Can Test-Time Compute Help LLMs Write Low-Resource Parallel Code Better?

Gautam Singh[1], **Arjun Guha**[2], **Bhavya Kailkhura**[1]*, **Harshitha Menon**[1]*
[1]Lawrence Livermore National Laboratory [2]Northeastern University

Abstract

Although LLMs excel at data-rich coding tasks *e.g.*, writing general Python scripts, they often struggle at writing low-resource languages for High-Performance Computing (HPC). Recently, test-time program search driven by LLMs has emerged as a promising approach to enhance LLMs' capabilities. However, there is a lack of systematic studies investigating test-time search for low-resource HPC coding tasks. In this work, we conduct the first such study to our knowledge, providing empirical data about how different test-time search methods perform when moving from a high-resource to a low-resource language for HPC. Under a simple test-time search framework, we evaluate different choices of proposers and verifiers. Our experiments on the ParEval benchmark *(i)* show on average a 23–26% boost in `pass@1` using test-time search with a small search budget, and *(ii)* reveal gaps in LLMs as proposers, verifiers, and feedback providers.

1 Introduction

Although LLMs have shown impressive performance for coding tasks in data-rich programming languages, e.g., Python or JavaScript [9, 35, 32, 44, 16, 17], using LLMs to write code for High-Performance Computing (HPC) remains elusive [28, 30, 14, 15, 42, 17, 41, 27]. This is because HPC involves the use of niche programming models and domain-specific libraries on specialized and ever-changing systems—creating a long tail of data-scarce and low-resource tasks that LLMs struggle with [5, 26]. If solved, this would extend the productivity benefits of LLMs beyond simple coding tasks to the realistic codebases and systems that serve as workhorses for many critical applications.

Recently, scaling test-time compute has emerged as a promising paradigm for enhancing LLM capabilities across diverse tasks [45, 23, 1, 37]. In this line, a key approach is *program search*: combining the LLM's creativity in proposing programs with rigorous verification and selection [37, 13, 31]. However, these successes are largely confined to data-rich coding tasks, while for low-resource HPC code generation, the use of search is still in its infancy [34, 43, 38]. Prior HPC studies, to our knowledge, have focused on a single programming language and search method [43], without systematically examining *how different test-time search methods perform when moving from a high-resource to a low-resource language*. Such a study would provide valuable insights into the dos and don'ts of applying test-time search for low-resource coding for HPC.

In this paper, we take a simple testbed where a proposer generates candidate solutions, while a verifier scores them to pick the best one. We evaluate different choices of proposer and verifier on two representative parallel programming frameworks: OpenMP, a mature and widely adopted shared-memory API well-represented in LLM training data, and Kokkos, a modern C++ abstraction layer for performance portability that remains relatively low-resource and niche in code corpora.

Contributions. *(1)* For parallel code generation using LLMs, we present the first study (to our knowledge) comparing test-time search methods on high-resource and low-resource programming

*Equal advising.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: DL4C.

languages side-by-side. *(2)* Within a simple test-time search framework, we evaluate distinct proposers, verifiers, and feedback sources; and show what the implications are when moving from OpenMP to Kokkos. *(3)* Our experiments on the ParEval benchmark show, on-average, a 23–26% improvement in `pass@1` with test-time search, with a search budget of 10 candidates. *(4)* For low-resource settings, we identify key gaps in using LLMs as proposers, verifiers, and feedback providers. Notably, we see LLMs struggle with writing syntactically correct low-resource code, which seems to burden the search budget and prevents efficient search at an algorithmic level. *(5)* Finally, we find that high-quality feedback can mitigate LLM's shortcomings when generating low-resource code.

2 Test-Time Search for Parallel Code Generation

As a testbed for this study, we take a simple yet broadly applicable test-time search framework: *Propose-Verify-Select*. In this framework, given a problem x, N candidate solutions are proposed via an LLM-based proposer followed by verification and selection of the best candidate [37]. This is described in Algorithm 1. Also, see Appendix B and C for more details.

Algorithm 1 Propose–Verify–Select. Given problem x, the procedure returns a code y.

```
1: define ProposeVerifySelect(x)
2:     y_1, ..., y_N ← Proposer(x)
3:     i_best ← arg max_{i=1,...,N} Verifier(x, y_i)
4:     return y_{i_best}
```

2.1 Proposers

A proposer proposes N potential code solutions $y_1, \ldots, y_N$ for the given problem x. In this work, we test three proposers implemented using `Llama-3.3-70B-Instruct`[2]: *(1)* Independent and Identically Distributed (IID) Sampling, *(2)* Self-Refinement, and *(3)* Diversity Prompting.

IID (Nucleus) Sampling. One of the simplest and widely adopted proposer for test-time search is IID Sampling, which generates N code samples in parallel from the LLM: $\{y_1, \ldots, y_N\} \sim p(y|x)$. For this, we employ Nucleus Sampling with `temperature` $= 0.6$ and `top_p` $= 0.95$.

Sequential Refinement. Different from IID Sampling, Sequential Refinement offers a way to iteratively improve the previous candidates. That is, given problem x, an LLM first proposes an initial solution y_1. In each n-th refinement iteration, we sample a refinement-feedback z_n for the current solution y_n, and then use that feedback to refine y_n to obtain a new proposed solution y_{n+1}. We can summarize this process as follows:

$$y_1 \sim p_{\text{initial}}(y_1|x) \quad \Longrightarrow \quad z_n \sim p_{\text{feedback}}(z_n|y_n, x) \quad \Longrightarrow \quad y_{n+1} \sim p_{\text{refine}}(y_{n+1}|z_n, y_n, x).$$

In implementing this, we test two sources of refinement feedback of varying quality: *(1) LLM Feedback:* We prompt an LLM to write feedback for the code directly. *(2) Tool Feedback:* In this case, an external build tool first provides the ground-truth build outcome, which is then fed to the LLM to write feedback. The latter approach is expected to produce higher-quality feedback since it is informed by the ground-truth build outcome [36].

