A View of Large Language Models in HPC: Challenges and Opportunities

Anonymous EMNLP submission

Abstract

There is a growing interest in using machine learning techniques to automate and improve the process of generating code. With the rapid development of large language models (LLMs), various models have been created to help write and optimize code. However, they do not yet meet the stringent requirements of high-performance computing (HPC), where highly optimized and efficient code is essential. This paper explores the research direction of adapting and using LLMs for HPC code generation. We present the reasoning behind our position and suggest how existing ideas can be adapted and enhanced to meet the demands of HPC applications.

1 Introduction

Large language models (LLMs) represent a significant advancement in natural language processing (NLP), with one of their most notable applications being code generation. Both general-purpose LLMs with code ability, like GPT-4 (Achiam et al., 2023a), and LLMs specifically designed for code generation, such as CodeLlama (Roziere et al., 2023), have proven invaluable in software development. Integrating these models into the coding workflow helps developers accelerate coding tasks, automate repetitive processes, and even debug and improve existing code, significantly enhancing productivity and innovation.

While LLMs have demonstrated significant potential in general-purpose code generation (Xu et al., 2022; Liu et al., 2024; Zhong and Wang, 2023), there is now a growing interest in applying these models to high-performance computing (HPC). HPC is a specialized domain that utilizes parallel processing techniques on modern multi-core and many-core architectures to solve large-scale, complex computational problems. It plays a crucial role in various important applications such as climate modeling, computational chemistry, biomedical research, and astrophysical simulations. By providing a framework for scalable processing and analysis of massive datasets, HPC is fundamental to advancing scientific and technological frontiers (Netto et al., 2018). Consequently, the application of LLMs to HPC is attracting increasing attention.

Despite the recent success of code LLMs in tasks such as code generation and translation, there have been few detailed studies on applying LLMs to HPC tasks. It is essential to explore this application due to the significance of HPC. Appendix A showcases how LLMs can benefit the HPC domain. However, using LLMs for HPC poses unique challenges because of the distinct characteristics of this domain. HPC code differs from general code by focusing on maximizing computational performance through parallelism, low-level optimization, and efficient resource utilization. Each of these aspects presents unique challenges and requires specialized approaches to effectively leverage LLMs.

Position. This paper posits that integrating large language models into HPC parallel code generation holds significant benefits and potential. The distinct characteristics of HPC programs present challenges for current code LLMs to perform effectively. This paper explores how LLMs can be adapted for parallel code generation while addressing the associated challenges. The proposed research directions provide an HPC perspective and advocate for focused research on the development of LLMs for HPC, ultimately contributing to the broader field of machine learning and its applications in scientific computing.

2 Background

Large language models (LLM) and code LLM. LLMs represent a significant advancement in NLP. These models, trained on extensive textual datasets, excel in understanding and generating human language, demonstrating capabilities across various

NLP applications. **Code LLMs** are LLMs specifically designed for programming tasks. Typically, code LLMs are trained with both natural language (NL) and programming language (PL) corpora. Consequently, the knowledge of PL has led to remarkable outcomes of these models in various programming language tasks, such as code generation (Poldrack et al., 2023; Achiam et al., 2023b), code explanation (Khan and Uddin, 2022), and software testing (Schäfer et al., 2023). These achievements underscore LLMs' capabilities in supporting a wide range of programming tasks, enhancing code completion accuracy, and facilitating interactions between NL and PL. They have also inspired recent attempts (Chen et al., 2023c; Nichols et al., 2023; Dai et al., 2024; Chen et al., 2023b) to apply LLMs to the HPC domain. These works have shown promising potential in utilizing LLMs for HPC tasks, including parallel code generation.

High-Performance Computing Tasks. HPC plays a pivotal role in fields such as scientific research and data analytics. An HPC ecosystem is a comprehensive HPC environment that encompasses all the hardware, software, workflows, networking, and storage solutions (Grandinetti et al., 2018). HPC tasks refer to challenges and problems addressed within the HPC ecosystem. The uniqueness of HPC tasks arises from the specific characteristics of the HPC field itself. This work focuses on HPC code generation, which focuses on generating parallel code and ensuring compatibility with parallel computing frameworks such as OpenMP and MPI. It highlights the specialized focus required for HPC-related activities.

