MonoCoder: Domain-Specific Code Language Model for HPC Codes and Tasks

Tal Kadosh[1, 2], Niranjan Hasabnis[3], Vy A. Vo[4], Nadav Schneider[1, 2], Neva Krien, Mihai Capotă[4], Abdul Wasay[4], Guy Tamir[5], Ted Willke[4], Nesreen Ahmed[4], Yuval Pinter[1], Timothy Mattson and Gal Oren[6, 7]

[1]Ben-Gurion University, [2]IAEC, [3]Code Metal, [4]Intel Labs, [5]Intel, [6]Technion, [7]Stanford University

talkad@post.bgu.ac.il, niranjan@codemetal.ai, vy.vo@intel.com,
nadavsch@post.bgu.ac.il, nevo.krien@gmail.com, mihai.capota@intel.com,
abdul.wasay@intel.com, guy.tamir@intel.com, ted.willke@intel.com,
nesreen.k.ahmed@intel.com, pintery@bgu.ac.il, tim@timmattson.com,
galoren@stanford.edu

Abstract*—With easier access to powerful compute resources, there is a growing trend in AI for software development to develop large language models (LLMs) to address a variety of programming tasks. Even LLMs applied to tasks from the high-performance computing (HPC) domain are huge in size and demand expensive compute resources for training. This is partly because LLMs for HPC tasks are obtained by finetuning existing LLMs that support several natural and/or programming languages. We found this design choice confusing — *why do we need LLMs trained on natural languages and programming languages unrelated to HPC for HPC-specific tasks?

In this line of work, we aim to question choices made by existing LLMs by developing smaller language models (LMs) for specific domains — we call them *domain-specific LMs*. Specifically, we start with HPC as a domain and build an HPC-specific LM, named MonoCoder, which is orders of magnitude smaller than existing LMs but delivers better performance on non-HPC and HPC codes. Specifically, we pre-trained MonoCoder on an HPC-specific dataset (named HPCorpus) of C and C++ programs mined from GitHub. We evaluated the performance of MonoCoder against state-of-the-art multi-lingual LLMs. Results demonstrate that MonoCoder, although much smaller than existing LMs, outperforms other LLMs on normalized-perplexity tests (in relation to model size) while also delivering competing CodeBLEU scores for high-performance and parallel code generations. In other words, results suggest that MonoCoder *understands* HPC code better than state-of-the-art LLMs.

MonoCoder source code is available at our GitHub repository.

I. Introduction

Recent breakthroughs in the field of AI have led significant attention to language models (LMs) due to their advanced capabilities in natural language processing (NLP) [1]. Large language models (LLMs), particularly exemplified by models such as GPT-3 [2] and its successors [3], deployed in the conversational form of ChatGPT [4], have demonstrated the potential to grasp intricate linguistic structure and semantics, sparking exploration of their applicability beyond NLP.

In parallel, the field of high-performance computing (HPC) has been tackling increasingly complex and data-intensive problems [5]. The field of HPC has experienced advancements in hardware, software, and algorithms, resulting in improvements in computational performance and efficiency [6], [7]. Combining the two trends, integrating LMs into HPC workflows has emerged as a compelling avenue for innovation [8]. For instance, several recent efforts have explored the application of LLMs in assisting HPC programmers in automatically inserting OpenMP pragmas or MPI functions in code [9]–[16], overcoming the limitations of existing static tools [17]–[21].

Although existing LLMs have shown remarkable results on programming tasks [22], such as code generation, or bug fixing, we found several limitations. First, we found that they perform surprisingly poorly on the HPC-related programming tasks, such as code parallelization and vectorization [8], [11]–[13], [23]. Our observations are corroborated by the latest work by Nichols et al. [23] which specifically evaluates parallel code generation performance of start-of-the-art LLMs, such as GPT-4 by building a benchmark named ParEval (Parallel Code Generation Evaluation). Nichols et al. find that "*LLMs are significantly worse at generating parallel code than they are at generating serial code*". Moreover, they further comment that "*the poor performance of LLMs on ParEval benchmark indicates that further efforts are necessary to improve the ability of LLMs to model parallel code and/or create new LLMs that are specialized for parallel code generation.*"

This finding led us to ponder several research questions: *(i) How well do existing LLMs perform on domain-specific tasks such as those for the HPC domain, (ii) Will more domain-specific training data help in improving the performance?, (iii) When applying LMs for a specific domain, do we finetune existing LMs, or should we train them from scratch?* This last question is specifically important in the context of the enormous training costs of LLMs, which is the second limitation of existing LLMs. As an example, HPC-Coder [13], a recently-introduced LM for

This research was supported by the Israeli Council for Higher Education (CHE) via the Data Science Research Center, Ben-Gurion University of the Negev, Israel; Intel Corporation (oneAPI CoE program); Pazy Foundation; and the Lynn and William Frankel Center for Computer Science. Computational support was provided by HPE HPC & AI Cloud, Intel Developer Cloud, and the NegevHPC project. Part of this work was completed when Niranjan Hasabnis was at Intel Labs and Gal Oren with NRCN.