Diversity Prompting. In Nucleus Sampling, the candidates are often surface-level variations of the same high-level algorithm, thus harming algorithm-level exploration. With diversity prompting, we seek to alleviate this by prompting the LLM to first write N distinct algorithm-level strategies $z_1, \ldots, z_N$ to solve x, and then write the surface-level implementations $y_1, \ldots, y_N$ for those N strategies, respectively.

2.2 Verifiers

A *verifier* or a *scorer* takes a parallel code as input and assigns it a *scalar score* or a *reward*. To test where LLM-as-a-verifier falls in the spectrum between the best-possible and an uninformative verifier, we evaluate the following three verifiers: *(1) LLM Verifier.* For this, we prompt an LLM to assign the given code a score between 0 and 10. *(2) Oracle Verifier.* As a gold-standard verifier, the Oracle Verifier builds and executes the proposed code and then assigns it a score based on the build success and the execution outcome. *(3) Random Verifier.* Finally, as an example of an uninformative verifier, this verifier assigns a random score to each code candidate.

[2] `https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct`

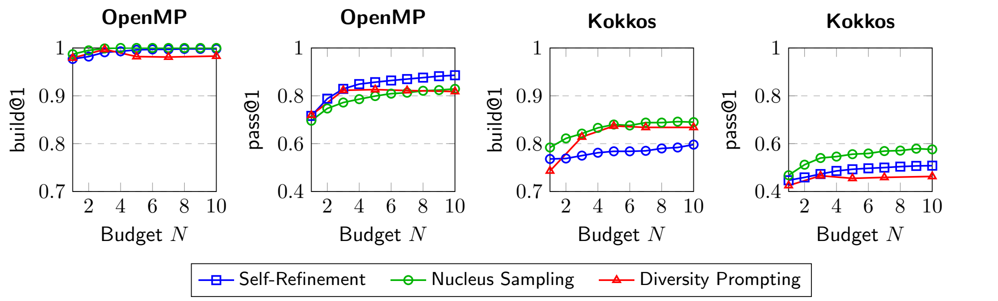

Figure 1: **Comparison of Proposers for Serial-to-Parallel Translation Task.** We report `build@1` and `pass@1` with respect to sampling budget N. The focus of this plot is to compare the proposers; therefore, we employ the Oracle verifier to select the best candidate. Moving from OpenMP to Kokkos, we find *(1)* build@1 drops, showing LLMs struggle with low-resource syntax, and *(2)* as a result, algorithm-level exploration—where Self-Refinement and Diversity Prompting shine in OpenMP—fails to benefit Kokkos.

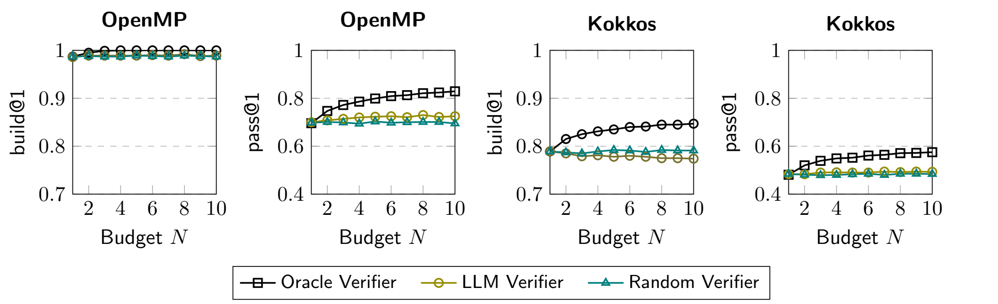

Figure 2: **Comparison of Verifiers for Serial-to-Parallel Translation Task.** We report `build@1` and `pass@1` with respect to sampling budget N. We use `Nucleus Sampling` as the proposer. We note that LLM Verifier performs similarly to the naive and uninformed Random Verifier baseline.

3 Related Work

A number of works have started to utilize LLMs for HPC tasks [8, 10, 7]. Several works [40, 20, 24, 25] have attempted to enhance LLMs' capabilities for HPC in ways that are orthogonal to our current work, e.g., by incorporating novel tokenization, representation, and retrieval. Several works [29, 42, 14, 15, 18, 19, 2] have focused on code generation for HPC, however, these do not investigate test-time computation. Some works develop datasets for task-specific training [22] while others propose benchmarks [4, 28, 11, 33] for evaluating parallel code. However, these works also do not focus on test-time search methods. Although test-time computing and tool-use [45, 23, 1, 37, 31, 50, 12, 49, 51, 47, 48, 39, 3, 21, 46] has started being leveraged for HPC [34, 43, 38], these works do not compare high and low resource languages for HPC, unlike our work. *OMPGPT* [6] leverages chain-of-thought as a test-time compute method, but their study is limited to OpenMP.

4 Experiments

In experiments, we seek to address the following questions: (1) What is the effect of the choice of the proposer? (2) What is the effect of the choice of the verifier? (3) What is the effect of the feedback source in sequential refinement? (4) Importantly, how do the answers to these questions change when we shift from a high-resource to a low-resource programming language?

Metrics and Benchmark. We report `build@1` and `pass@1` computed on a parallel programming evaluation benchmark called *ParEval* [28]. We evaluate on the task of translating a serial code to a parallelized implementation in OpenMP and Kokkos.

