OMPar: Automatic Parallelization with AI-Driven Source-to-Source Compilation

Tal Kadosh
Ben-Gurion University, IAEC
Israel
talkad@post.bgu.ac.il

Niranjan Hasabnis
Code Metal
United States
niranjan@codemetal.ai

Prema Soundararajan
University of Alabama at Birmingham
United States
prema@uab.edu

Vy A. Vo
Intel Labs
United States
vy.vo@intel.com

Mihai Capotă
Intel Labs
United States
mihai.capota@intel.com

Nesreen K. Ahmed
Cisco
United States
nesahmed@cisco.com

Yuval Pinter
Ben-Gurion University
Israel
pintery@bgu.ac.il

Gal Oren
Stanford University, Technion
United States
galoren@stanford.edu

Abstract

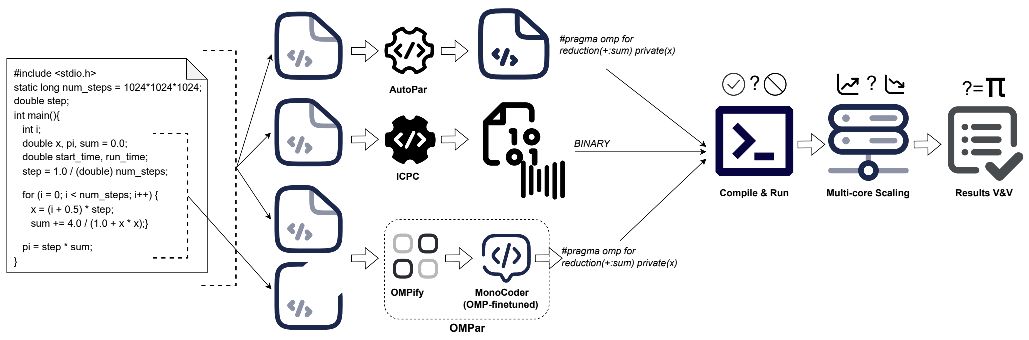
Existing automatic code parallelization tools are either too conservative (formal-methods based tools) or are too inaccurate (AI-based tools). This paper introduces OMPar, an AI-driven tool that breaks the problem into two sub-problems of parallelism detection and parallel pragma generation and then integrates two state-of-the-art models to solve the problem. We evaluate OMPar and competing existing tools in terms of accuracy (in suggesting correct pragma), syntax, semantics, and run-time performance of suggested pragmas. Overall, we found that OMPar outperforms existing tools in accurately suggesting parallelization pragmas. Moreover, we found that OMPar-suggested pragmas are also syntactically- and semantically valid (high compilation and test success rate), and they also deliver performance improvement over corresponding baseline serial programs. The sources of this work are available at our OMPar repository.

1 Introduction

Automatic parallelization has been an active area of research in the parallel programming community to eliminate/reduce the need of manual parallelization. Automatic parallelization approaches can be broadly classified into two types: (1) formal approaches of source-to-source transformation tools or auto-parallelizing compilers, and (2) AI-based approaches.

Formal approaches for automatic parallelization use deterministic methods (such as syntax-driven translation rules etc). These tools translate the program's source code into a parallelized version, typically by inserting parallelization directives[1] or annotations (e.g., OpenMP pragmas [27]) into the code. Our recent evaluation of AutoPar, a state-of-the-art source-to-source auto-parallelization tool, revealed several limitations (covered in details in Appendix A.1.1), such as incorrect parallelization that changed the program semantics, missing the parallelization opportunity altogether, etc.

[1]OpenMP pragma annotations on `for` loops allow such loops to improve performance by using multi-threading to process computations inside the loop concurrently.



```
#include <stdio.h>
static long num_steps = 1024*1024*1024;
double step;
int main(){
  int i;
  double x, pi, sum = 0.0;
  double start_time, run_time;
  step = 1.0 / (double) num_steps;

  for (i = 0; i < num_steps; i++) {
    x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);}

  pi = step * sum;
}
```

```
#pragma omp for
reduction(+:sum) private(x)
```

Figure 1: Overview of end-to-end workflow with OMPAR and other competing tools

Recent breakthroughs in AI, such as large language models (LLMs) for natural language processing (NLP) and code generation, have led to considerable interest in developing specific AI models for automatic code parallelization [5, 20, 26]. Yet when Nichols et al. recently evaluated popular LLMs such as GPT-4 and LLaMa-2 with their ParEval benchmark [30], they found that "*LLMs are significantly worse at generating parallel code than they are at generating serial code*". They further comment that "*the poor performance of LLMs on ParEval indicates that further efforts are necessary to improve the ability of LLMs to model parallel code and/or create new LLMs that are specialized for parallel code generation.*"

Our paper proposes a novel automatic parallelization approach, named OMPAR, that employs a modular approach by integrating multiple state-of-the-art AI models in the parallelization pipeline, each designed for a specific task – OMPIFY for assessing the parallelization potential of loops and MONOCODER-*OMP* for generating precise OpenMP pragmas. This modular approach allows for greater flexibility (such as fine-tuning them) and accuracy compared to monolithic LLMs that attempt to handle all aspects of code generation within a single model and can output a wide range of responses. In fact, MONOCODER-*OMP* is a domain-specific (HPC-specific) model that precisely exploits this capability by leveraging C/C++ serial and parallel programs for fine-tuning, thereby improving its performance on the task of parallelization and pragma generation.

Contributions. ① We propose OMPAR, a novel approach that combines two state-of-the-art AI models: OMPIFY [22] for loop parallelization determination and MONOCODER [21] (fine-tuned on OpenMP codes, MONOCODER-*OMP*) for pragma generation. ② We evaluate the accuracy and performance of OMPAR using ground-truth labels and compile-and-run checks against the HeCBench benchmark, and comparing it against existing automatic parallelization compilers, namely ICPC and AutoPar. ③ Our results show that OMPAR achieves superior performance compared to existing automatic parallelization compilers, even when provided with partial codes, highlighting the robustness and scalability of our approach.

2 OMPAR: LLM-based source-to-source parallelization compiler

Task objective. While a lot of progress has been made in integrating machine learning into the identification of parallelism opportunities in C/C++ programs [4, 20, 22, 43], there has been a notable gap in *connecting parallelism detection with code generation*. Our work bridges this gap by leveraging OMPIFY for parallelism detection and MONOCODER for parallel code generation. Specifically, we fine-tune MONOCODER on *HPCorpus$_{OMP}$* and call it MONOCODER-*OMP* (see Appendix B for fine-tuning details.)