HPC tasks, is obtained by finetuning PolyCoder [24] on an HPC dataset; PolyCoder itself is a code LM (not specific to HPC) that is trained on a corpus made up of 249 GB of programs written in 12 programming languages. As another example, HPC-GPT [25], LM4HPC [8], and AutoParLLM [26] rely on LLaMA as their base model — with up to 34 billion parameters — or even on GPT-4. Such setups looked counter-intuitive to us — *Is it not enough to train an LM on HPC-specific languages only? In other words, why do we need an LM trained on Java or Python — with tens or hundreds of billion parameters — for HPC-specific tasks?* More importantly, we believe that such *domain-specific LMs* would be computationally as well as financially efficient to train.

In this paper, we hypothesize that HPC-specific LMs (e.g., smaller LMs that are designed and trained specifically on HPC datasets) would perform better than existing LMs for HPC tasks. We perform two experiments to validate this hypothesis. The purpose of the first experiment is to build a smaller LM that performs similar, if not better, than existing LLMs in terms of generic language understanding tasks (such as code completion). Towards that end, in the first experiment, we build a smaller LM, called MonoCoder, by reducing the number of layers of PolyCoder by a factor of 4 and pretraining *only* on C and C++ codes. We then empirically validate that MonoCoder, despite being a smaller model than PolyCoder, achieves a comparable perplexity score to PolyCoder in generic code completion task. The purpose of the second experiment is then to evaluate the performance of MonoCoder on HPC-specific tasks. Towards that end, we obtain the CodeBLEU scores of parallel code generation models with increasing context.

We exploit one insight that we learned while building MonoCoder. Specifically, we found that existing code LMs capture "local" semantics of code structures that leads to their degraded performance on the code completion task. An example of "local" semantics could be a variable named `i` is an index variable of a `for` loop (in most of the programs for that matter). We address this limitation in MonoCoder by implementing a code pre-processing scheme that eliminates any local semantics that the LLM may capture. Our experimental evaluation demonstrates that MonoCoder outperforms existing code LMs in virtually all of the code completion settings and context lengths, with and without applying local semantics elimination (henceforth, LSE). Moreover, and in contrast to MonoCoder, the performance of other LMs degraded considerably when LSE was applied, suggesting their reliance on local semantics.

Contributions. This paper makes following contributions:

- By drawing insights from the limitations of existing LLMs, we propose a domain-specific, small language model, called MonoCoder, for tasks related to high-performance computing.
- We design a pre-processing method, called Local Semantics Elimination (LSE), that eliminates syntactical constructs that could lead to local semantics.
- We compare MonoCoder against PolyCoder and GPT-3.5 for general-purpose programming tasks as well as tasks related to HPC programming.
- Finally, we also measure the parallel code generation performance of MonoCoder and other LLMs over a dataset of 20k OpenMP codes by calculating the CodeBLEU score. Our results demonstrate that MonoCoder, although orders of magnitude smaller in size, performs similar to these LLMs in language comprehension, while outperforming them in HPC-specific tasks.

	Repos	Size (GB)	Files (#)	Functions (#)
C	144,522	46.23	4,552,736	87,817,591
C++	150,481	26.16	4,735,196	68,233,984

TABLE I: Statistics on the subset of HPCorpus dataset [27] that we used in our study: ∼300k repos, ∼70 GB, ∼9M files, and ∼155M functions across C and C++ code from GitHub.

The remainder of the paper is organized as follows: In section II we delve into the details of the HPC code dataset HPCorpus. Then section III provides an in-depth exploration of MonoCoder, our language model pre-trained on HPCorpus. In section IV, we introduce LSE, our code preprocessing step designed for HPC code. In section V, we perform evaluations of MonoCoder and state-of-the-art models on non-HPC and HPC tasks. Finally, we conclude the paper and outline directions for the future research in section VII.

II. HPCorpus: HPC Code Corpus

In order to build an HPC-specific model, we first decided to gather a dataset of HPC specific programs. Towards that end, we compiled HPCorpus, a dataset of publicly-visible C and C++, and Fortran programs from GitHub [27]. In this work we decided to focus on C and C++ languages only, and hence we used a subset of HPCorpus as shown in Table I.

In [27], we discovered that many of those repos employed one (or more) parallel API. For instance, 45% employed shared memory parallelism with OpenMP (primarily for threading, but also for SIMD and GPU offloading), 27% employed distribution with MPI, 21% employed direct GPU programming (half with CUDA and half with OpenCL), and the rest mainly employed other threading APIs (such as TBB and Cilk).