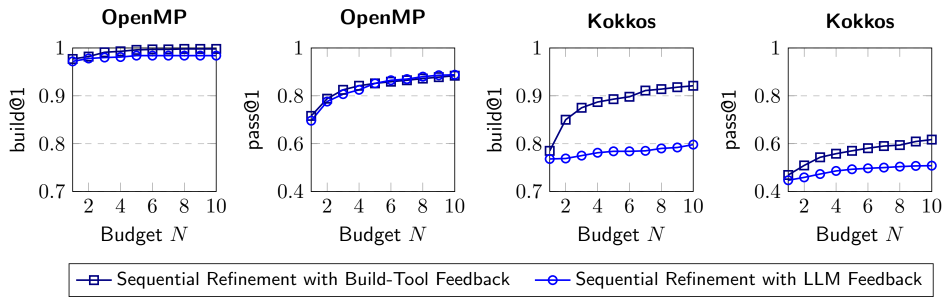

Figure 3: **Comparison of Feedback Source in Sequential Refinement.** We report `build@1` and `pass@1` with respect to sampling budget N. We compare distinct sources of refinement-feedback for sequential refinement. We find that while high-quality build-tool feedback has negligible benefits for OpenMP, such feedback has noticeable benefits for the low-resource language Kokkos.

High-Resource versus Low-Resource Language. In this work, we take *OpenMP* as an instance of a relatively high-resource parallel programming language and *Kokkos* as an instance of an emerging and low-resource parallel programming language.

4.1 Results

Test-Time Search Improves `pass@1` On-Average. In Figure 1, we observe a general upward trend in `pass@1` of the generated code as the search budget increases from $N = 1$ to $N = 10$. OpenMP's `pass@1` is 0.7 without test-time search and improves by nearly 26%, reaching 0.88 at $N = 10$ using test-time search, with Sequential Refinement performing the best. In Kokkos, we see worse `pass@1` than OpenMP, as Kokkos is a low-resource language. Yet, we see an upward scaling trend as the search budget is increased, with Nucleus Sampling performing the best. It must be noted, however, that the above results hold *on average* over the 54 problems in ParEval, while the picture may vary on a per-problem basis (see Figures 4 and 5 for plots by problem-type).

Syntactic Correctness and Its Effect on Test-Time Search. A key observation in Figure 1 is that while `build@1` is near-perfect (≈ 0.97) for high-resource OpenMP, it is much lower (≈ 0.8) for low-resource Kokkos. This suggests that LLMs inherently struggle with writing syntactically correct code for low-resource language. For test-time search, this creates a major hurdle: *it prevents us from conducting efficient program search at the level of algorithms and puts additional burden on the search budget to look for syntactic correctness*. This effect can be noticed from the fact that Self-Refinement and Diversity Prompting are among the best-performers for OpenMP, while they are the worst-performers for Kokkos. When refinement-feedback or diversity prompt asks the LLM to write an algorithmic variation or improvement, the LLM is capable of implementing it in OpenMP with correct syntax, but not for Kokkos. As such, for Kokkos, syntactic noise produced by Nucleus Sampling is more effective for producing performance gains.

LLM Shows Gaps in Identifying Good Solutions and Providing Good Feedback. In Figure 2, we find that LLM verifier's ability to distinguish good and bad solutions is lacking, as it performs similarly to the random verifier. In Figure 3, we compare whether and how Sequential Refinement benefits from the quality of the refinement-feedback. For high-resource OpenMP, we find that build-tool feedback has negligible benefits over LLM-only feedback since the LLM is already fluent at writing OpenMP syntax. On the contrary, for low-resource Kokkos, the build-tool feedback provides noticeable benefits, making it outperform Nucleus Sampling under identical search budget.

5 Discussion

In this work, we shed light on how LLM-based test-time search methods perform as we move from a high-resource to a low-resource programming language. Our results suggest that for low-resource languages, LLMs struggle at writing syntactically correct code preventing efficient search at the level of algorithms and adding an additional burden of syntactical correctness. We also reveal gaps in using LLMs as a verifier. Results also suggest that high-quality syntax-related feedback may mitigate syntactic correctness issues in low-resource languages. For limitations of this work, see Appendix A.

Acknowledgments and Disclosure of Funding

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, through solicitation DE-FOA-0003264, “Advancements in Artificial Intelligence for Science," under Award Number DE-SC0025598. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory (LLNL) under Contract DE-AC52-07NA27344 (LLNL-CONF-2011022).

References

[1] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[2] Aman Chaturvedi, Daniel Nichols, Siddharth Singh, and Abhinav Bhatele. Hpc-coder-v2: Studying code llms across low-resource parallel languages. In *ISC High Performance 2025 Research Paper Proceedings (40th International Conference)*, pages 1–14. Prometeus GmbH, 2025.

[3] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[4] Le Chen, Nesreen Ahmed, Mihai Capotă, Ted Willke, Niranjan Hasabnis, and Ali Jannesari. Pcebench: A multi-dimensional benchmark for evaluating large language models in parallel code generation. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 546–557. IEEE, 2025.

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

[6] Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. Ompgpt: A generative pre-trained transformer model for openmp. In *European Conference on Parallel Processing*, pages 121–134. Springer, 2024.

[7] Le Chen, Xianzhong Ding, Murali Emani, Tristan Vanderbruggen, Pei-Hung Lin, and Chunhua Liao. 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, 2023.

[8] 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.

[9] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[10] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

[11] Joshua H Davis, Daniel Nichols, Ishan Khillan, and Abhinav Bhatele. Pareval-repo: A benchmark suite for evaluating llms with repository-level hpc translation tasks. *arXiv preprint arXiv:2506.20938*, 2025.

