.10536 (2020)
- [50] Brill, E.: Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. Computational Linguistics **21**(4), 543–565 (1995)
- [51] Hellendoorn, V.J., Devanbu, P.: Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 763–773 (2017)
- [52] LeClair, A., *et al.*: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195 (2020)
- [53] Srivastava, N., *et al.*: Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research **15**(1), 1929–1958 (2014)
- [54] Loshchilov, I., Hutter, F.: Fixing weight decay regularization in adam (2018)
- [55] Bendersky, E., *et al.*: Pycparser (2010)
- [56] Zhang, Y., *et al.*: Understanding bag-of-words model: a statistical framework. International Journal of Machine Learning and Cybernetics **1**(1), 43–52 (2010)
- [57] Law, M.T., *et al.*: Bag-of-words image representation: Key ideas and further insight. In: Fusion in Computer Vision, pp. 29–52. Springer, ??? (2014)
- [58] Sikka, K., *et al.*: Exploring bag of words architectures in the facial expression domain. In: European Conference on Computer Vision, pp. 250–259 (2012).

- [59] Quinlan, D.: Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters* **10**(02n03), 215–226 (2000)
- [60] Lee, S.-I., *et al.*: Cetus—an extensible compiler infrastructure for source-to-source transformation. In: *International Workshop on Languages and Compilers for Parallel Computing*, pp. 539–553 (2003). Springer
- [61] Yuki, T.: Understanding polybench/c 3.2 kernels. In: *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, pp. 1–5 (2014)
- [62] Dixit, K.M.: The spec benchmarks. *Parallel computing* **17**(10-11), 1195–1209 (1991)
- [63] Juckeland, G., *et al.*: SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. In: *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pp. 46–67 (2014). Springer
- [64] Grauer-Gray, S., *et al.*: Auto-tuning a high-level language targeted to gpu codes. In: *2012 Innovative Parallel Computing (InPar)*, pp. 1–10 (2012). Ieee
- [65] Pouchet, L.-N., *et al.*: Polybench: The polyhedral benchmark suite. URL: <http://www.cs.ucla.edu/pouchet/software/polybench> (2012)
- [66] Gottschlich, J., Solar-Lezama, A., Tatbul, N., Carbin, M., Rinard, M., Barzilay, R., Amarasinghe, S., Tenenbaum, J.B., Mattson, T.: The three pillars of machine programming. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 69–80 (2018)
- [67] Ribeiro, M.T., *et al.*: "why should I trust you?": Explaining the predictions of any classifier. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pp. 1135–1144 (2016)
- [68] Adadi, A., *et al.*: Peeking inside the black-box: a survey on explainable artificial intelligence (xai). *IEEE access* **6**, 52138–52160 (2018)
- [69] Chefer, H., *et al.*: Transformer interpretability beyond attention visualization. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 782–791 (2021)
- [70] Intel: Intel Developer Cloud. <https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html>. [Online] (2023)
- [71] NegevHPC Project. <https://www.negevnhpc.com>. [Online]

Example	Original Directive	PRAGFORMER Prediction
<pre> 1 for (i = 0; i < POLYBENCH_LOOP_BOUND(4000, n); i++) 2 for (j = 0; j < POLYBENCH_LOOP_BOUND(4000, n); j++) 3 x1[i] = x1[i] + (A[i][j] * y_1[j]); </pre>	<pre> #pragma omp \parallel for \parallel private(j) </pre>	With OpenMP
<pre> 1 for (i = 0; i < n; i++) { 2 fprintf(stderr, "%0.21f ", x[i]); 3 if ((i % 20) == 0) 4 fprintf(stderr, "\n");} </pre>	Without OpenMP	Without OpenMP
<pre> 1 for (i = 0; i < ((ssize_t) image->colors); i++) 2 image->colormap[i].opacity = (IndexPacket) i; </pre>	<pre> #pragma omp \parallel for \parallel schedule(dynamic,4) </pre>	Without OpenMP
<pre> 1 for (i = 0; i < maxgrid; i++) 2 for (j = 0; j < maxgrid; j++){ 3 sum_tang[i][j] = (int) ((i + 1) * (j + 1)); 4 mean[i][j] = (((int) i) - j) / maxgrid; 5 path[i][j] = (((int) i) * (j - 1)) / maxgrid; } </pre>	Without OpenMP	With OpenMP

Table A1: Classification examples and PRAGFORMER prediction.