Training data preprocessing. While many code LMs include natural language in their pre-training data, this will likely be unhelpful for HPC programmers and downstream HPC tasks of interest. On the other hand, it is much more critical that the model understands the structure of the code. To this end, we preprocess the code files in HPCorpus such that no natural language is included; only structured blocks from deduplicated files are included; and code blocks are greater than 100 tokens and less than 1MB (as done in PolyCoder). By that, we gathered ∼155M functions that are more suitable for pre-training a code language model that emphasizes concise code structure.

```c
// Source code:
int main() {
  int r[2800 + 1];
}
```

→

```c
// LSE:
int func_252() {
 int arr_88[num_34 + num_842];
}
```

→

```
// Lexicalized tokens:
["int", "func", "_", "252",
"()", "{", "int", "arr", "_"
... (tokens continue)
```

Fig. 1: Local Semantics Elimination (LSE) pipeline overview: Given a source code, the code turns into a semantic-less version using AST knowledge, and eventually, the lexicalized tokens are fed into MONOCODER.

III. MONOCODER: AN HPC-SPECIFIC CODE LM

To create a domain-specific model for HPC, we pre-trained a decoder-only transformer model on the language modeling objective (i.e., given past tokens as context, predict the next token in the code) using only C and C++ code from HPCorpus. We named this domain-specific model as MONOCODER, in contrast to the multilingual Polycoder.

Model size. In the pursuit of deploying domain-specific small language models on average computer systems, careful consideration of model size becomes paramount. To ensure compatibility with such resource constraints, we aimed to design a model that strikes a balance between complexity and memory efficiency. The choice of a model comprising just under 1B parameters emerged as an optimal compromise. The decision is underpinned by the following calculations: Assuming each parameter is represented as a 32-bit floating-point number requiring 4 bytes, the formula `RAM size (in bytes) = Number of Parameters` $\times$ `Size of one parameter (in bytes)`. Consequently, for 0.9B parameters, the resulting RAM size amounted to 3.6GB. This selected model size not only accommodates a 4GB RAM constraint – typically the maximum memory per core in HPC systems [28] – but also leaves additional headroom for other computational processes within the system to accommodate model inference. Note that this worst-case analysis did not consider inference-time optimizations such as quantization, or pruning, which would reduce memory consumption further.

We constructed MONOCODER model of 0.9B parameters by reusing the PolyCoder 2.7B model architecture [24], but reducing the number of layers. This led to an 8-layer model with a hidden dimension of 2560 and 32 attention heads per layer. We used the same tokenizer vocabulary (50,257 tokens). This resulted in an 889.3M parameter model ($\approx$ 0.9B parameters).

Limitations. Our domain-specific model inherits the principles of a left-to-right language model, which is especially amenable to code generation tasks. By adhering to the left-to-right nature, our model aligns with the established models such as CodeGPT (124M) [29], GPT-Neo [30] and PolyCoder (2.7B) [24], GPT-J (6B) [31], Codex (12B) [32], StarCoder (15.5B) [33], and GPT-NeoX (20B) [34], which are known for their proficiency in code completion.

However, we acknowledge the inherent challenge posed by the left-to-right approach, which limits the model's ability to consider context beyond the immediate token sequence. As part of our ongoing research, we are exploring strategies to address this limitation and further refine MONOCODER's capability to leverage broader contextual information, thereby advancing its effectiveness in the nuanced landscape of HPC code synthesis.

Pre-training details. We pre-trained the MONOCODER model using the GPTNeoX framework [35] on 4 NVIDIA A40 48GB GPUs with `fp16` precision. For pre-training, we used the Adam optimizer and followed a learning rate schedule with a linear warmup for the first 1% of steps and a cosine decay over the remaining steps. Gradients were clipped at 1.0. Each training sample had a maximum length of 2048 tokens and was trained in mini-batches of 16 samples (4 per GPU). The model was trained for 160K steps at a learning rate of 0.00008. The training and validation losses initially start high at around 3.0, rapidly decrease over the first 25,000 steps, stabilize thereafter, converge closely without overfitting, and finally settle below 0.5 by approximately 150,000 steps, indicating minimal loss.