OMPAR *pipeline design.* Figure 1 shows the high-level overview of our pipeline with OMPAR. OMPAR first uses OMPIFY to determine whether the loop should have a pragma. If the pragma is not applicable, OMPAR will return this result. If the pragma is deemed necessary, MONOCODER is then used to generate the complete pragma for the given `for` loop.

We chose OMPIFY and MONOCODER-*OMP* out of comparative models from Table 4 as they are ① SOTA models with excellent performance for their tasks with C and C++ languages, ② trained on the same large-scale parallel-computing-oriented dataset, and ③ designed as targeted small domain-specific language models, which can be deployed even on average computer systems.

3 Experimental evaluation

Our experimental evaluation is designed to answer the following research questions:

- **RQ1:** How accurately can OMPAR predict correct OpenMP pragma for a given `for` loop?
- **RQ2:** Do programs with OMPAR-suggested pragmas compile and run correctly? If so, what is their runtime performance and scalability in a multi-core, shared-memory environment?

The first research question evaluates OMPAR as an AI model; the second research question evaluates its ability to suggest syntactically-correct and high-performance pragmas.

3.1 Experimental setup

HeCBench benchmark. HeCBench is a novel heterogeneous computing benchmark suite with over 350 benchmarks, written in CUDA, SYCL, HIP, and OpenMP, spanning diverse domains, including machine learning, image processing, etc [19]. Moreover, all the OpenMP benchmarks from HeCBench follow a uniform compilation recipe[2] and also contain test inputs, reference output, and a check that compares reference output with the generated output.

Preprocessing. We first checked that HeCBench was not part of the pre-training or fine-tuning dataset of MONOCODER. Next, we performed a basic sanity check of OpenMP benchmarks by attempting to compile them, run them with a timeout of 10 minutes (to choose a reasonable time on our test system), and check that the benchmarks passed the output correctness check. We found that 223 OpenMP benchmarks passed all the tests.

Test dataset generation. We then extracted all the `for` loops from those 223 benchmarks that had some OpenMP pragma[3]. In all, the dataset contained 385 `for` loops having an OpenMP pragma. We balanced the dataset in terms of "negative" `for` loops — loops that do not contain any OpenMP pragma — by randomly choosing 385 such `for` loops from HeCBench's OpenMP benchmarks. In summary, our dataset for the evaluation contained 770 `for` loops. In terms of OpenMP benchmarks, it covered 175 of 223 benchmarks.

Baselines. We use AutoPar [24] from ROSE-0.11.46.0.1, a static source-to-source-based automatic parallelization tool, and ICPC, Intel's auto-parallelizing compiler from Intel(R) oneAPI DPC++/C++ Compiler 2022.0.0, as baselines for this experiment. Specifically, because both AutoPar and ICPC operate on complete programs (as opposed to a single `for` loop), we generate serial versions of HeCBench's OpenMP benchmarks by removing all OpenMP pragmas from them[4]. We ran compiled benchmarks on a dual-socket, 80-core, Intel Xeon Platinum 8380 CPU, running at 2.30GHz with hyper-threading enabled.

Evaluation methodology. Our evaluation methodology is broken down into two sub-steps (Figure 1, right): ① *Accuracy test.* This test is designed to answer **RQ1**. In this test, we feed each loop out of 770 `for` loops to all three tools separately and compare the pragma generated by them or lack thereof with the ground-truth pragma (i.e., label). We use standard machine learning metrics (precision, recall, accuracy) to report the performance of every tool on this test. ② *Compile & run, scale and validation test.* This test is designed to answer **RQ2**. If the tool that we are evaluating generates some OpenMP pragma for the input `for` loop, then we insert that pragma in the OpenMP benchmark, which contains that loop. In other words, we replace the original pragma of the loop with the inferred pragma. Then, we attempt to compile that benchmark, execute it with the test input (that comes along with the benchmark), and compare the expected benchmark output with the actual output. We also measure the execution runtime for different values of `OMP_NUM_THREADS`, the environment variable that controls the number of threads in a *parallel for* region. As a baseline for the execution runtime comparison, we run HeCBench OpenMP benchmarks in their default setting.

3.2 Results

*Results on accuracy test (**RQ1**).* Overall, we found that OMPAR was reporting a decent accuracy of 74% on the HecBench loops dataset (the first row of the Table 1). After a careful analysis, we found

[2]Although HeCBench is a benchmark suite for heterogeneous environment, its OpenMP benchmarks could be compiled for CPU-only environment using `make DEVICE=cpu`.

[3]As OMPAR does not support offloading-specific OpenMP pragmas such as `#pragma omp target data`, we did not consider them for our selection.

[4]For the pragma erasing, we used AutoParBench's PragmaRemover tool: https://github.com/LLNL/AutoParBench/tree/master/tools/PragmaRemover

Test setup	TP	FP	TN	FN	Precision	Recall	Accuracy
OMPar accuracy with ground-truth label	311	127	262	70	71%	**81%**	**74%**
AutoPar accuracy with ground-truth label	63	25	365	317	71%	17%	56%
ICPC accuracy with ground-truth label	95	11	285	379	**90%**	25%	62%
OMPar accuracy with compile and run check	407	31	262	70	92%	**85%**	**86%**
AutoPar accuracy with compile and run check	24	25	365	356	49%	6%	50%
ICPC accuracy with compile and run check	68	5	312	385	**93%**	15%	49%

Table 1: Result of accuracy test on HeCBench (The best scores are in bold).

Threads	C&R pass	Compile fail	Run fail	Timeout
1	706 (91.68%)	53	0	11
4	710 (92.2%)	53	7	0
8	710 (92.2%)	53	7	0
16	707 (91.81%)	53	10	0

Table 2: Compile and run (C&R) test results for OMPar on 770 loops from HeCBench.