3 LLMs for HPC Code Generation: Problems and Directions

Motivation. Various works have applied LLMs to HPC parallel code generation. Chen et al. (2023c) were pioneers in applying LLMs to HPC tasks, including parallelism detection. Their findings indicated that LLMs can achieve competitive performance in determining whether sequential code can be parallelized. In a subsequent work (Chen et al., 2024), they focused on generating OpenMP pragmas, where their tailored LLM outperformed GPT-4 on this task. Moreover, Nichols et al. (2024) examined several existing general-purpose LLMs from an HPC perspective, evaluating the speed and scalability of the generated parallel code. Their findings showed that LLMs are significantly less effective at generating parallel code compared to serial code.

Previous works have demonstrated the potential of applying LLMs to HPC parallel code generation. However, these studies also highlight the necessity of a specialized approach to effectively harness the capabilities of LLMs for HPC tasks. This section delves into the challenges faced by current LLMs in this area and explores directions for optimizing LLMs for parallel code generation.

3.1 Dataset Considerations for HPC-Oriented LLMs

Problem 1: *As their name suggests, a central aspect of LLMs' powerful performance is the size of the dataset it is trained on. However, current LLMs are not specifically designed for HPC, and the datasets they are trained on do not focus on HPC code.*

Programming language focus. General-purpose LLMs or code LLMs are typically trained on datasets with a diverse set of programming languages, reflecting the wide array of languages used in software development. For example, the widely use Stack dataset (Kocetkov et al., 2022) covers 358 programming languages. However, within the HPC community, the dominant languages are C, C++, and Fortran due to their performance capabilities. Consequently, an LLM for parallel code generation should prioritize these languages to ensure relevance and applicability.

Beyond the choice of programming languages, it is crucial to include code that utilizes various parallel programming frameworks such as OpenMP, MPI, and CUDA. These frameworks are integral to HPC as they enable the parallel execution of code across multiple processors or cores, a fundamental aspect of achieving high performance. Therefore, a well-rounded HPC dataset should encompass a wide range of examples from these frameworks to cover the spectrum of parallel programming practices. Kadosh et al. (2023a) proposed an HPC dataset, HPCORPUS, by collecting C, C++, and Fortran codes from GitHub. LLMs developed by Chen et al. (2024) and Kadosh et al. (2023c) are trained on this dataset and outperform GPT-4 in specific HPC tasks.

However, directly collecting HPC code from public repositories may not be sufficient. Previous NLP studies (Xie et al., 2023; Lee et al., 2021) have demonstrated the importance of training data quality to the model performance. Unlike general-purpose code generation, HPC codes are typically expected to be compilable and executable. Various works (Chen et al., 2023a; Wang et al., 2022) have leveraged compiler feedback to improve data quality, enhancing both compilability and executability. These approaches can significantly improve the quality of the training dataset, thereby enhancing the performance of LLMs trained on higher-quality data.

Research direction 1: *The training dataset for an HPC-oriented LLM should focus on the predominant programming languages used in the HPC community and include a diverse range of parallel programming frameworks. Additionally, the quality of the HPC dataset should emphasize its compilability and executability.*

3.2 Training Strategies

Problem 2: *Traditional LLM training primarily focuses on general-purpose code, which lacks the specific considerations required for parallel code generation. Additionally, the existing models do not fully leverage the rich multimodal data available in the HPC ecosystem, which is crucial for understanding the comprehensive performance characteristics of HPC code.*

Fill-in-the-Middle (FIM). FIM (Bavarian et al., 2022) has been widely adopted by code LLMs due to its effectiveness in generating coherent and contextually accurate code. Different LLMs use various FIM configurations, either PSM (Prefix-Suffix-Middle) or SPM (Suffix-Prefix-Middle), and employ different FIM rates. For example, DeepSeek-Coder (Guo et al., 2024) adopted a 50% PSM configuration after conducting an ablation study. The FIM strategy is particularly critical for parallel code generation in the HPC domain, as parallelism typically exists in the middle of the code. Different parallel frameworks have their specific patterns. For example, in OpenMP, this involves inserting a single line of OpenMP pragmas at appropriate points within the code, while MPI requires adding multiple lines of MPI function calls to enable communication between parallel processes. An effective LLM for parallel code generation should determine its FIM strategy through experimental validation.