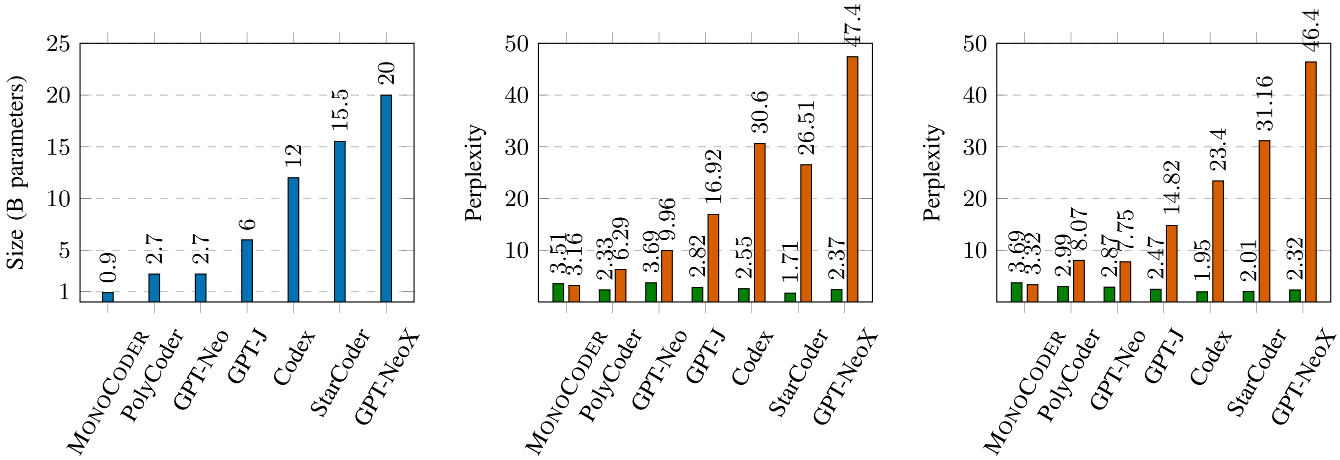
IV. LOCAL SEMANTICS ELIMINATION (LSE)

An effective code LM for HPC tasks must understand both the syntax and structure of the code, without memorizing potentially misleading human semantics [36] (e.g., variable names). We propose a preprocessing method based on an abstract syntax tree (AST) to remove misleading semantic information. A code LM that performs well on this anonymized preprocessed code is clearly relying on its understanding of code structure, rather than natural language-derived semantics. We name this preprocessing method as LSE (for local semantics elimination). LSE is inspired by input tokenization and more importantly ensures functionally correct and compilable code.

Tokenizing code for LLMs necessitates specialized techniques to accommodate programming language syntax.[1] LLMs geared towards code comprehension, such as GPT-3.5-Turbo for code, likely combine several techniques, prioritizing syntax-aware tokenization to effectively process and generate code snippets in various programming languages and tasks.

In short, LSE preprocessing (Figure 1) replaces variable names, numbers, and strings with random variable names and removes superfluous input (e.g., comments, extra whitespace). The detailed steps are as follows:

[1] These approaches include utilizing BPE and subword tokenization akin to natural language [37], employing syntax-aware tokenization to identify language-specific elements like keywords and identifiers [38], constructing tokens based on the AST to capture structural information [39], implementing language-specific lexers following grammar rules [40], preserving character integrity with character-level tokenization [41], tailoring tokenization to unique syntax rules, and leveraging dedicated code tokenization libraries.



(a) Model Size

(b) Perplexity (in green) and Normalized -to-size Perplexity (in orange) for C

(c) Perplexity (in green) and Normalized -to-size Perplexity (in orange) for C++

Fig. 2: Comparison of code language models based on their model size, perplexities, and normalized-to-size perplexities for C and C++. The results demonstrate that smaller models, such as MonoCoder, tend to have much better normalized perplexity scores (lower is better), indicating better performance relative to their size. Data for PolyCoder, GPT-Neo, GPT-J, Codex, StarCoder, and GPT-NeoX are taken from [24] and [33].

1) *AST Generation*: Parse the code using TreeSitter[2] or any suitable parser to generate an AST.
2) *Generate Replaced Code*: Create a version of the original code with anonymized variable names, numbers, and strings. The intuition behind this step is to eliminate misleading semantics, such as variable `i` being an index variable of a `for` loop in the C language.
3) *AST to Code*: Transform the updated AST back into code, while eliminating any comments that may interfere with anonymization.
4) *Random Number Attachment*: For recurrent tokens (e.g., `var_1` or `num_2`), attach random integers from a predefined range (e.g., 1 to 1000) during tokenization. The attached numbers are randomly chosen without any relation to the type or order of the replaced tokens or the file/function length. This step also eliminates misleading semantics. For instance, if variable `i` is consistently replaced with `var_1`, then the model may learn that `var_1` is an index variable of `for` loops.

V. MonoCoder Evaluation

A. Language Modeling Evaluation

We start by measuring the perplexity of MonoCoder and other LLMs on unseen test programs in C and C++.

Metric: Perplexity. Perplexity is a metric commonly employed in natural language processing and language modeling to assess the efficacy of a probabilistic model in predicting a given sequence of tokens. It serves as a measure of the model's uncertainty or confusion when assigning probabilities to a set of observed data. A lower perplexity value signifies the better predictive performance of a model. In the context of our research, we measured perplexity to evaluate the predictive capabilities of the pre-trained MonoCoder language model. That is, this assessed how well the model assigned probabilities to the token sequences in the C and C++ portions of the HPCorpus test set.

Setup. For this test, we use the training procedure outlined earlier to pre-train MonoCoder on HPCorpus. We then measure perplexity on C and C++ code from the test set of HPCorpus. We then compared this perplexity to the perplexity scores of various pre-trained LLMs based on the published works [24], [33]. The StarCoder values are with a 2K context window, similar to MonoCoder's 2048 context window.