Tool	Pass	Failed
OMPar	**130**	**45**
AutoPar	42	133
ICPC	72	17

Table 3: Compile and run test results for 175 OpenMP benchmarks from HeCBench (The best scores are in bold).

that OMPar was predicting the possibility of parallelizing several `for` loops that originally had no OpenMP pragma (i.e., "negative" for loops). To confirm if OMPar suggested loops can indeed be parallelized (using the pragma it was suggesting), we decided to subject all such 438 loops (i.e., 311 + 127) to compile and run the test. To our surprise, we found that 96 of 127 false positives were indeed not false positives - meaning 96 loops that did not have any OpenMP pragma as a ground truth could, in fact, be parallelized using OMPar-suggested OpenMP pragma and they also passed the compile and run check. The remaining 31 false positives were indeed loops that could not be parallelized. In this particular case, we consider compile and run tests to have higher credibility over the original loop label to determine the potential of parallelizing a loop; a loop may not be parallelized for several reasons such as the programmer missing out on the opportunity, potential performance improvement with the parallelization, etc. *Hence, in conclusion,* OMPar *achieves 86% accuracy, 92% precision, and 85% recall on the accuracy test.*

We then evaluated AutoPar as well as ICPC using the same methodology. Both of them achieve high precision (up to 93%) but low recall (25% or lower). *Higher precision values of ICPC point to the conservative nature of ICPC, while its low recall rate suggests a high number of false negatives, indicating missed parallelization opportunities. AutoPar shows a similar behavior trait as ICPC, though its precision and recall scores are not as good as ICPC.* We have a detailed analysis of AutoPar and ICPC in Appendix E.

*Results on compile and run test (**RQ2**).* We performed the compile and run test on all 770 loops from OMPar's accuracy test results and found that 717 of those passed the compilation test, while 706-710 of those 717 could pass the output verification test for different settings of `OMP_NUM_THREADS`. Table 2 shows the detailed results. *In summary, for* OMPar*, around 92% of the 770 loops successfully passed the compilation as well as the output verification test.*

In order to account for different operating modes of OMPar and AutoPar (i.e., parallelizing a loop individually vs parallelizing the whole program), we also report the performance of both at the level of OpenMP benchmarks (as opposed to individual loops) in HeCBench. Specifically, out of a total of 175 OMP benchmarks covered by 770 loops, OMPar could successfully parallelize all the loops from 130 benchmarks that also passed compile and run test (output verification). To be precise, loops belonging to the remaining 45 benchmarks either could not be parallelized by OMPar or the parallelized loops failed compilation/run check. AutoPar, on the other hand, could parallelize a total of 109 benchmarks, with 63 of those passing the compilation test and 42 passing the output verification test. Table 3 shows the overall results for 175 OpenMP benchmarks from HeCBench.

Results on scale test. For scale test, we categorize the performance of every benchmark over baseline into improvement (shown in green) or degradation (shown in red) and also different magnitudes (1x-2x, 2x-5x, 5x-10x, and >10x), as detailed in Figure 3 in Appendix C. Furthermore, since these tools are parallelizing different number of benchmarks (as observed in Figure 3a), we calculate percentages of the benchmarks belonging to these 8 categories to allow easier comparison. Overall, the results show that ① the percentage of benchmarks degrading over baseline is considerably less than those showing the improvement and ② *although AutoPar is able to achieve >10x improvement for more % of benchmarks than* OMPar, OMPar *is showing similar performance, even though it is parallelizing almost 3X more benchmarks than AutoPar.*

Acknowledgments

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); Pazy Foundation; and the Lynn and William Frankel Center for Computer Science. Computational support was provided by HPE HPC & AI Cloud, Intel Developer Cloud, and the NegevHPC project.

References

[1] OpenMP Compilers and Tools. `https://www.openmp.org/resources/openmp-compilers-tools/`. [Online].

[2] Mehdi Amini et al. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.

[3] Le Chen, Nesreen K Ahmed, Akash Dutta, Arijit Bhattacharjee, Sixing Yu, Quazi Ishtiaque Mahmud, Waqwoya Abebe, Hung Phan, Aishwarya Sarkar, Branden Butler, et al. Position paper: The landscape and challenges of hpc research and llms. *arXiv preprint arXiv:2402.02018*, 2024.

[4] Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. Ompgpt: A generative pre-trained transformer model for openmp, 2024.

[5] Le Chen, Pei-Hung Lin, Tristan Vanderbruggen, Chunhua Liao, Murali Emani, and Bronis De Supinski. Lm4hpc: Towards effective language model application in high-performance computing. In *International Workshop on OpenMP*, pages 18–33. Springer, 2023.

[6] Le Chen, Quazi Ishtiaque Mahmud, Hung Phan, Nesreen Ahmed, and Ali Jannesari. Learning to parallelize with openmp by augmented heterogeneous ast representation. *Proceedings of Machine Learning and Systems*, 5, 2023.

[7] Intel corp. Automatic Parallelization with Intel Compilers. `https://www.intel.com/content/www/us/en/developer/articles/technical/automatic-parallelization-with-intel-compilers.html`, 2018.

[8] Intel Corporation. Intel c++ compiler user and reference guides (304968-022us).

[9] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[10] Paul Barton David McCandless, Tom Evans. The Rise and Rise of A.I. Large Language Models (LLMs) & their associated bots like ChatGPT. `https://informationisbeautiful.net/visualizations/the-rise-of-generative-ai-large-language-models-llms-like-chatgpt/`. [Online].

[11] Michael Dever. *AutoPar: automating the parallelization of functional programs*. PhD thesis, Dublin City University, 2015.

[12] Jacob Devlin et al. 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)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[13] LLVM Foundation. Polly - A Loop Optimizer in LLVM. `https://polly.llvm.org/docs/Architecture.html`, 2023.

[14] William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. Evaluation of OpenAI Codex for HPC Parallel Programming Models Kernel Generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ICPP Workshops '23, pages 136–144, New York, NY, USA, September. Association for Computing Machinery.

[15] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[16] Re'em Harel, Yuval Pinter, and Gal Oren. Learning to parallelize in a shared-memory environment with transformers. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022.

[17] Re'em Harel, Yuval Pinter, and Gal Oren. 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*, pages 450–452, 2023.

[18] Re'em Harel, Idan Mosseri, Harel Levin, Lee-or Alon, Matan Rusanovsky, and Gal Oren. Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential. *International Journal of Parallel Programming*, 48:1–31, 2020.

[19] Zheming Jin and Jeffrey S Vetter. A benchmark suite for improving performance portability of the sycl programming model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 325–327. IEEE, 2023.

[20] Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, Gal Oren, et al. Pragformer: Data-driven parallel source code classification with transformers. 2023.

[21] Tal Kadosh, Niranjan Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Mihai Capota, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, et al. Domain-specific code language models: Unraveling the potential for hpc codes and tasks. *arXiv preprint arXiv:2312.13322*, 2023.

[22] Tal Kadosh, Nadav Schneider, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. Advising openmp parallelization via a graph-based approach with transformers. *arXiv preprint arXiv:2305.11999*, 2023.