[12] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*, 2023.

[13] Alhussein Fawzi and Bernardino Romera Paredes. Funsearch: Making new discoveries in mathematical sciences using large language models. *DeepMind Google*, 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*, pages 136–144, 2023.

[15] William F Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S Vetter. Large language model evaluation for high-performance computing software development. *Concurrency and Computation: Practice and Experience*, 36(26):e8269, 2024.

[16] Nam Huynh and Beiyu Lin. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint arXiv:2503.01245*, 2025.

[17] Sathvik Joel, Jie JW Wu, and Fatemeh H Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*, 2024.

[18] Tal Kadosh, Niranjan Hasabnis, Prema Soundararajan, Vy A Vo, Mihai Capota, Nesreen Ahmed, Yuval Pinter, and Gal Oren. Ompar: Automatic parallelization with ai-driven source-to-source compilation. *arXiv preprint arXiv:2409.14771*, 2024.

[19] Tal Kadosh, Niranjan Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Mihai Capotă, Abdul Wasay, Guy Tamir, Ted Willke, Nesreen Ahmed, et al. Monocoder: Domain-specific code language model for hpc codes and tasks. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2024.

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

[21] Kuang-Huei Lee, Ian Fischer, Yueh-Hua Wu, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. Evolving deeper llm thinking. *arXiv preprint arXiv:2501.09891*, 2025.

[22] Bin Lei, Caiwen Ding, Le Chen, Pei-Hung Lin, and Chunhua Liao. Creating a dataset for high-performance computing code translation using llms: a bridge between openmp fortran and c++. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2023.

[23] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[24] 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.

[25] Yusuke Miyashita, Patrick Kin Man Tung, and Johan Barthélemy. Llm as hpc expert: Extending rag architecture for hpc data. *arXiv preprint arXiv:2501.14733*, 2024.

[26] Federico Mora, Justin Wong, Haley Lepe, Sahil Bhatia, Karim Elmaaroufi, George Varghese, Joseph E Gonzalez, Elizabeth Polgreen, and Sanjit A Seshia. Synthetic programming elicitation for text-to-code in very low-resource programming and formal languages. *Advances in Neural Information Processing Systems*, 37:105151–105170, 2024.

[27] Noujoud Nader, Patrick Diehl, Steve Brandt, and Hartmut Kaiser. Llm & hpc: Benchmarking deepseek's performance in high-performance computing tasks. *arXiv preprint arXiv:2504.03665*, 2025.

[28] Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 281–294, 2024.

[29] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. Hpc-coder: Modeling parallel programs using large language models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–12. Prometeus GmbH, 2024.

[30] Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, Todd Gamblin, and Abhinav Bhatele. Performance-aligned llms for generating fast code. *arXiv preprint arXiv:2404.18864*, 2024.

[31] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[32] OpenAI. Introducing ChatGPT. `https://openai.com/index/chatgpt/`, 2022.

[33] Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint arXiv:2406.06647*, 2024.

[34] Asif Rahman, Veljko Cvetkovic, Kathleen Reece, Aidan Walters, Yasir Hassan, Aneesh Tummeti, Bryan Torres, Denise Cooney, Margaret Ellis, and Dimitrios S Nikolopoulos. Marco: A multi-agent system for optimizing hpc code generation using large language models. *arXiv preprint arXiv:2505.03906*, 2025.

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

[36] Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *arXiv preprint arXiv:2303.14100*, 2023.

[37] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[38] Zachariah Sollenberger, Rahul Patel, Saieda Ali Zada, and Sunita Chandrasekaran. Llm4vv: Evaluating cutting-edge llms for generation and evaluation of directive-based parallel programming model compiler tests. *arXiv preprint arXiv:2507.21447*, 2025.

[39] Vighnesh Subramaniam, Yilun Du, Joshua B Tenenbaum, Antonio Torralba, Shuang Li, and Igor Mordatch. Multiagent finetuning: Self improvement with diverse reasoning chains. *arXiv preprint arXiv:2501.05707*, 2025.

[40] Ali Tehrani, Arijit Bhattacharjee, Le Chen, Nesreen K Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. *Advances in Neural Information Processing Systems*, 37:100965–100999, 2024.

[41] Keita Teranishi, Harshitha Menon, William F Godoy, Prasanna Balaprakash, David Bau, Tal Ben-Nun, Abhinav Bhatele, Franz Franchetti, Michael Franusich, Todd Gamblin, et al. Leveraging ai for productive and trustworthy hpc software: Challenges and research directions. *arXiv preprint arXiv:2505.08135*, 2025.

[42] 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.

[43] Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel ai agent & evaluation benchmarks. *arXiv preprint arXiv:2507.23194*, 2025.

[44] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[46] Kyle Wong, Alfonso Amayuelas, Liangming Pan, and William Yang Wang. Investigating the transferability of code repair for low-resource programming languages. *arXiv preprint arXiv:2406.14867*, 2024.

[47] Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, James Xu Zhao, Min-Yen Kan, Junxian He, and Michael Xie. Self-evaluation guided beam search for reasoning. *Advances in Neural Information Processing Systems*, 36:41618–41650, 2023.

[48] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

[49] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[50] Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024.

[51] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023.

A Limitations

There are multiple avenues to expand the current study: *(i)* including more test-time compute approaches, e.g., tree search, evolutionary algorithms, etc. *(ii)* including more base LLMs, *(iii)* including more instances of high and low-resource languages, and *(iv)* testing additional sources of refinement-feedback.

B Details of the Experimental Setup

Sampling Parameters. In all the responses generated from the LLMs, we use a `temperature` of 0.6 and a `top_p` of 0.95. This follows the setup originally provided by the ParEval benchmark[28].