Results. The perplexity comparison can be seen in Figure 2. We can see that despite the much smaller size of MonoCoder, it suffers from minor performance degradation than much larger models. For C language, it is performing better than the 3x-larger GPT-Neo model.

B. Code Completion Evaluation

In this second evaluation, we assess the ability of MonoCoder and other LLMs in completing a block of code when provided with varying amounts of prior context. While perplexity provides information about the uncertainty of the model, it does not necessarily measure the quality of the generated code. For this reason, we further evaluate code understanding with the CodeBLEU [42] score.

Metric: CodeBLEU score. CodeBLEU [42] is the metric of choice in the code completion tasks. CodeBLEU amalgamates the robustness of BLEU score [43](from NLP), incorporating n-gram matching, with an innovative integration of code syntax and semantics through AST and data-flow structures. This holistic approach provides a nuanced evaluation that

[2] https://github.com/tree-sitter/tree-sitter

```
// INPUT to LLM:

int main (int argc,
    char *argv[]) {

// LLM COMPLETION:

int i;
int n = argc - 1;
int m = argc - 2;
int k = argc - 3;
int l = argc - 4;
int r = argc - 5;
int c = argc - 6;
int d = argc - 7;
int e = argc - 8;
int f = argc - 9;
int g = argc - 10;
int h = argc - 11;
int i1 = argc - 12;
int i2 = argc - 13;
```

(a) Given function signature

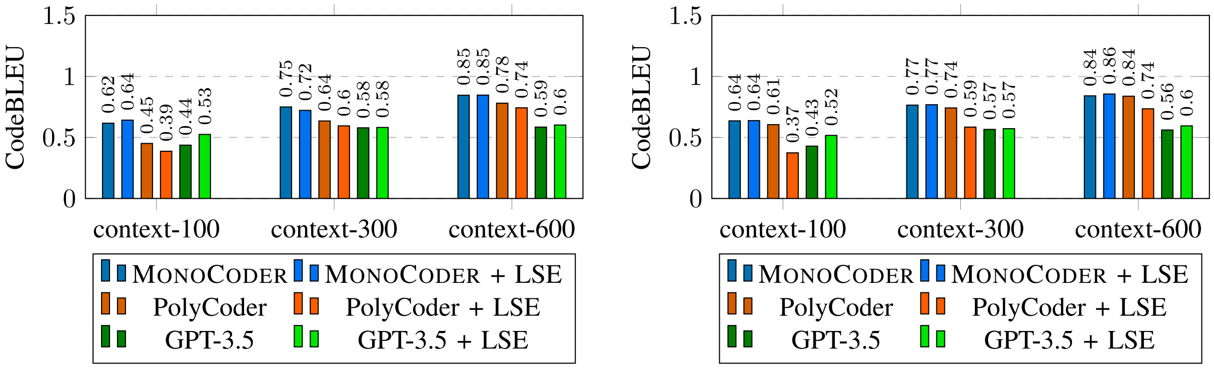
```
// INPUT to LLM:

int main (int argc, char *argv[]) {
int i,j;
double x;
double pi, sum = 0.0;
double start, delta;
step = 1.0/(double) steps;
for (j=1; j<= MAX_THREADS; j++) {
printf("running on %d threads: ", j);
omp_set_num_threads(j);

// LLM COMPLETION:

start = omp_get_wtime();
for (i=0; i<steps; i++) {
x = (double) i;
pi = 3.141592653589793238
4626433832795028841971693
9937510582097494459230781
6406286208998628034825342...
```

(b) Given initialization

```
// INPUT to LLM:

int main (int argc, char *argv[]) {
int i,j;
double x;
double pi, sum = 0.0;
double start, delta;
step = 1.0/(double) steps;
for (j=1; j<= MAX_THREADS; j++) {
printf("running on %d threads: ", j);
omp_set_num_threads(j);
sum = 0.0;
double start = omp_get_wtime();
#pragma omp parallel for
    reduction(+:sum) private(x)

// LLM COMPLETION:

for (i=0; i<steps; i++) {
   x = (double) i;
   sum += x;} ...
```

(c) Given context of parallel execution

Fig. 3: Evaluating code completion performance of HPC code by the foundation models (PolyCoder example): Evaluating machine-generated code, given different contexts of the initial HPC codes and measuring the similarity to the true reference.



(a) Code completion performance on *General* dataset.

(b) Code completion performance on *OpenMP* dataset.

Fig. 4: Code Completion Performance on General and OpenMP Datasets — CodeBLEU scores (higher is better) for MonoCoder, PolyCoder, and GPT-3.5 models, both with and without Local Semantic Elimination (LSE), across varying context lengths (100, 300, and 600 tokens). MonoCoder and MonoCoder + LSE consistently outperform other models, with the addition of LSE generally enhancing performance across all models.

extends beyond token matching, considering the significance of keywords, syntactic accuracy, and semantic correctness. In the case of HPC code without natural language, CodeBLEU is the appropriate metric to check similarity to generations.