[23] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[24] Chunhua Liao, Daniel J Quinlan, Jeremiah J Willcock, and Thomas Panas. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, 38:361–378, 2010.

[25] Yinhan Liu et al. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[26] Quazi Ishtiaque Mahmud, Ali TehraniJamsaz, Hung D Phan, Nesreen K Ahmed, and Ali Jannesari. Autoparllm: Gnn-guided automatic code parallelization using large language models. *arXiv preprint arXiv:2310.04047*, 2023.

[27] Timothy Mattson, Yun (Helen) He, and Alice Koniges. *The OpenMP Common Core: Making OpenMP Simple Again (Scientific and Engineering Computation)*. The MIT Press, 2019.

[28] Reed Milewicz, Peter Pirkelbauer, Prema Soundararajan, Hadia Ahmed, and Tony Skjellum. Negative perceptions about the applicability of source-to-source compilers in hpc: A literature review. In *International Conference on High Performance Computing*, pages 233–246. Springer, 2021.

[29] Idan Mosseri, Lee-Or Alon, Re'Em Harel, and Gal Oren. Compar: Optimized multi-compiler for automatic openmp s2s parallelization. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems: 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings*, page 247–262, Berlin, Heidelberg, 2020. Springer-Verlag.

[30] Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? *arXiv preprint arXiv:2401.12554*, 2024.

[31] Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code?, January 2024.

[32] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. Modeling parallel programs using large language models. *arXiv preprint arXiv:2306.17281*, 2023.

[33] OpenAI. OpenAI ChatGPT. https://openai.com/blog/chatgpt. [Online].

[34] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 11 2015.

[35] Félix-Antoine Ouellet. *Parallélisation Automatique de Programmes Scientifiques Pour Systèmes Distribués*. PhD thesis, Université de Sherbrooke, 2016.

[36] Soratouch Pornmaneerattanatri, Keichi Takahashi, Yutaro Kashiwa, Kohei Ichikawa, and Hajimu Iida. Parallelizable loop detection using pre-trained transformer models for code understanding. In *International Conference on Parallel and Distributed Computing: Applications and Technologies*, pages 32–42. Springer, 2023.

[37] S Prema, R Jehadeesan, and BK Panigrahi. Identifying pitfalls in automatic parallelization of nas parallel benchmarks. In *Parallel Computing Technologies (PARCOMPTECH), 2017 National Conference on*, pages 1–6. IEEE, 2017.

[38] S Prema, Rupesh Nasre, R Jehadeesan, and BK Panigrahi. A study on popular auto-parallelization frameworks. *Concurrency and Computation: Practice and Experience*, 31(17):e5168, 2019.

[39] GNU Project. GNU Offloading and Multi-Processing Project (GOMP). https://gcc.gnu.org/projects/gomp/#omp5.0, 2023.

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[41] Miguel Romero Rosas, Miguel Torres Sanchez, and Rudolf Eigenmann. Should ai optimize your code? a comparative study of current large language models versus classical optimizing compilers. *arXiv preprint arXiv:2406.12146*, 2024.

[42] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[43] Yuanyuan Shen, Manman Peng, Shiling Wang, and Qiang Wu. Towards parallelism detection of sequential programs with graph neural network. *Future Generation Computer Systems*, 125:515–525, 2021.

[44] Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S Vetter. Comparing llama-2 and gpt-3 llms for hpc kernels generation. *arXiv preprint arXiv:2309.07103*, 2023.

A Background

A.1 Formal approaches to automatic parallelization

To mitigate the difficulties of manual parallelization, various automatic parallelization tools have been developed to assist programmers in automatically converting sequential programs into parallel ones. These tools free programmers from the need to manually insert parallelization directives, thus simplifying the software development process. These tools typically focus on loops, as loops are where programs spend the majority of their execution time.

Broadly, auto-parallelization tools can be classified into two types: compilers and source-to-source transformation tools.

Given that automatic parallelization is a form of program transformation, popular compilers like GCC [39], LLVM [13], and ICC [7] also perform automatic parallelization. For instance, GCC has supported automatic parallelization since version 4.3, released in 2012, and LLVM's Polly [15, 35] provides loop optimization and parallelization accessible through Clang [23]. These compilers leverage existing program analysis infrastructure, such as data-flow analysis, to ensure the correctness of the parallelization. They also use heuristics to prevent performance degradation by ensuring that parallelized loops have sufficient iterations.

An alternative approach to compiler-based parallelization is the source-to-source (S2S) transformation. S2S tools convert sequential code into parallel code while maintaining the original source code, allowing subsequent compilation with any standard compiler. Examples of S2S tools include AutoPar [11], Par4all [2], and Cetus [9], or a combination of those with ComPar [29]. S2S tools have the advantage of keeping the source code clean and enabling the use of different compilers for further optimization and deployment.

Despite their advantages, automatic parallelization tools face significant challenges [37, 28, 38]. One major issue is the substantial manual effort required to develop and maintain these tools. They are typically rule-based and rely on pattern matching, which necessitates continuous updates to keep up with revisions to the OpenMP specification [1]. For instance, the latest version of GCC does not fully support all features of OpenMP v5.0, and many S2S tools fail to incorporate newer OpenMP capabilities, such as task-based parallelism and offloading kernels to devices. Moreover, automatic parallelization tools can be overly strict, often failing to recognize opportunities for parallelization [18]. This conservativeness, while ensuring correctness, can lead to missed opportunities where parallelization could have been safely and beneficially applied. Additionally, when these tools do perform parallelization, they can sometimes result in suboptimal performance scalability [18]. The resulting parallelized code might not efficiently utilize the available computational resources, leading to limited performance gains or even performance degradation in some cases.

A.1.1 A motivating case study of the limitations of formal auto-parallelizing tools

To concretely assess the limitations of these formal tools, we decided to conduct a case study of two state-of-the-art auto-parallelization tools in their ability to parallelize serial programs. Specifically, we chose AutoPar and ICPC. AutoPar [24] is a state-of-the-art static source-to-source-based automatic parallelization tool. ICPC is the Intel C++ compiler, which provides advanced auto-parallelization capabilities as part of high-performance computing tools. ICPC [8] performs automatic parallelization at compile-time, leveraging both static analysis and dynamic profiling data to optimize code for modern multi-core architectures. Unlike AutoPar, which focuses on OpenMP directive insertion, ICPC integrates more advanced optimizations such as vectorization and memory access improvements, targeting low-level hardware features

Test dataset. As these tools would have been evaluated on existing serial-code benchmarks, we decided to generate a synthetic benchmark of serial code. Specifically, we decided to leverage ParEval [31], the latest work that has developed a benchmark to evaluate the ability of LLMs to generate parallel programs. Specifically, ParEval contains LLM prompts that represent 60 different coding problems related to scientific and parallel computing, with each problem, solved using programs written in 7 different parallel programming languages (such as CUDA, OpenMP, etc.) Moreover, every program has pre-defined test inputs as well as expected outputs. We collected 60 OpenMP-based programs generated by GPT-3.5-turbo and converted these parallel programs into serial ones by removing all OpenMP pragmas[5]. We found that these programs were using the C++17 standard.