Multimodal learning and code representation. Multimodal learning has recently gained significant attention in foundational model research. This approach leverages data from multiple modalities to enhance the model's understanding and generation capabilities. The HPC community has developed numerous tools to provide critical information about HPC code. Information such as runtime data, dependency analysis, and performance reports is crucial for understanding HPC code from both static and dynamic perspectives. Moreover, code representation studies (TehraniJamsaz et al., 2023; Cummins et al., 2021; Chen et al., 2023d) in the HPC domain have shown that different representations of HPC code can offer various levels of insight, thereby enhancing the model's performance on HPC tasks. Previous studies (Chen et al., 2022; Steinert et al., 2023) applying machine learning in the HPC domain have utilized this multimodal information and achieved remarkable results.

Research direction 2: *To effectively harness the capabilities of LLMs for HPC parallel code generation, future research should focus on developing specialized training strategies that incorporate domain-specific characteristics and leverage multimodal data.*

3.3 Fine-Tuning Strategies

Problem 3: *Most existing LLMs are not fine-tuned for parallel code generation, and no well-recognized dataset exists for this task.*

Fine-tuning is a critical step in adapting LLMs to specific tasks or domains. By leveraging domain-specific datasets and training techniques, fine-tuning can significantly enhance the performance and applicability of LLMs in specialized fields. Fine-tuning for parallel code generation can benefit from advanced training techniques such as transfer learning and continual learning. Transfer learning involves starting with a pre-trained general-purpose LLM and then fine-tuning it on an HPC-specific dataset. Continual learning, on the other hand, involves continuously updating the model with new data as it becomes available. This is particularly useful in the rapidly evolving field of HPC, where new programming techniques and optimizations are constantly being developed.

However, no well-recognized dataset exists for fine-tuning LLMs specifically for parallel code generation. Most datasets collect as much public code as possible without examining the quality of the data. The HPC domain has developed several benchmarks, such as NAS Parallel Benchmarks (Bailey et al., 1993), SPEC (Müller et al., 2010), and Polybench (Yuki, 2014), for parallelization studies. Constructing fine-tuning datasets based on these benchmarks can help models generate efficient and optimized parallel code.

Research direction 3: *To address the lack of well-recognized fine-tuning datasets, future research should focus on creating high-quality, benchmark-based datasets for fine-tuning LLMs in parallel code generation. These datasets should be curated to include diverse examples from established HPC benchmarks, ensuring a comprehensive representation of parallel programming challenges.*

3.4 Prompt Engineering

Problem 4: *Effectively using prompt engineering techniques to guide LLM to generate optimized parallel code is challenging. They should consider the contextual details and domain-specific knowledge required for parallel code generation.*

Prompt engineering involves carefully crafting input prompts to guide LLMs to generate more accurate and contextually relevant outputs. In the context of parallel code generation, prompt engineering can be particularly effective when the prompts include relevant facts from previous interactions or external resources. For example, when converting sequential code to parallel code using OpenMP, the prompt can include details about the specific loops or sections that need parallelization, the desired level of parallelism, and any hardware constraints. Mahmud et al. (2023) crafted prompts for parallel code generation by including parallelization patterns generated by GNNs. Their superior performance indicates that LLMs can benefit from prompts enriched with external knowledge, which can be obtained from previous HPC or ML tools. Moreover, this approach can help models leverage external resources such as performance metrics, hardware specifications, and domain-specific libraries, further enhancing the accuracy and efficiency of the generated code.