Definitions of `build@1` and `pass@1`. We employ the standard definitions and implementations of these metrics as in the ParEval benchmark [28]. Following ParEval, for all reported metrics, we execute the test-time search procedure 20 times and report an average over those.

Compute Resource Usage. Each inference run was performed on a single node with a total CPU RAM and VRAM equal to 512GB. We used the `Llama-3.3-70B-Instruct` model. We used vLLM[3] to accelerate the inference speed. The computation time needed depends on the search procedure, however, a single experiment over all 54 Serial-to-Parallel ParEval translation problems takes less than 8 hours. Each code's build success and correctness was evaluated on CPU using 1, 4, and 16 threads.

C Prompts

In this section, we provide the prompts used for each proposer.

Listing 1: Prompt for Nucleus Sampling Proposer.

```
{PROBLEM PROMPT}

Complete the code. The code MUST start with the line ‘{SIGNATURE}‘. Do
     NOT write any other code or explanations. Enclose your response
    inside a markdown code block i.e., ‘‘‘cpp and ‘‘‘.
```

Listing 2: Prompt for Initial Code Generation for the Sequential Refinement Proposer.

```
{PROBLEM PROMPT}

Complete the code. The code MUST start with the line ‘{SIGNATURE}‘. Do
     NOT write any other code or explanations. Enclose your response
    inside a markdown code block i.e., ‘‘‘cpp and ‘‘‘.
```

Listing 3: Prompt for Obtaining LLM Feedback for the Sequential Refinement Proposer.

```
{PROBLEM WITH CANDIDATE SOLUTION PROMPT}

Write in plain text how to improve the above completion to resolve
    build errors, runtime errors, algorithmic or logical errors, and
    other computational inefficiencies (if any). Be precise. Each item
     in the list should clearly indicate the code snippet or line to
    be edited, what it must be edited to, and why.
```

[3] `https://github.com/vllm-project/vllm`

Listing 4: Prompt for Refining a Previous Code using Feedback for the Sequential Refinement Proposer.

```
{PROBLEM WITH CANDIDATE SOLUTION PROMPT}

Complete the code again making use of this feedback:

{FEEDBACK}

The code MUST start with the line ‘{SIGNATURE}‘. Do NOT write any
    other code or explanations. Enclose your response inside a
    markdown code block i.e., ‘‘‘cpp and ‘‘‘.
```

Listing 5: Prompt for *Diversity Prompting* Proposer.

```
{PROBLEM PROMPT}

Write {N} distinct possible completions of this code. First, write in
    plain text your thought process about these {N} possible
    completions. Each completion MUST be fundamentally and
    algorithmically distinct from others. Then write the code for
    these {N} completions. Enclose each code completion you write in a
     C++ markdown block i.e., ‘‘‘cpp and ‘‘‘. The contents of each
    code block MUST start with the line ‘{SIGNATURE}‘.
```