Evaluation methodology. Both AutoPar and ICPC operate on complete programs (as against a single `for` loop) for automatic parallelization. Consequently, we applied both tools to parallelize every serial program from the test set, attempted to compile generated OpenMP-parallelized programs, run with the test inputs (if the compilation was successful), and compared their output with the expected output.

Findings. We found that ICPC could parallelize 2 of these 60 programs and 5 `for` loops in total. Similar to ICPC, AutoPar was able to parallelize 2 programs and 2 loops in total. When subjected to compile and run tests, ICPC-parallelized programs timed out, while AutoPar-parallelized programs encountered errors (see elaboration in Appendix D). We found that one of the reasons for the inability of these tools to parallelize input programs was possibly their incomplete support for the C++17 standard that is used by the input programs. We believe this limitation points to the need for manual efforts to support these tools.

We present three concrete examples of Autopar's incorrect parallelization in Figure 2. Specifically, the first two examples are of incorrect parallelization (false positive) — `for` loops that cannot be parallelized as per OpenMP specification — while the third example is a missed parallelization opportunity (false negative). In particular, the first two examples contain `for` loops that have a return statement, and the behavior of the loops is strictly based on the order of loop iterations. Parallelizing such a loop provably changes the semantics of the loop from its serial version. The third example contains a `for` loop that can be parallelized with OpenMP's `reduction(+:count)` clause, but AutoPar missed the parallelization opportunity. The missed opportunity could be because of non-affine constructs (i.e., `A [ i * n + j]`) in the loop body that AutoPar could not analyze statically. To summarize, we found that existing formal tools for auto-parallelization suffer from several limitations. Later in the evaluation, we present several more examples of incorrect parallelizations by AutoPar and ICPC.

```
size_t findLastShortBook(
  const class std::vector<Book>,
  std::allocator<Book>> &books) {
#pragma omp parallel for
 for (size_t i = books.size();
       i >= ((unsigned long )0) + 1;
       i += -1) {
   if (books[(i-1)].pages < 100) {
     return i - 1;}}
 // If no book with pages < 100 is
 // found, return an appropriate
 // value (e.g., books.size()).
 return books.size();
 }
```

Listing 1: Example 1

```
size_t findFirstEven(
  const class std::vector<int ,
  std::allocator<int>> &x) {
#pragma omp parallel for
 for (size_t i = 0;
       i <= x.size() - 1;
       i += 1) {
   if (x[i] % 2 == 0) {
    // Check if current element
    // is even. Return the index
    // if found
    return i;}}
 // Return -1 otherwise.
 return (-1);}
```

Listing 2: Example 2

```
int edgeCount(
  std::vector<int> const& A,
  size_t N) {
 int count = 0;
 size_t i = 0, j = 0;
#pragma omp parallel for
  reduction(+:count)
 for (i = 0; i < N; ++i) {
   for (j = 0; j < N; ++j) {
     if (A[i * N + j] == 1) {
       #pragma omp critical
       count++;}}}
 return count;
 }
```

Listing 3: Example 3

Figure 2: Examples of incorrect parallelization by AutoPar: The first two examples show `for` loops were parallelized by AutoPar but the parallelization changed the loop semantics. The last example shows the loop that can be parallelized, but AutoPar missed the parallelization opportunity.

A.2 Leveraging LLMs for parallel program generation

The emergence of LLMs, particularly those built on transformer architectures like the GPT (Generative Pre-trained Transformer) series [10], has sparked a revolution in NLP. These models have exhibited exceptional prowess in comprehending and producing human language [12, 25, 40]. Leveraging the inherent similarities between code and natural language, researchers have extended these capabilities to the realm of programming, paving the way for automating various software development tasks, including parallelization.

AI-based tools for automatic parallelization manifest in diverse forms, each offering a distinct approach and emphasis. OpenMP-specific tools address the challenge of OpenMP parallelization, scrutinizing serial code and proposing suitable OpenMP pragmas. Examples encompass PragFormer [16], OMPIFY [22], Graph2Par [6], HPCoder [21], and AutoParLLM [26]. On the other

[5]For the pragma erasing, we used AutoParBench's PragmaRemover tool: https://github.com/LLNL/AutoParBench/tree/master/tools/PragmaRemover

Authors	**Month**	**Year**	**Model**	**Task**	**Usage**	**Dataset**	**Ref**
Harel	April	2022	PragFormer	Classification	Fine-tuning	OpenOMP dataset	[17, 20]
Chen	May	2023	Graph2Par	Classification	Fine-tuning	OMPSerial dataset	[6]
Kadosh	May	2023	OMPIFY	Classification	Fine-tuning	HPCorpus dataset	[22]
Chen	June	2023	LM4HPC	Classification	Inference	OMP4Par dataset	[5]
Godoy	June	2023	Codex	Generation	Inference	Numerical Kernels	[14]
Nichols	June	2023	HPC-Coder	Generation	Fine-tuning	HPC-Coder dataset	[32]
Valero-Lara	September	2023	Llama-2, GPT-3	Generation	Inference	Numerical Kernels	[44]
Mahmud	October	2023	AutoParLLM	Classification	Fine-tuning, Inference	OMPSerial dataset	[26]
Pornman'	November	2023	CodeT5-FT	Classification	Fine-tuning	BigQuery public dataset	[36]
Kadosh	December	2023	MONOCODER	Generation	Pre-training	HPCorpus dataset	[21]
Nichols	January	2024	CodeLlama, StarCoder, GPT-3.5, GPT-4	Generation	Inference	ParEval	[30]
Chen	January	2024	OMPGPT	Generation	Pre-training, Fine-tuning	HPCorpus dataset	[4]
Rosas	June	2024	GPT-4, CodeLlama-70B	Generation	Inference	NAS, PolyBench	[41]

Table 4: Comparison of LLM-based models for OpenMP code, showing their usage (Inference, Fine-tuning, Pre-training + Fine-tuning) and corresponding training data. Almost all of the models use common ML metrics for success, especially OpenMP pragma string-based comparison.

hand, pre-trained HPC-oriented models such as MONOCODER [21] and OMP-GPT [3] are initially trained on expansive datasets before being fine-tuned for OpenMP-related tasks, capitalizing on their expansive comprehension of code structures and parallelization principles. We capture comparative details of these AI-based automatic parallelization approaches in Table 4, which highlights the diverse methodologies used in the field. These models differ not only in their underlying architectures but also in the nature of the tasks they are designed for and the datasets they leverage.