Research direction 4: *Future research should focus on developing advanced prompt engineering techniques tailored for HPC parallel code generation. This includes integrating domain-specific knowledge, performance metrics, and hardware constraints into the prompts. Additionally, leveraging external resources and tools to provide context and enhance the prompts can significantly improve the quality of the generated code.*

3.5 Evaluation Metrics

Problem 5: *Current evaluation methods, such as HumanEval (Chen et al., 2021), are mostly designed for general-purpose code. Metrics and evaluation datasets specifically designed for generated HPC parallel code are needed.*

Evaluating the performance of LLMs in generating parallel code is crucial to ensure their effectiveness and applicability in the HPC domain. Evaluation impacts not only the assessment of the generated code but also other processes, such as fine-tuning. Unlike general code generation tasks, where correctness and readability are primary concerns, parallel code generation must also meet stringent performance criteria. Nichols et al. (2024) has made the first step in evaluating parallel code generated by LLMs from an HPC perspective. However, there is significant potential in this direction to create metrics and evaluation datasets for various HPC languages and parallelization frameworks.

Research direction 5: *Future research should focus on developing comprehensive evaluation methods and datasets specifically for HPC parallel code generation. This includes creating metrics that assess not only the correctness and readability of the code but also its performance, scalability, and efficiency in an HPC environment. Additionally, designing evaluation datasets that encompass a wide range of HPC languages and parallelization frameworks will provide a more robust and thorough assessment of LLM capabilities.*

4 Conclusion

In this paper, we have explored the potential and challenges of utilizing large language models for high-performance computing parallel code generation. While LLMs have demonstrated remarkable success in various general-purpose code generation tasks, adapting these models to the specific demands of HPC requires specialized approaches and considerations. We highlighted several critical areas that need attention for the effective integration of LLMs into parallel code generation from the perspective of HPC. By addressing these key areas, we believe it is possible to harness the full potential of LLMs for parallel code generation and other HPC tasks, ultimately contributing to advancements in both machine learning and high-performance computing.

5 Limitation

While our exploration into the potential of large language models (LLMs) for high-performance computing (HPC) parallel code generation highlights significant opportunities, it is also important to acknowledge the limitations and challenges that persist in this field. One of the primary limitations is the quality and availability of datasets specifically tailored for HPC tasks. Current datasets often lack the necessary diversity and depth required to train models effectively for HPC applications. Furthermore, many available datasets do not adequately address the compilability and executability of the generated code, which are critical factors for HPC. Second, the use of LLMs in HPC may also raise ethical and security concerns. The potential for biased outputs, data privacy issues, and the inadvertent generation of insecure code are significant risks that need to be addressed. Ensuring that LLMs adhere to ethical guidelines and security best practices is essential to their safe and effective deployment.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023a. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023b. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

David H Bailey, Eric Barszcz, Leonardo Dagum, and Horst D Simon. 1993. Nas parallel benchmark results. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):43–51.

David H Bailey et al. 1991. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. 2024. Ompgpt: A generative pre-trained transformer model for openmp. *arXiv preprint arXiv:2401.16445*.

Le Chen, Arijit Bhattacharjee, Nesreen K Ahmed, Niranjan Hasabnis, Gal Oren, Bin Lei, and Ali Jannesari. 2023a. Compcodevet: A compiler-guided validation and enhancement approach for code dataset. *arXiv preprint arXiv:2311.06505*.

Le Chen, Xianzhong Ding, Murali Emani, Tristan Vanderbruggen, Pei-Hung Lin, and Chunhua Liao. 2023b. Data race detection using large language models. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 215–223.

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

Le Chen, Quazi Ishtiaque Mahmud, and Ali Jannesari. 2022. Multi-view learning for parallelism discovery of sequential programs. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 295–303. IEEE.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Béatrice Creusillet et al. 2009. Par4all: Auto-parallelizing c and fortran for the cuda architecture. *Computer*.

Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. 2021. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning*, pages 2244–2253. PMLR.

Liuyao Dai, Hao Qi, Weicong Chen, and Xiaoyi Lu. 2024. High-speed data communication with advanced networks in large language model training. *IEEE Micro*.

Chirag Dave et al. 2009. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12).

Michael Dever. 2015. *AutoPar: automating the parallelization of functional programs*. Ph.D. thesis, Dublin City University.