D Additional Experimental Results

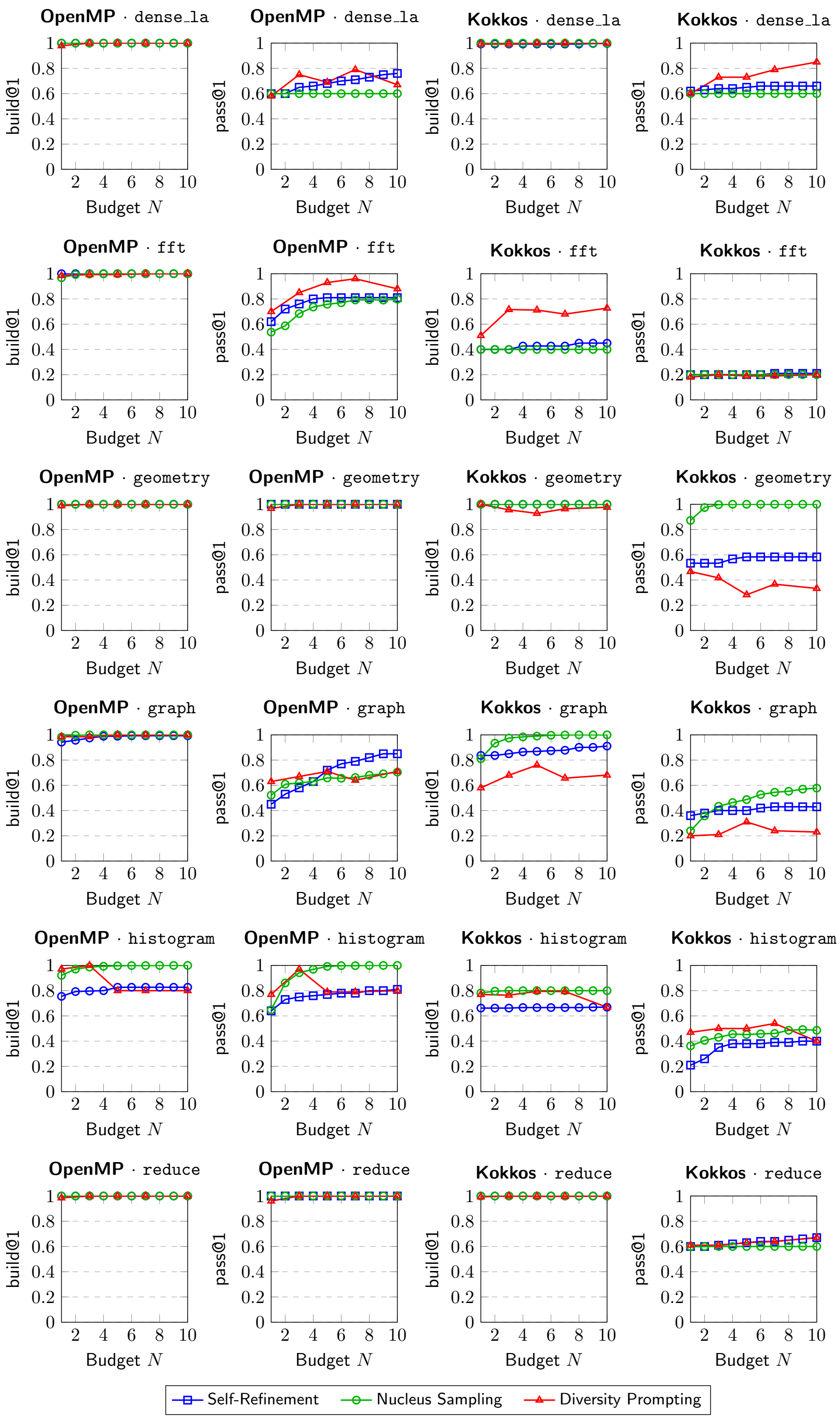

Figure 4: **Comparison of Proposers by Problem-Type for Serial-to-Parallel Translation Task.** We report `build@1` and `pass@1` with respect to varying sampling budgets. Since the focus of this plot is to compare the proposers, we employ the Oracle verifier to select the best candidate.

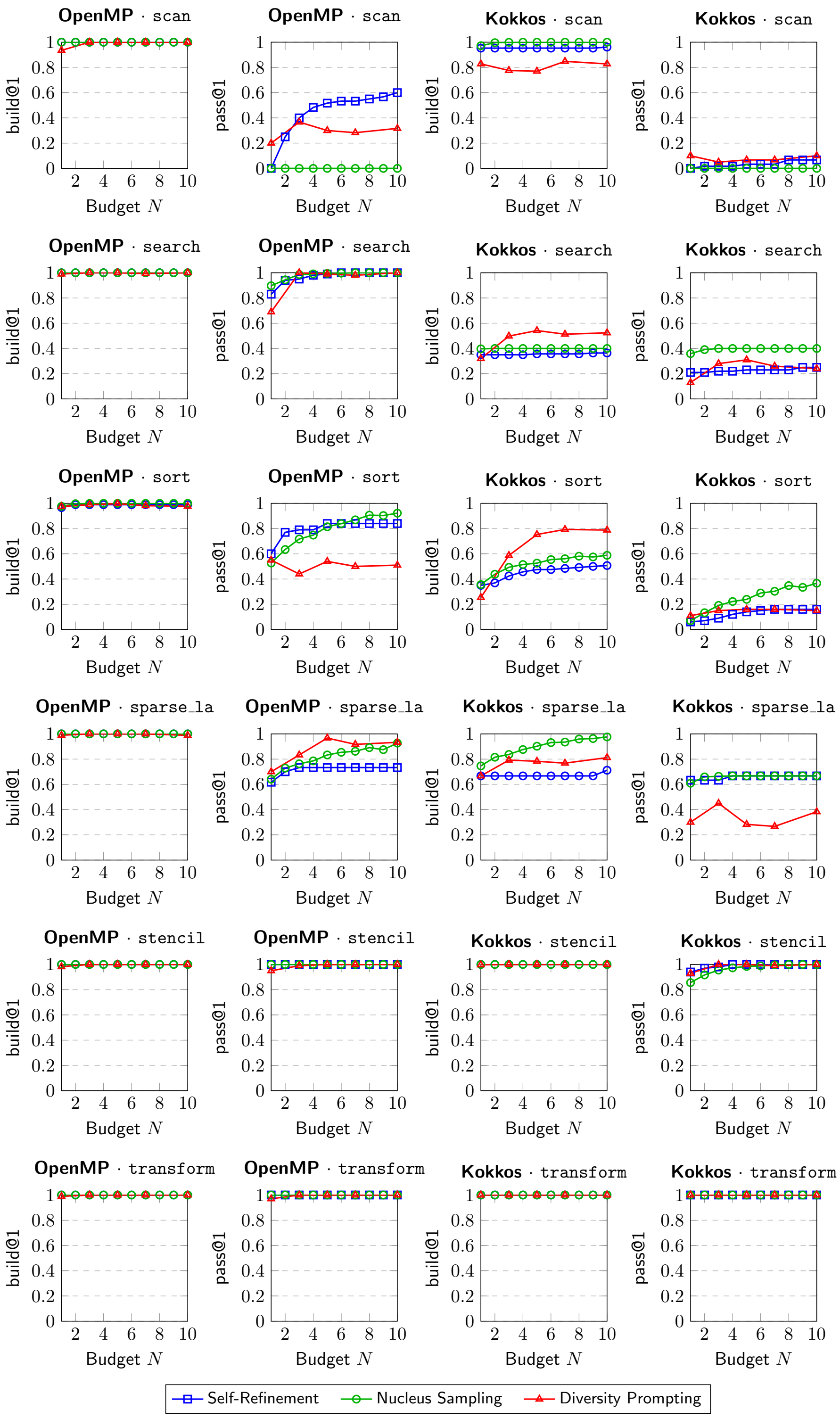

Figure 5: **Comparison of Proposers by Problem-Type for Serial-to-Parallel Translation Task.** We report `build@1` and `pass@1` with respect to varying sampling budgets. Since the focus of this plot is to compare the proposers, we employ the Oracle verifier to select the best candidate.