Meanwhile, general-purpose LLMs such as GPT-4 [33] and CodeLLaMa [42] are highly versatile and capable of addressing a wide range of programming tasks, including OpenMP parallelization. However, they lack specialized training in parallel programming paradigms. Due to the limited exposure to these paradigms in their training data, these models often struggle with key parallel programming challenges, such as reasoning about data distribution, managing race conditions, and implementing complex parallel algorithms [30].

Performance evaluations have been increasingly demonstrating the superiority of AI-based approaches over traditional methods. Notably, PragFormer has surpassed the source-to-source tool ComPar [29] in discerning parallelization potential. Similarly, AI-based tools have been advancing rapidly. Graph2Par has exhibited greater precision in predicting applicable OpenMP clauses compared to PragFormer. Moreover, both OMPIFY and PragFormer have outperformed ChatGPT in discerning the parallelization potential of loops.

However, despite these advancements, a significant gap remains in matching the capabilities of automatic parallelization compilers. Unlike previous AI-specific metrics that primarily focused on next-token prediction accuracy and comparison to ground truth, automatic parallelization tools are traditionally evaluated based on machine performance metrics such as runtime, scalability, and accuracy of computation results. OMPAR addresses this challenge by validating performance using HPC-centric methodologies as recommended by ParEval [30].

A.2.1 Limitations of LLMs in auto-parallelization

We have covered comparative analysis of existing parallelization-specific AI models in Table 4. We now talk briefly about the limitations of LLMs in automatic parallelization. Several recent papers [30] have systematically evaluated ability of LLMs in generating parallel code. In particular, authors of ParEval [30] have presented a systematic analysis of ability of various popular LLMs, including GPT-4, in auto-parallelization. As such, we quote the findings from these studies to summarize: "*LLMs are* *significantly worse at generating parallel code than they are at generating serial code*". Moreover, they further comment that "*the poor performance of LLMs on ParEval benchmark indicates that further efforts are necessary to improve the ability of LLMs to model parallel code and/or create new LLMs that are specialized for parallel code generation.*"

One major limitation contributing to the poor performance of current LLMs in parallelization tasks is that these models are primarily trained on raw source code, without leveraging the deeper code representations used by compilers, such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Data Flow Graphs (DFG). While source code provides surface-level syntax and semantics, it lacks the structural insights necessary for complex tasks like parallelization, which require understanding dependencies, data flow, and control flow. Compiler-level representations like ASTs capture the hierarchical structure of code, while DFGs model how data moves through variables and computations, both of which are crucial for identifying independent tasks that can be parallelized. The absence of these representations limits the ability of LLMs to detect parallelization opportunities, manage data dependencies, and ensure race condition-free execution, making them less effective in generating optimized parallel code.

B OMPar components

B.1 OMPify

The first major component of OMPar is OMPify. OMPify is an encoder-only transformer model trained on `for` loops in C and C++ to classify whether OpenMP is needed or applicable. Specifically, it was trained on approximately 54K `for` loops, with about half of them containing OpenMP and the rest not. The OMPify pipeline operates as follows: First, it generates the given code's data flow graph (DFG). Then, the model is fed with multiple code modalities, including the DFG and the plain code. The model returns a binary classification for the detection task. Results demonstrate that OMPify outperforms existing approaches, such as the general-purpose and popular ChatGPT (GPT-3.5) and the targeted PragFormer models, in terms of F1 score and accuracy. Specifically, OMPify achieves up to 90% accuracy on commonly used OpenMP benchmark tests such as NAS, SPEC, and PolyBench.

B.2 MonoCoder-*OMP*

The second major component of OMPar is the fine-tuned version of MonoCoder, MonoCoder-*OMP*. MonoCoder is a decoder-only model initially trained on corpora of HPC-programming languages. We fine-tuned it on approximately 25K `for` loops so that for a given `for` loop, MonoCoder generates the corresponding OpenMP pragma, particularly the *"#pragma omp for"* along with private and reduction clauses. For evaluation, MonoCoder was compared to the general-purpose ChatGPT (GPT-3.5) on the task of OpenMP pragma generation. MonoCoder significantly outperformed ChatGPT in both predicting the correct clauses and using the relevant variables inside the clauses.

Dataset. To create a dataset for the automatic OpenMP parallelization generation task, we curated a subcorpus of HPCorpus named *HPCorpus$_{OMP}$*. This subcorpus specifically includes all the `for` loops that have either `#omp parallel for` pragma that distributes the workload across multiple CPU cores or offloads tasks to a team of threads for GPU execution using `#omp target teams distribute`. The dataset also includes OpenMP pragmas containing `private`[6] and `reduction` clauses. It is important to note that we excluded the context of the loops to maintain focus on the loop-specific information. The details of the dataset in terms of these clauses are present in Table 5.

MonoCoder-*OMP training.* We fine-tuned the pretrained MonoCoder model to the OpenMP pragma generation task using the Huggingface `transformers` library on 2 NVIDIA V100 32GB GPUs at `fp32` precision. Fine-tuning used the AdamW optimizer with a linear warmup over the first 100 steps, followed by a linear decay over the remaining steps. Fine-tuning samples also had a maximum length of 2048 tokens, trained in 16 sample minibatches (8 per GPU). The fine-tuning was run for 4 epochs at the learning rate of 0.000016.