William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. 2023. Evaluation of openai codex for hpc parallel programming models kernel generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, pages 136–144.

Lucio Grandinetti, Seyedeh Leili Mirtaheri, and Reza Shahbazian. 2018. *Big data and HPC: ecosystem and convergence*, volume 33. IOS Press.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Re'em Harel, Yuval Pinter, and Gal Oren. 2023. 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.

Re'em Harel et al. 2020. Source-to-source parallelization compilers for scientific shared-memory multicore and accelerated multiprocessing: analysis, pitfalls, enhancement and potential. *International Journal of Parallel Programming*, 48(1):1–31.

Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *Proceedings of the web conference 2020*, pages 2704–2710.

Intel. 2023. New 5th gen Intel® Xeon® processors are built with ai acceleration in every core. Retrieved from Intel.

Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023a. Quantifying openmp: Statistical insights into usage and adoption. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE.

Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, Gal Oren, et al. 2023b. Pragformer: Data-driven parallel source code classification with transformers. *Research Square*.

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

Tal Kadosh, Niranjan Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, Yuval Pinter, et al. 2023d. Scope is all you need: Transforming llms for hpc code. *arXiv preprint arXiv:2308.09440*.

Tal Kadosh, Nadav Schneider, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023e. Advising openmp parallelization via a graph-based approach with transformers. In *International Workshop on OpenMP*, pages 3–17. Springer.

Junaed Younus Khan and Gias Uddin. 2022. Automatic code documentation generation using gpt-3. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2021. Deduplicating training data makes language models better. *arXiv preprint arXiv:2107.06499*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

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

Idan Mosseri, Lee-or Alon, Re'Em Harel, and Gal Oren. 2020. 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 16*, pages 247–262. Springer.

Matthias S Müller, Matthijs Van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C Brantley, Chris Parrott, Tom Elken, et al. 2010. Spec mpi2007—an application benchmark suite for parallel systems using mpi. *Concurrency and Computation: Practice and Experience*, 22(2):191–205.

Marco AS Netto, Rodrigo N Calheiros, Eduardo R Rodrigues, Renato LF Cunha, and Rajkumar Buyya. 2018. Hpc cloud for scientific and business applications: taxonomy, vision, and research challenges. *ACM Computing Surveys (CSUR)*, 51(1):1–29.

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

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

Russell A Poldrack, Thomas Lu, and Gašper Beguš. 2023. Ai-assisted coding: Experiments with gpt-4. *arXiv preprint arXiv:2304.13187*.

S Prema et al. 2017. Identifying pitfalls in automatic parallelization of nas parallel benchmarks. In *Parallel Computing Technologies (PARCOMPTECH), 2017 National Conference on*, pages 1–6. IEEE.

S Prema et al. 2019. A study on popular auto-parallelization frameworks. *Concurrency and Computation: Practice and Experience*, 31(17):e5168.

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

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*.

Nadav Schneider, Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. Mpi-rical: Data-driven mpi distributed parallelism assistance with transformers. *arXiv preprint arXiv:2305.09438*.

Patrick Steinert, Stefan Wagenpfeil, Paul Mc Kevitt, Ingo Frommholz, and Matthias Hemmje. 2023. Parallelization strategies for graph-code-based similarity search. *Big Data and Cognitive Computing*, 7(2):70.

Ali TehraniJamsaz, Quazi Ishtiaque Mahmud, Le Chen, Nasreen K Ahmed, and Ali Jannesari. 2023. Perfograph: A numerical aware program graph representation for performance optimization and program analysis. *arXiv preprint arXiv:2306.00210*.

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

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132*.

Sang Michael Xie, Shibani Santurkar, Tengyu Ma, and Percy S Liang. 2023. Data selection for language models via importance resampling. *Advances in Neural Information Processing Systems*, 36:34201–34227.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

Tomofumi Yuki. 2014. Understanding polybench/c 3.2 kernels. In *International workshop on polyhedral compilation techniques (IMPACT)*, pages 1–5.

Li Zhong and Zilong Wang. 2023. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335*.