Setup. For this evaluation, we first devise two sub-datasets: (1) *General* dataset of about 20k examples of C and C++ programs from HPCorpus with HPC-orientation[3] and (2) *OpenMP* dataset of about 20k examples of C and C++ programs containing OpenMP code (e.g., `parallel for`). The method to test the model's understanding is by incrementally supplying it more parts of those programs (first 100, 300, and 600 tokens, out of an average of 1200 tokens per code) and comparing the code generated by the model with the expected ground-truth code using CodeBLEU. Demonstration of the idea is presented in Figure 3.

Results. Figure 4a shows code completion performance of the models on *General* dataset. For a context of the first 100 tokens, MonoCoder achieves a CodeBLEU score of 0.62, while PolyCoder and GPT-3.5 attain scores of 0.45 and 0.44, respectively. This discrepancy can be attributed to PolyCoder's diverse training across programming languages and GPT-3.5's general nature, resulting in versatile but less HPC-specific answers. Expanding the context window to 600 tokens further widens the CodeBLEU score gap between MonoCoder and GPT-3.5, emphasizing lack of HPC knowledge in GPT. Overall, the model's reduced orientation towards HPC codes accentuates the discernible gap in HPC code knowledge among the models.

[3] Ones that contained OpenMP's `parallel for` pragmas. However, those pragmas were intentionally removed to keep clean C and C++ programs that have HPC orientation

Similar trends are evident on *OpenMP* dataset in Figure 4b, depicting the CodeBLEU scores of models when completing functions containing OpenMP pragmas. Generally, the results are slightly lower than those on *General* dataset, underscoring the models' limited understanding of OpenMP.

It is noteworthy that, despite the LSE pre-processing having a minor impact on the performance of MonoCoder and GPT-3.5, this code representation significantly impairs the performance of PolyCoder. This observation suggests that PolyCoder heavily relies on local semantics for code completion.

VI. Related Work

We present the related work along two directions: 1) code LMs that are not specifically designed for HPC tasks but can solve HPC tasks, 2) LLMs designed specifically for HPC tasks.

The code LM literature is somewhat divided into NLP-oriented approaches and software engineering approaches. Popular LLMs such as GPT-3.5, GPT-4, LLaMa, or even code specific LLMs such as CodeT5 [44], etc., are trained on code datasets that also contain natural language comments. As a result, most of these models, if not all, can solve programming tasks with or without natural language prompts. These NLP-oriented approaches typically evaluate code LMs with perplexity and datasets like HumanEval [32] or Mostly Basic Programming Problems [45], which assess both natural language comprehension and general reasoning ability of a model along with code comprehension. These models also solve coding-specific tasks such as code search, code completion, etc. In addition to natural language and programming-specific tasks, these models can also solve HPC tasks to certain extent. Because the training datasets of these models most likely contain HPC code also (in APIs such as OpenMP, CUDA, etc.), these models possess some understanding of HPC languages also. Our experiments with GPT-3.5 revealed that it had a reasonable understanding of code parallelization problem (e.g., analyzing if a loop can be parallelized). However, our experiments also revealed that these LLMs have limited understanding of HPC tasks — the same observation also reported by Nichols et al. in their evaluation of LLMs for HPC tasks [23]. Specifically, their experimental results reveal several limitations of existing LLMs for HPC tasks.

Given the limitations of popular LLMs on HPC tasks and also their expensive training costs, several works have recently explored HPC specific LMs. Specifically, several groups have developed LMs for popular HPC problems such as OpenMP pragma prediction and generation [8]–[13], [26], [46], MPI code generation [14], [47], and race detection [25]. We found that these works mostly finetune a pre-trained LM, which are larger in size than required for their HPC tasks. We believe that our work complements these papers by demonstrating a smaller, domain-specific LM that can solve HPC tasks.

VII. Conclusion & Future Work

Existing LLMs, such as GPT-3.5, are multi-lingual and are trained on languages unrelated to HPC. This phenomenon leads to huge model sizes and demands expensive compute resources to train. Thus, we decided to evaluate if we can build a smaller, domain-specific model that can perform similar, if not better, than existing LLMs on HPC-related programming tasks. Towards that end, we built MonoCoder using an existing code-oriented LLM but by carefully selecting the model size such that the resulting model can fit on a commodity hardware. Our experimental results demonstrate that MonoCoder, although orders of magnitude smaller in size than existing LLMs, performs similar to the existing LLMs in terms of language comprehension task (perplexity score), while outperforming them in code completion task (CodeBLEU score) for general-purpose and HPC-specific C and C++ programs.