Evaluation setup. In the fine-tuning of MonoCoder on the *HPCorpus$_{OMP}$* subcorpus, the task was set as a generation task in which the complete `for` loop is input to the model, and the expected

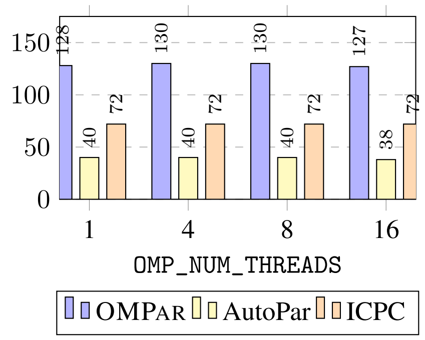
[6]To avoid the loss of crucial details, we designed *HPCorpus$_{OMP}$* such that the `private` clause encompasses instances of both `firstprivate` and `lastprivate`, as both of these clauses serve the purpose of creating a private instance of variables for each thread.

generated output was the pragma and its associated variables. We then evaluated MONOCODER-*OMP* on a test set of *HPCorpus*$_{OMP}$. For comparison, we assessed GPT-3.5 turbo in a zero-shot manner. Specifically, we supplied a prompt instructing it to *"Generate the optimal OpenMP pragma for the provided code"* when presented with the `for` loop.

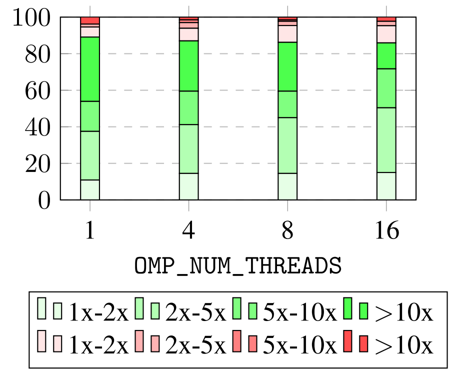
	`private`	`reduction`	`target`	`parallel for`
C	3,526	713	145	8,764
C++	2,122	1,424	345	18,151

Table 5: OpenMP clause breakdown in *HPCorpus*$_{OMP}$.

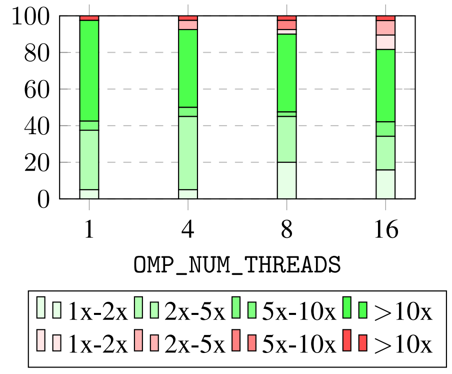
C OMPAR scale test results on HeCBench



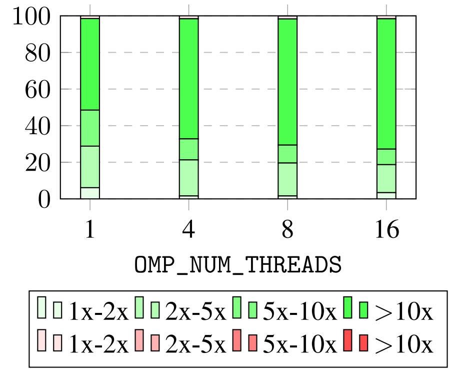
(a) Number of benchmarks completing the scale test for different automatic parallelization tools



(b) % of OMPAR-parallel benchmarks (Y-axis) with improvements (in Green) and degradations (in Red) over baseline



(c) % of AutoPar-parallel benchmarks (Y-axis) with improvements (in Green) and degradations (in Red) over baseline



(d) % of ICPC-parallel benchmarks (Y-axis) with improvements (in Green) and degradations (in Red) over baseline

Figure 3: Scale test performance of OMPAR-, AutoPar-, and ICPC-parallelized OpenMP benchmarks from HeCBench.

D Evaluating OMPAR on ParEval dataset

Since we presented the performance of AutoPar and ICPC on the ParEval dataset in the motivation section (subsubsection A.1.1), we decided to evaluate OMPAR on the ParEval dataset as well. Notice that ParEval is a much smaller dataset than HeCBench, and OMPAR has outperformed both AutoPar and ICPC on HeCBench. But we are presenting ParEval results for the sake of completeness and curious readers. As the performance of AutoPar and ICPC on ParEval was poor (parallelized only 2 benchmarks out of 6), we did not evaluate them on accuracy and scale tests.

ParEval OpenMP loop dataset. As we are only interested in obtaining OpenMP programs, we fed ParEval's 60 coding problems to GPT-3.5 and collected the output OpenMP programs. ParEval also comes with a set of compilation recipes and test inputs to evaluate if LLM-generated programs are syntactically- and semantically-correct. Using these recipes and inputs, we found that 32 programs of those 60 passed the compilation and runtime check. We then used these 32 OpenMP programs to extract 55 `for` loops, 37 of which had OpenMP `parallel for` pragma ("positive" loops) while the remaining 18 had no OpenMP pragma ("negative" loops). We call the dataset of 55 loops as ParEval OpenMP dataset.

Evaluation methodology. We used the dataset of 55 `for` loops to evaluate OMPAR on the accuracy test and the compile, run, and scale test.

Results on accuracy test. Figure 4 shows the performance of OMPAR on the accuracy test when subjected to 55 loops from the ParEval OpenMP dataset. As can be seen in the first row of the table, we found that the accuracy was relatively low (67%). The reason is also obvious from the table — we found that OMPAR was suggesting that all 18 "negative" `for` loops can be parallelized using OpenMP pragma. To confirm if this suggestion was indeed correct, we decided to subject OMPAR to compile and run the test. Similar to the observations made from the evaluation on HeCBench, we found that 11 of those 18 "negative" loops can indeed be parallelized using suggested OpenMP pragma, and more importantly, the suggested pragma and the loop when inserted in the program can compile correctly and also pass output verification check. In other words, 11 of those 18 false positives are actually true positives. Consequently, the accuracy of OMPAR improved to 87% (as shown in the second row of the table).

Results of scale test. Figure 5 shows the performance of OMPAR on the scale test with ParEval OpenMP loop dataset. For the scale test, we insert the loop along with its OMPAR suggested pragma in the original pragma and run the program with its default inputs. More importantly, we vary the number of threads `OMP_NUM_THREADS` from 1, 4, 8, and 16 while running those programs. The X-axis in the figure represents every loop from the dataset, while the Y-axis shows the speedup obtained by its corresponding program against the baseline of running the same program with a single thread. Overall, as can be seen, as the number of threads is increased from 4 to 16, the performance of the programs goes up as well – indicating that OMPAR suggested pragmas scale well with different number of threads.