Appendix A Explainability with LIME

In Table A1, we present representative examples, obtained from our benchmarks, and the corresponding predictions by PRAGFORMER. To gain an intuition for the predicted outcome of PRAGFORMER, we applied LIME tool on those examples. The output of LIME for these examples is presented in Figure A1.

The first example presents a code snippet from PolyBench, in which PRAGFORMER correctly identified the possibility of parallelizing it (with an OpenMP directive). In this example, LIME pinpoints the variables `j`, `POLYBENCH_LOOP_BOUND`, `A` and `y_1` as the main contributors to the decision of the model. This indicates that PRAGFORMER focuses on the loop index variables and the arrays, as it should.

The second example presents a snippet taken from PolyBench and contains an I/O operation, thus, there is no OpenMP directive. LIME identifies the keyword `fprintf` and `stderr` as the reason for model’s classification as the snippet cannot be parallelized. We removed these two keywords to verify this observation, and PRAGFORMER predicts that the example can be parallelized with an OpenMP directive. This indicates that PRAGFORMER understands that the presence of these specific keywords indicates that there cannot be an OpenMP directive.

The third example, taken from SPEC-OMP, contains an OpenMP directive that PRAGFORMER fails to predict correctly. By observing LIME’s result, the two variables, `ssize_t` and `IndexPacket`, are the main reason to its incorrect prediction. After removing both the variables, PRAGFORMER predicts the possibility of an OpenMP directive. This might be due to model’s unfamiliarity with these keywords and how they affect the code.

The fourth example, taken from PolyBench, contains an assignment of three arrays. While PRAGFORMER predicts (correctly) that there should be an OpenMP directive, the example does not. As with the first example, LIME pinpoints the loop index

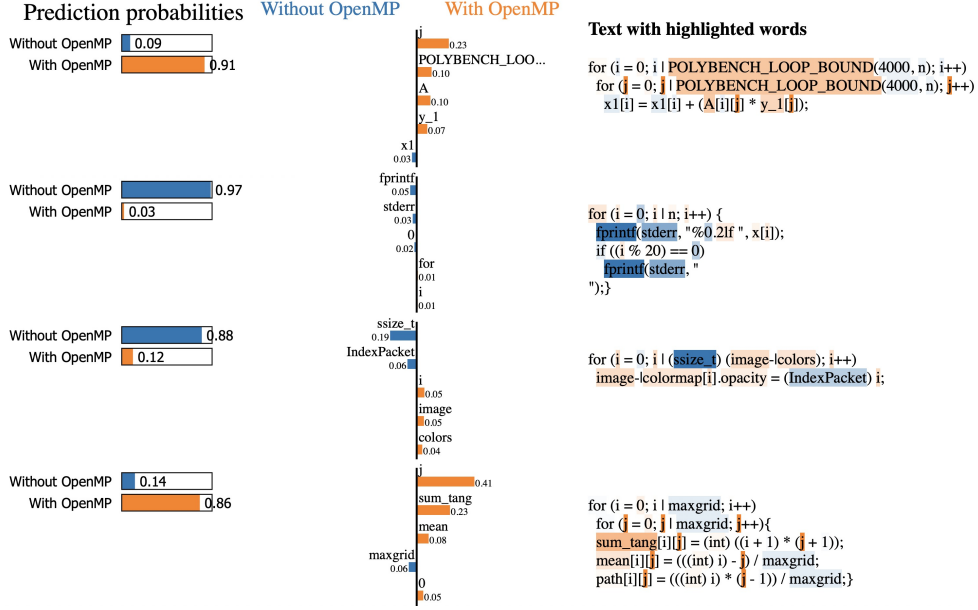


Fig. A1: LIME's output on the four examples in Table A1. From left to right, LIME presents the label distributions given to the instance by the model, followed by the most influential tokens and their weighted importance on the decision made, next highlighting their location in the input text.

variable j and the arrays as the reason for its prediction. This further indicates that the model focuses on the correct variables for its predictions.

In summary, we observe that PRAGFORMER focuses on the core variables and statements to determine if and how the code can be parallelized. However, there are still cases in which it fails to predict correctly, possibly due to unfamiliar keywords or statements.