A Instances of LLMs Enhancing HPC Parallel Code Generation

In this section, we discuss an application of LLMs to the HPC problem of automatically generating parallel programs for shared memory systems (using OpenMP pragmas).

Shared memory systems are characterized by multiple compute cores (e.g., CPU cores) that share access to common caches (e.g., L3 cache). For instance, systems based on the 5th generation Intel Xeon processor (codenamed Emerald Rapids) (Intel, 2023), contain anywhere between 8 to 64 cores, all of which share access to the last level cache (L3 typically). Getting the best performance out of such systems requires writing parallel code, which divides the problem into subproblems and executes them in parallel on different cores. Writing a parallel version of serial code, however, is tricky, courtesy of typical multi-threading problems — it requires reasoning of data dependence, race conditions, deadlocks, etc. Programming standards such as OpenMP simplify this task considerably to the extent that OpenMP is the most popular parallel programming API in open-source (Kadosh et al., 2023a).

```
 //Serial code for element-wise multiply
 for (int i = 0; i < a.size(); i++) {
   a[i] = b[i] * c[i];
 }

//Parallel version of the above code
#pragma omp parallel for
for (int i = 0; i < a.size(); i++) {
  a[i] = b[i] * c[i];
}
```

Figure 1: Comparison between serial and parallel implementations of element-wise multiplication.

As an example, the first code snippet in Figure 1 shows a serial version of code that performs element-wise multiplication on two `std::vectors`, while the following code snippet shows the parallel version of the serial code. The `#pragma omp parallel for` pragma causes the OpenMP runtime to create a team of threads, where each thread operates on an individual subset of the iteration space, leading to the better utilization of underlying multiple compute cores. While standard compilers, such as GCC, LLVM, etc., and source-to-source translation tools (S2S), such as Cetus (Dave et al., 2009), AutoPar (Dever, 2015), Par4All (Creusillet et al., 2009), ComPar (Mosseri et al., 2020), etc., can automatically generate parallel versions of serial code, they, however, had limited success (Harel et al., 2020; Prema et al., 2017, 2019), especially because of a lack of robustness.

The limitations of the existing tools in automatically generating parallel versions of serial code have led to the introduction of AI-based tools for programming assistance. Instead of relying on formal program analysis passes (such as loop dependence analysis in compilers), AI-based tools for this problem leverage recent advancements in the field of NLP (especially Transformer architecture) to accurately determine the parallelization potential of code. A simple categorization of these AI-based tools could be as follows: (1) *OpenMP-specific tools*, such as PragFormer (Harel et al., 2023; Kadosh et al., 2023b), OMPify (Kadosh et al., 2023e), Graph2Par (Chen et al., 2023d), HPCoder (Nichols et al., 2023), AutoParLLM (Mahmud et al., 2023), etc., that are solely designed for the OpenMP parallelization problem, (2) *Pre-trained HPC-oriented models* that are the fine-tuned for OpenMP, such as MonoCoder (Kadosh et al., 2023c) and OMPGPT (Chen et al., 2024), and (3) *general-purpose tools*, such as ChatGPT, CodeLLaMa (Roziere et al., 2023), etc., that can solve the OpenMP parallelization problem, in addition to several other programming related or unrelated tasks (Godoy et al., 2023; Valero-Lara et al., 2023; Nichols et al., 2024). We will review these tools along with different design choices. (Since the last category of tools are not specifically designed for the OpenMP parallelization problem, we will not discuss their design choices.)

- *Problem formulation:* The problem of automatically parallelizing serial code using OpenMP can be divided into multiple subproblems. To be precise, the problem that these approaches attempt to solve can be defined as: *Given a piece of serial code (mostly* `for` *loops), determine if the code can be parallelized, and if so, suggest appropriate OpenMP pragma.* As the first part of the problem statement is a boolean question, tools such as PragFormer, OMPify, and Graph2Par formulate it as a binary classification problem (this same formulation also applied to other parallelization strategies, such as MPI (e.g., MPI-rical (Schneider et al., 2023)). Once these approaches determine the parallelization potential of a loop, the next subproblem is to suggest appropriate OpenMP pragma as a multi-class classification problem. Specifically, Graph2Par considers four specific items from OpenMP (`target`, `simd`, `private`, `reduction`) that could apply to a parallel loop. PragFormer and OMPify, on the other hand, consider two additional OpenMP clauses (`private` and `reduction`). Given the large number of clauses, library functions, and pragmas in OpenMP (Kadosh et al., 2023a), these approaches have a long way to go before the full range of OpenMP can be applied to HPC programming problems.