E Qualitative Examples of LLM-Generated Responses

```
// CANDIDATE 1

double smallestArea(Kokkos::View<const Point*> &points) {
  double minArea = std::numeric_limits<double>::max();
  Kokkos::parallel_reduce(Kokkos::RangePolicy<>(0, points.size()),
    [&](int i, double& lmin) {
      for (int j = i + 1; j < points.size(); ++j) {
        for (int k = j + 1; k < points.size(); ++k) {
          double area = triangleArea(points(i), points(j), points(k));
          lmin = std::min(lmin, area);
        }
      }
    },
    Kokkos::Min<double>(minArea));
  return minArea;
}
```

```
// CANDIDATE 2

double smallestArea(Kokkos::View<const Point*> &points) {
  double minArea = std::numeric_limits<double>::max();
  Kokkos::parallel_reduce(Kokkos::RangePolicy<>(0, points.size()),
    [&](int i, double& lmin) {
      for (int j = i + 1; j < points.size(); ++j) {
        for (int k = j + 1; k < points.size(); ++k) {
          double area = triangleArea(points(i), points(j), points(k));
          lmin = std::min(lmin, area);
        }
      }
    },
    Kokkos::Min<double>(minArea));
  return minArea;
}
```

```
// CANDIDATE 3

double smallestArea(Kokkos::View<const Point*> &points) {
  double minArea = std::numeric_limits<double>::max();
  Kokkos::parallel_reduce(Kokkos::RangePolicy<>(0, points.size()),
    [&](int i, double& lmin) {
      for (int j = i + 1; j < points.size(); ++j) {
        for (int k = j + 1; k < points.size(); ++k) {
          double area = triangleArea(points(i), points(j), points(k));
          lmin = std::min(lmin, area);
        }
      }
    },
    Kokkos::Min<double>(minArea));
  return minArea;
}
```

Figure 6: Generated Code Candidates using IID (Nucleus) Sampling in Kokkos for a ParEval Problem (12_geometry_smallest_triangle).

```cpp
// ITERATION 1

double smallestArea(Kokkos::View<const Point*> &points) {
  double minArea = std::numeric_limits<double>::max();
  Kokkos::parallel_reduce(Kokkos::RangePolicy<>(0, points.size()),
    [&](const int i, double& lmin) {
      for (int j = i + 1; j < points.size(); ++j) {
        for (int k = j + 1; k < points.size(); ++k) {
          double area = triangleArea(points(i), points(j), points(k));
          if (area < lmin) {
            lmin = area;
          }
        }
      }
    },
    Kokkos::Min<double>(minArea));
  return minArea;
}
```

```cpp
// ITERATION 2

double smallestArea(Kokkos::View<const Point*> &points) {
  return Kokkos::parallel_reduce(Kokkos::RangePolicy<>(0, points.size()),
    [&](const int i, double& lmin) {
      const Point& pi = points(i);
      for (int j = i + 1; j < points.size(); ++j) {
        const Point& pj = points(j);
        for (int k = j + 1; k < points.size(); ++k) {
          const Point& pk = points(k);
          double area = triangleArea(pi, pj, pk);
          if (area < lmin) {
            lmin = area;
          }
        }
      }
    },
    Kokkos::Min<double>());
}
```

```cpp
// ITERATION 3

double smallestArea(Kokkos::View<const Point*> &points) {
  return Kokkos::parallel_reduce(Kokkos::RangePolicy<>(0, points.size()),
    [&](const int i, double& lmin) {
      lmin = std::numeric_limits<double>::max();
      const Point& pi = points(i);
      for (int j = i + 1; j < points.size(); ++j) {
        const Point& pj = points(j);
        for (int k = j + 1; k < points.size(); ++k) {
          const Point& pk = points(k);
          double area = triangleArea(pi, pj, pk);
          if (area < lmin) {
            lmin = area;
          }
        }
      }
    },
    Kokkos::Min<double>(std::numeric_limits<double>::max()));
}
```

Figure 7: Generated Code Candidates using Self-Refinement in Kokkos for a ParEval Problem (`12_geometry_smallest_triangle`).

```cpp
// CODE

double smallestArea(Kokkos::View<const Point*> &points) {
  double minArea = std::numeric_limits<double>::max();
  Kokkos::parallel_reduce(Kokkos::RangePolicy<>(0, points.size()),
    [&](const int i, double& lmin) {
      for (int j = i + 1; j < points.size(); ++j) {
        for (int k = j + 1; k < points.size(); ++k) {
          double area = triangleArea(points(i), points(j), points(k));
          if (area < lmin) {
            lmin = area;
          }
        }
      }
    },
    Kokkos::Min<double>(minArea));
  return minArea;
}
```

```
# LLM GENERATED FEEDBACK

Here's a list of potential improvements to the code:

1. **Initialization of minArea**: The line `double minArea = std::numeric_limits<double
    >::max();` initializes `minArea` to the maximum possible double value. However, this
     value is not being used correctly in the `Kokkos::parallel_reduce` function. To fix
     this, the `minArea` variable should be initialized within the lambda function, and
    the `Kokkos::Min` reducer should be used to update the `minArea` variable.
   - Code snippet to edit: `double minArea = std::numeric_limits<double>::max();` and `
       Kokkos::Min<double>(minArea)`.
   - Edit to: Remove `double minArea = std::numeric_limits<double>::max();` and use `
       Kokkos::Min<double>()` instead of `Kokkos::Min<double>(minArea)`.
   - Why: The `Kokkos::Min` reducer will automatically initialize the minimum value to
       the maximum possible double value and update it accordingly.

2. **Redundant copy of points**: In the lambda function, `points(i)`, `points(j)`, and `
    points(k)` are used to access the points. However, these points are being accessed
    by value, which may involve a copy of the point objects.
   - Code snippet to edit: `double area = triangleArea(points(i), points(j), points(k))
       ;`.
   - Edit to: `double area = triangleArea(const Point& points(i), const Point& points(j),
        const Point& points(k));` or use `const Point& pi = points(i); const Point& pj =
        points(j); const Point& pk = points(k);` and then pass `pi`, `pj`, and `pk` to `
       triangleArea`.
   - Why: This avoids unnecessary copies of the point objects and improves performance.

3. **Potential division by zero**: Although not present in this specific code, if the `
    triangleArea` function is used elsewhere with points that are collinear (i.e., have
    zero area), it may cause division by zero errors or NaN (Not a Number) values.
   - Code snippet to edit: None in this specific code, but consider adding checks in the
       `triangleArea` function.
   - Edit to: Add checks to handle the case where the points are collinear.
   - Why: This prevents potential division by zero errors or NaN values.

4. **Error handling**: The code does not handle potential errors that may occur during
    execution, such as out-of-bounds access or invalid input.
   - Code snippet to edit: None in this specific code, but consider adding error handling
        mechanisms.
   - Edit to: Add try-catch blocks or other error handling mechanisms to handle potential
        errors.
   - Why: This improves the robustness of the code and prevents crashes or unexpected
       behavior.

5. **Parallelization**: The code uses `Kokkos::parallel_reduce` to parallelize the
    computation, but it may not be optimal for large inputs due to the nested loops.
   - Code snippet to edit: The lambda function within `Kokkos::parallel_reduce`.
   - Edit to: Consider using a more efficient parallelization strategy, such as
       parallelizing the outer loop only or using a different parallelization library.
   - Why: This can improve the performance and scalability of the code for large inputs.
```