Test setup	TP	FP	TN	FN	P	R	Acc
With GT label	37	18	0	0	67%	100%	67%
With C&R check	48	7	0	0	87%	100%	87%

Figure 4: OMPAR's performance on the accuracy test on the ParEval OpenMP dataset. (TP=True Positive, FP=False Positive, TN=True Negative, FN=False Negative, P=Precision, R=Recall, Acc=Accuracy)

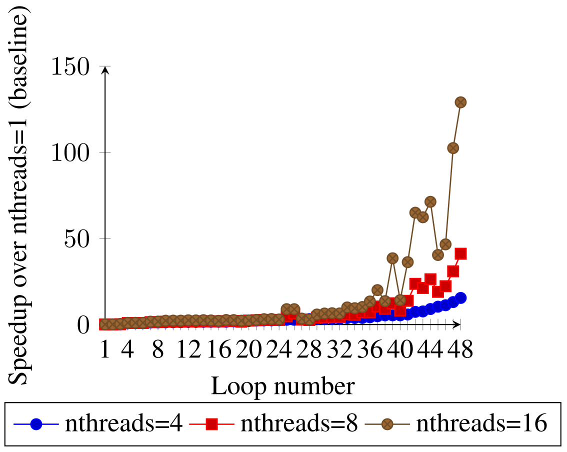


Figure 5: Performance of OMPAR on the scale test with 48 True Positives from ParEval OpenMP dataset

E Result analysis: What were some of AutoPar failures?

Given that AutoPar is a rule-based, state-of-the-art source-to-source transformation tool for automatic parallelization, it is natural to expect it to parallelize most, if not all, of the possible `for` loops. Our experimental evaluation on HeCBench revealed otherwise. Hence, we decided to investigate some of the cases where AutoPar failed to parallelize or incorrectly parallelized a loop. Below we present the most frequent failures that we observed in our experiments.

E.0.1 Cases of incorrect parallelization

AutoPar parallelized code failed to compile, prompting further analysis for examination. Upon investigation, we found that AutoPar had performed several erroneous code transformations.

(1) Applies parallelization (i.e., adds `#pragma`) to non-canonical loops. Section 2.6 of OpenMP-v4.5 [34] specifies a particular form of loops (called *canonical form*) that can be parallelized using OpenMP. All other loop forms are then non-canonical forms.

```
// AutoPar suggested OpenMP pragma
#pragma omp parallel for private (i)
for (i = 0;
     ((long) i) <= dim_cpu.space_elem - 1;
     i = i + 1) {
  rv_cpu[i].v = ((rand() % 10 + 1) / 10.0);
  ...}
```

We found that AutoPar suggested OpenMP parallel pragma for non-canonical loops also. For example, a loop shown above (obtained from HeCBench's `lavaMD-omp`) contains a typecasting expression involving the loop iteration variable, and such an expression is considered non-canonical by OpenMP. In spite of this, AutoPar parallelized the particular loop, thus leading to a compiler error.

(2) Applies parallelization to loops with `return` statements. OpenMP specifications do not allow the parallelization of loops containing a `return` statement in its body. This constraint ensures that all threads synchronize before exiting the code. In spite of this restriction, AutoPar suggested OpenMP parallelization pragma for such a loop as shown in the example obtained from HeCBench's `grep-omp` benchmark.

```
inline int ispmatch(::List *l) {
  int i;
  // AutoPar suggested OpenMP pragma
  #pragma omp parallel for private (i)
  for (i = 0; i <= l->n-1; i += 1) {
    if (l->s[i]->c == Match)
      return 1;}
  return 0;}
```

(3) Applies privatization and reduction to the same variable in OpenMP parallelization pragma. Privatization and reduction of the same variable are not allowed in OpenMP because `private` and `reduction` are conflicting directives for the same variable. In spite of this rule, AutoPar suggested OpenMP pragma for a loop (example below from `page-rank-omp` benchmark of HeCBench) that violated the rule.

```
// AutoPar suggested OpenMP pragma
#pragma omp parallel for \
  private (nb_links,i,j) reduction (+:nb_links)
for (i = 0; i <= n - 1; i += 1) {
  #pragma omp parallel for \
    private (j) reduction (+:nb_links)
  for (j = 0; j <= n - 1; j += 1) {
    nb_links += pages[i * n + j];}}
```

E.0.2 Cases of missed parallelization opportunities

Performance analysis of AutoPar-parallelized programs revealed that AutoPar had missed parallelization opportunities in several HeCBench benchmarks.

One commonly occurring theme among the loops that AutoPar missed was that many of them contained function calls. For instance, in the code shown below (obtained from `ace-omp` benchmark) AutoPar did not parallelize the loop as the loop body contains multiple function calls. Specifically, for loops containing function calls, AutoPar assumes that the function call might modify shared data, or the callee may contain loops, or the call may be data-dependent. AutoPar does not perform inter-procedural analysis or optimizations, such as function inlining, which could have alleviated this limitation. AutoPar missed parallelization opportunities at multiple places within the same benchmark for this exact same reason.

```
for (int ix = 0; ix <= 99; ix += 1) {
 for (int iy = 0; iy <= 99; iy += 1) {
   for (int iz = 0; iz <= 99; iz += 1) {
    if (ix < 100 - 1 && iy < 100 - 1 &&
        iz < 100 - 1 && ix > 0 && iy > 0 &&
        iz > 0) {
      double px =
        GradientX(phi,dx,dy,dz,ix,iy,iz);
      double py =
        GradientY(phi,dx,dy,dz,ix,iy,iz);
      double pz =
        GradientZ(phi,dx,dy,dz,ix,iy,iz);
      double sqGphi = px * px + py * py +
                      pz * pz;
      double c = 16.0 * W0 * epsilon;
      double w = Wn(px,py,pz,epsilon,W0);
      double w2 = w * w;
      Fx[ix][iy][iz] = w2 * px +
        sqGphi * w * c * dFunc(px, py, pz);
      Fy[ix][iy][iz] = w2 * py +
        sqGphi * w * c * dFunc(py, pz, px);
      Fz[ix][iy][iz] = w2 * pz +
        sqGphi * w * c * dFunc(pz, px, py);
    } else {
      Fx[ix][iy][iz] = 0.0;
      Fy[ix][iy][iz] = 0.0;
      Fz[ix][iy][iz] = 0.0;}}}}
```

In summary, we found that even AutoPar, which is a state-of-the-art source-to-source transformation tool based on a formal, rule-based approach, either missed parallelization opportunities or suggested incorrect pragmas. We believe this behavior can be attributed to missing newer rules (e.g., limited support for OpenMP-4.5) or conservative assumptions. Our experimental results reveal that AI-based approaches, such as ours, can mitigate these limitations, by learning from vast amounts of code and identifying parallelization patterns that rule-based systems may overlook.