- *Source code representation:* The representation of the input serial code, is an important design decision for this problem as the accurate predictions depend upon the ability of the AI model to learn to reason about certain program properties (such as loop-carried dependence) that determine the parallelism potential. Treating source code as text and employing a sequence of tokens representation did not yield satisfactory results (Kadosh et al., 2023e), consequently, all of these approaches have leveraged sophisticated compiler-based code representations such as abstract-syntax tree (AST), data-flow graph (DFG) (in OMPify), or even specialized ones such as heterogeneous augmented abstract syntax tree (Augmented-AST) in Graph2Par (Chen et al., 2023d). Also, some of these approaches have devised new tokenization strategies. For instance, Kadosh et. al. have devised TokomPiler (Kadosh et al., 2023d) to address specific requirements of preprocessing HPC code (written mostly in C, C++, and Fortran) and compilation-centric tasks.

- *Training dataset:* The lack of curated, publicly-available datasets has forced teams working on these techniques to synthesize their own training datasets using various sources such as open-source programs containing OpenMP pragmas, parallel programming benchmarks (e.g., NAS parallel benchmark (Bailey et al., 1991)), etc. Specifically, a common approach followed for synthesis is to search C/C++ programs containing `for` loops that have OpenMP parallel loops (e.g., `#pragma omp parallel for`). The `for` loops are then used as input to the model, while their OpenMP pragmas (or their lack of) are used to generate appropriate labels. Thankfully, authors of these approaches have released their datasets publicly for further research (e.g., OMP_Serial by Graph2Par, Open-OMP by PragFormer). The most comprehensive HPC-oriented training dataset to this date is the HPCorpus (Kadosh et al., 2023a) dataset, containing a total of 300K repos, 70 GB, 9M files across C, C++, and Fortran code from GitHub, with hundreds of thousands of those functions able to compile successfully (Chen et al., 2023a). This repo includes common parallel programming APIs, such as MPI, CUDA, OpenCL, TBB, Cilk, OpenACC, and SYCL.

- *Model architecture:* These approaches employ popular deep learning innovations such as Transformer architecture, graph neural networks (as source code can be represented as a graph), etc., to find parallelism opportunities within serial code and then generate parallel versions by automatically inserting OpenMP pragmas. Specifically, Graph2Par uses a modified transformer model called heterogeneous graph transformer (HGT) (Hu et al., 2020), while OMPify builds on top of GraphCodeBERT(Guo et al., 2020), a pre-trained model for programming languages that considers the inherent structure of the code by accepting source code along with its dataflow graph. Models employed by these approaches are typically smaller than LLMs such as CodeLLaMa, GPT-3.5, etc., that can also parallelize serial code. In spite of the smaller sizes, these approaches have outperformed larger models such as ChatGPT on the task of parallelizing serial code (Kadosh et al., 2023c; Chen et al., 2024).

- *Results:* Overall, better and problem-specific code representations have helped these OpenMP-specific approaches outperform code LLMs on the OpenMP parallelization problem. Specifically, PragFormer has shown that it can outperform a formal, source-to-source tool called ComPar on the task of detecting parallelization potential of a loop (0.8 vs 0.5 accuracy). Graph2Par, on the other hand, has shown that it can outperform PragFormer on the task of predicting OpenMP clauses applicable to a parallel loop (0.89 vs 0.85 accuracy in predicting the applicability of `private` clause). More importantly, both OMPify and PragFormer have shown that they can outperform ChatGPT (GPT-3.5) on determining the parallelization potential of a loop (0.4 vs 0.86 accuracy) (Kadosh et al., 2023e).