In the near future, we intend to integrate additional code representations, such as the data-flow graph (DFG) and the intermediate representation (IR) [48], to enhance model understanding as shown in the closely related works [49], [50]. Moreover, we believe that pre-training on a compilable subset of HPCorpus will enhance the performance of the model for compilation-oriented tasks (as partially demonstrated in [51]). Then, we intend to fine-tune those pre-trained models for HPC-specific downstream tasks, such as OpenMP pragma generation [12], [13], [46] and MPI domain decomposition distribution [13], [14], [47]. In general, our research vision is to systematically analyze each and every element of existing LLMs (model architecture, dataset, etc.) and redesign them as needed for HPC-specific tasks.

References

[1] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Computing Surveys*, 2021.

[2] L. Floridi and M. Chiriatti, "GPT-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[3] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[4] OpenAI, "OpenAI ChatGPT," https://openai.com/blog/chatgpt, 2023, [Online].

[5] D. Reed, D. Gannon, and J. Dongarra, "Reinventing high performance computing: challenges and opportunities," *arXiv preprint arXiv:2203.02544*, 2022.

[6] J. Dongarra, "HPC: Where we are today and a look into the future," *Parallel Processing and Applied Mathematics, PPAM: Gdansk, Poland*, 2022.

[7] D. Reed, D. Gannon, and J. Dongarra, "HPC Forecast: Cloudy and Uncertain," *Communications of the ACM*, vol. 66, no. 2, pp. 82–90, 2023.

[8] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, and B. de Supinski, "LM4HPC: Towards Effective Language Model Application in High-Performance Computing," *arXiv preprint arXiv:2306.14979*, 2023.

[9] L. Chen, Q. I. Mahmud, H. Phan, N. Ahmed, and A. Jannesari, "Learning to Parallelize with OpenMP by Augmented Heterogeneous AST Representation," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[10] R. Harel, Y. Pinter, and G. Oren, “Learning to parallelize in a shared-memory environment with transformers,” in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 450–452.

[11] T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, G. Oren *et al.*, “PragFormer: Data-driven Parallel Source Code Classification with Transformers,” *arXiv*, 2023.

[12] T. Kadosh, N. Schneider, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, “Advising openmp parallelization via a graph-based approach with transformers,” *arXiv preprint arXiv:2305.11999*, 2023.

[13] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, “Modeling Parallel Programs using Large Language Models,” *arXiv preprint arXiv:2306.17281*, 2023.

[14] N. Schneider, T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, “MPI-RICAL: Data-Driven MPI Distributed Parallelism Assistance with Transformers,” in *Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2–10. [Online]. Available: https://doi.org/10.1145/3624062.3624063

[15] Y. Shen, M. Peng, Q. Wu, and G. Xie, “Multigraph learning for parallelism discovery in sequential programs,” *Concurrency and Computation: Practice and Experience*, vol. 35, no. 9, p. e7648, 2023.

[16] L. Chen, N. K. Ahmed, A. Dutta, A. Bhattacharjee, S. Yu, Q. I. Mahmud, W. Abebe, H. Phan, A. Sarkar, B. Butler *et al.*, “Position Paper: The Landscape and Challenges of HPC Research and LLMs,” *arXiv preprint arXiv:2402.02018*, 2024.

[17] R. Harel, I. Mosseri, H. Levin, L.-o. Alon, M. Rusanovsky, and G. Oren, “Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential,” *International Journal of Parallel Programming*, vol. 48, pp. 1–31, 2020.

[18] R. Milewicz, P. Pirkelbauer, P. Soundararajan, H. Ahmed, and T. Skjellum, “Negative Perceptions About the Applicability of Source-to-Source Compilers in HPC: A Literature Review,” in *International Conference on High Performance Computing*. Springer, 2021, pp. 233–246.

[19] I. Mosseri, L.-o. Alon, R. Harel, and G. Oren, “ComPar: optimized multi-compiler for automatic OpenMP S2S parallelization,” in *OpenMP: Portable Multi-Level Parallelism on Modern Systems: 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings 16*. Springer, 2020, pp. 247–262.

[20] S. Prema, R. Jehadeesan, and B. Panigrahi, “Identifying pitfalls in automatic parallelization of NAS parallel benchmarks,” in *Parallel Computing Technologies (PARCOMPTECH), 2017 National Conference on*. IEEE, 2017, pp. 1–6.

[21] S. Prema, R. Nasre, R. Jehadeesan, and B. Panigrahi, “A study on popular auto-parallelization frameworks,” *Concurrency and Computation: Practice and Experience*, vol. 31, no. 17, p. e5168, 2019.

[22] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large Language Models for Software Engineering: A Systematic Literature Review,” *arXiv preprint arXiv:2308.10620*, 2023.

[23] D. Nichols, J. H. Davis, Z. Xie, A. Rajaram, and A. Bhatele, “Can Large Language Models Write Parallel Code?” Jan. 2024.