Figure 8: An Example of LLM-Generated Feedback in Kokkos during a Self-Refinement Iteration for a ParEval Problem (`12_geometry_smallest_triangle`).

```
// CANDIDATE 1

double smallestArea(Kokkos::View<const Point*> &points) {
  // Brute Force Approach
  double minArea = std::numeric_limits<double>::max();
  Kokkos::parallel_for(Kokkos::RangePolicy<>(0, points.size()), KOKKOS_LAMBDA(const int i
      ) {
    for (int j = i + 1; j < points.size(); ++j) {
      for (int k = j + 1; k < points.size(); ++k) {
        double area = triangleArea(points(i), points(j), points(k));
        if (area < minArea) {
          minArea = area;
        }
      }
    }
  });
  return minArea;
}
```

```
// CANDIDATE 2

double smallestArea(Kokkos::View<const Point*> &points) {
  // Divide and Conquer Approach
  int numPoints = points.size();
  Kokkos::View<double*> minAreas("minAreas", numPoints);
  Kokkos::parallel_for(Kokkos::RangePolicy<>(0, numPoints), KOKKOS_LAMBDA(const int i) {
    double minArea = std::numeric_limits<double>::max();
    for (int j = i + 1; j < numPoints; ++j) {
      for (int k = j + 1; k < numPoints; ++k) {
        double area = triangleArea(points(i), points(j), points(k));
        if (area < minArea) {
          minArea = area;
        }
      }
    }
    minAreas(i) = minArea;
  });
  double smallest = minAreas(0);
  for (int i = 1; i < numPoints; ++i) {
    if (minAreas(i) < smallest) {
      smallest = minAreas(i);
    }
  }
  return smallest;
}
```

```
// CANDIDATE 3

double smallestArea(Kokkos::View<const Point*> &points) {
  // Random Sampling Approach
  int numSamples = 1000;
  Kokkos::View<double*> areas("areas", numSamples);
  Kokkos::Random_XorShift64_Pool<> randPool(12345);
  Kokkos::parallel_for(Kokkos::RangePolicy<>(0, numSamples), KOKKOS_LAMBDA(const int i) {
    int idx1 = randPool.get_state().rand() % points.size();
    int idx2 = randPool.get_state().rand() % points.size();
    int idx3 = randPool.get_state().rand() % points.size();
    double area = triangleArea(points(idx1), points(idx2), points(idx3));
    areas(i) = area;
  });
  double smallest = areas(0);
  for (int i = 1; i < numSamples; ++i) {
    if (areas(i) < smallest) {
      smallest = areas(i);
    }
  }
  return smallest;
}
```

Figure 9: Generated Code Candidates using Diversity Prompting in Kokkos for a ParEval Problem (`12_geometry_smallest_triangle`). We can see that each candidate implements a distinct algorithmic strategy (i.e., brute force, divide-and-conquer, and random sampling) due the the diversity prompt.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Yes, claims match the scope of our experimental results.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: A separate section on limitations is included.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: There are no theoretical results in this work.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We provide the LLM prompts in the appendix and other sampling parameters in the main paper.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: We intend to release our source code around the time of camera ready submissions. The benchmark that we have used in this work i.e., ParEval, is already publicly available.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [NA]

Justification: The paper does not require a training phase since the focus is on test-time compute on top of the base large-language model. As such, a split of training and test set is not required. The benchmark ParEval that we have evaluated on is treated as a test-set in its entirety.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [NA]

Justification: As this is a test-time compute approach, there is no randomness due to the training runs themselves.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We have provided the details in Appendix B.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Yes, the paper conforms to the Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: The paper seeks to advance the state of the art in HPC code generation. The ultimate societal impact is determined not by our current work but by the downstream HPC application that uses it.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We do not release any new models or data.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Yes, the benchmark that we have used in this paper to perform evaluations has been cited and credited.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: There are no new assets contributed by this work.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: There is no crowdsourcing or human experiment conducted in this work.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: There were no human subjects involved.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: Although LLM is an integral part of our methodology, its use in this work is standard as per the prior works.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.