[24] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, “A systematic evaluation of large language models of code,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[25] X. Ding, L. Chen, M. Emani, C. Liao, P.-H. Lin, T. Vanderbruggen, Z. Xie, A. E. Cerpa, and W. Du, “HPC-GPT: Integrating Large Language Model for High-Performance Computing,” *arXiv preprint arXiv:2311.12833*, 2023.

[26] Q. I. Mahmud, A. TehraniJamsaz, H. D. Phan, N. K. Ahmed, and A. Jannesari, “AUTOPARLLM: GNN-Guided Automatic Code Parallelization using Large Language Models,” *arXiv preprint arXiv:2310.04047*, 2023.

[27] T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, “Quantifying openmp: Statistical insights into usage and adoption,” 2023.

[28] A. Khan, H. Sim, S. S. Vazhkudai, A. R. Butt, and Y. Kim, “An analysis of system balance and architectural trends based on top500 supercomputers,” in *The International Conference on High Performance Computing in Asia-Pacific Region*, 2021, pp. 11–22.

[29] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” *arXiv preprint arXiv:2102.04664*, 2021.

[30] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, “Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow,” *If you use this software, please cite it using these metadata*, vol. 58, 2021.

[31] B. Wang and A. Komatsuzaki, “Gpt-j-6b: A 6 billion parameter autoregressive language model,” 2021.

[32] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.

[33] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.

[34] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang *et al.*, “Gpt-neox-20b: An open-source autoregressive language model,” *arXiv preprint arXiv:2204.06745*, 2022.

[35] A. Andonian, Q. Anthony, S. Biderman, S. Black, P. Gali, L. Gao, E. Hallahan, J. Levy-Kramer, C. Leahy, L. Nestler, K. Parker, M. Pieler, J. Phang, S. Purohit, H. Schoelkopf, D. Stander, T. Songz, C. Tigges, B. Thérien, P. Wang, and S. Weinbach, “GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch,” 9 2023. [Online]. Available: https://www.github.com/eleutherai/gpt-neox

[36] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, “What do code models memorize? an empirical study on large language models of code,” *arXiv preprint arXiv:2308.09932*, 2023.

[37] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” *arXiv preprint arXiv:1508.07909*, 2015.

[38] W. Zheng, S. Sharan, A. K. JAISWAL, K. Wang, Y. Xi, and Z. Wang, “Code means more than plain language: Bringing syntax structure awareness to algorithmic problem solution generation,” *arXiv*, 2022.

[39] Y. Xu and Y. Zhu, “A survey on pretrained language models for neural code intelligence,” *arXiv preprint arXiv:2212.10079*, 2022.

[40] N. D. Bui, H. Le, Y. Wang, J. Li, A. D. Gotmare, and S. C. Hoi, “Codetf: One-stop transformer library for state-of-the-art code llm,” *arXiv preprint arXiv:2306.00029*, 2023.

[41] D. KC and C. T. Morrison, “Neural machine translation for code generation,” *arXiv preprint arXiv:2305.13504*, 2023.

[42] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “Codebleu: a method for automatic evaluation of code synthesis,” *arXiv preprint arXiv:2009.10297*, 2020.

[43] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: A Method for Automatic Evaluation of Machine Translation,” in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, P. Isabelle, E. Charniak, and D. Lin, Eds. Association for Computational Linguistics, 2002, pp. 311–318.

[44] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[45] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program Synthesis with Large Language Models. [Online]. Available: http://arxiv.org/abs/2108.07732

[46] L. Chen, A. Bhattacharjee, N. Ahmed, N. Hasabnis, G. Oren, V. Vo, and A. Jannesari, “OMPGPT: A generative pre-trained transformer model for openmp,” *arXiv preprint arXiv:2401.16445*, 2024.

[47] N. Schneider, N. Hasabnis, V. A. Vo, T. Kadosh, N. Krien, M. Capotă, A. Wasay, G. Tamir, T. Willke, N. Ahmed *et al.*, “MPIrigen: MPI Code Generation through Domain-Specific Language Models,” *arXiv preprint arXiv:2402.09126*, 2024.

[48] A. Grossman, L. Paehler, K. Parasyris, T. Ben-Nun, J. Hegna, W. Moses, J. M. M. Diaz, M. Trofin, and J. Doerfert, “ComPile: A Large IR Dataset from Production Sources,” 2023.

[49] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, “Graphcodebert: Pre-training code representations with data flow,” *arXiv preprint arXiv:2009.08366*, 2020.

[50] M. Szafraniec, B. Roziere, H. Leather, F. Charton, P. Labatut, and G. Synnaeve, “Code translation with compiler representations,” *arXiv preprint arXiv:2207.03578*, 2022.

[51] L. Chen, A. Bhattacharjee, N. K. Ahmed, N. Hasabnis, G. Oren, B. Lei, and A. Jannesari, “Compcodevet: A compiler-guided validation and enhancement approach for code dataset,” *arXiv preprint arXiv:2311.06505*, 2023.