HPC-Coder: Modeling Parallel Programs using Large Language Models

Daniel Nichols[†], Aniruddha Marathe[*], Harshitha Menon[*], Todd Gamblin[‡], Abhinav Bhatele[†]

[†]*Department of Computer Science, University of Maryland, College Park, MD, USA*
[*]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA*
[‡]*Livermore Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA*
Email: dnicho@umd.edu, {marathe1, gopalakrishn1, tgamblin}@llnl.gov, bhatele@cs.umd.edu

***Abstract*—Parallel programs in high performance computing (HPC) continue to grow in complexity and scale in the exascale era. The diversity in hardware and parallel programming models make developing, optimizing, and maintaining parallel software even more burdensome for developers. One way to alleviate some of these burdens is with automated development and analysis tools. Such tools can perform complex and/or remedial tasks for developers that increase their productivity and decrease the chance for error. Until recently, such tools for code development and performance analysis have been limited in the complexity of tasks they can perform, especially for parallel programs. However, with recent advancements in language modeling, and the availability of large amounts of open-source code related data, these tools have started to utilize predictive language models to automate more complex tasks. In this paper, we show how large language models (LLMs) can be applied to tasks specific to high performance and scientific codes. We introduce a new dataset of HPC and scientific codes and use it to fine-tune several pre-trained models. We compare several pre-trained LLMs on HPC-related tasks and introduce a new model, *HPC-Coder*, fine-tuned on parallel codes. In our experiments, we show that this model can auto-complete HPC functions where generic models cannot, decorate `for` loops with OpenMP pragmas, and model performance changes in scientific application repositories as well as programming competition solutions.**

***Index Terms*—large language models, parallel code generation, performance modeling**

I. Introduction

In recent years, large language models (LLMs) have become the state of the art for many language modeling related tasks [1]. Their ability to model token probabilities within a sequential context make them desirable for language tasks such as text generation and sequence classification. In addition to being used for natural language, such models have recently been applied to many programming language related tasks [2]–[4]. The predictive capabilities of these models translate well to coding tasks, and the wealth of open-source code available online provides significant data for training large models.

LLMs trained on source code data have been utilized to automate numerous software development tasks such as code completion, malware detection, code refactoring, etc [3]–[12]. Additionally, they have been able to automate tasks previously considered impossible to automate such as code summarization and generation using natural language. Training LLMs for these tasks requires significant amounts of source code data that is fortunately available online from open-source code repositories on GitHub, gitlab etc. However, this data requirement for training LLMs is prohibitive for tasks where such data may not exist. One such task is that of modeling performance (execution time) based on source code. Another difficult task is modeling parallel and HPC code where there is less data available and it is often more complex code.

Performance data for arbitrary code is difficult to obtain at scale with large numbers of samples. First and foremost, it is non-trivial to automate the collection of performance data for arbitrary source code. The code needs to be built and run in order to measure performance, and this process can vary significantly across repositories. This can be particularly difficult for production scientific codes due to code complexity, dependence on external libraries, and the fact that it often needs to be run in parallel with many resources. Second, performance depends on numerous variables besides just the code such as input problem, architecture, and current machine load/congestion. These either need to be fixed in the dataset or accounted for within the modeling pipeline. Finally, source code needs to be considered holistically when modeling performance, since minor changes in one place may drastically impact performance elsewhere. For example, changing the data layout within a data structure will impact the performance of data access where that structure is used. This means that the entirety of the source code needs to be included in the dataset and performance needs to be collected at a finer granularity.

When a lack of data becomes a hurdle in machine learning tasks, it is typically solved through data augmentation and/or transfer learning. Data augmentation involves extending and/or duplicating data in a manner that still preserves meaning and representational capacity. Transfer learning is done by first training a model on a related or simpler task and then *transferring* that knowledge to a new problem requiring fewer samples to learn. For our task we employ transfer learning by using LLMs that have learned to model source code and then transferring that knowledge to then learn how to model performance of source code using fewer samples. In particular, we explore modeling parallel and HPC codes.

In this paper, we utilize LLMs to model high performance and scientific codes, and then apply that to the problem of performance modeling. In order to accomplish this, we

introduce a new dataset of HPC and scientific codes from popular open-source repositories. We first demonstrate how our trained model, *HPC-Coder*, outperforms other LLMs on HPC specific tasks such as code generation and OpenMP pragma labeling. A set of code generation tests specific to HPC are introduced and the model can pass these at up to 53% higher rate than the other models. Additionally, it is able to label `for` loops with OpenMP pragmas with 97% accuracy. Finally, we demonstrate how the model can predict relative performance of source code changes with up to 92% accuracy. In summary, this paper makes the following contributions:

- A large curated dataset containing HPC and scientific code from numerous open-source repositories.
- We present an LLM, *HPC-Coder*, fine-tuned to model HPC and scientific code. We show that it trains to better language modeling scores over HPC related code than other state-of-the-art models.
- We introduce a set of HPC code generation tasks and demonstrate that our model completes these tasks at a significantly better rate than other models on HPC-specific code.
- We demonstrate how our model can be used to predict OpenMP pragmas with high accuracy.
- We utilize our model to predict relative performance of source code changes for two distinct datasets from scientific application repositories and coding competition solutions.

II. Background

This section provides background on transformer-based language models and how they can be applied to source code.

A. Large Language Models

When applying machine learning to textual data we need a model that takes text as input and, through the process of training on previous data, learns how to predict some property of that text. In recent years such models have been mostly dominated by large transformer-based models. Transformers were first introduced by Vaswani et al. [13]. They are designed to work with sequential data much like recurrent and long short-term memory neural networks. However, they differ in their use of a self-attention mechanism to attribute importance weights to inputs into the model. Due to this mechanism transformers also process entire sequences at once unlike recurrent neural networks.

These self-attention units make up the basis of transformer networks. Weights are divided into query, key, and value weights (namely W_Q, W_K, W_V). These are multiplied by each input token i and stacked to form the matrices Q, K, and V, respectively. Given these matrices and the dimensions of the key vector d_k the attention can be computed as below.

$$\text{Attention}\left(Q,K,V\right) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

These weight matrices form a single attention head. Typically transformers employ several attention heads to form a multi-attention head layer. Having multiple attention heads allows each of them to learn, or *attend to*, different abstractions in the input, such as parts-of-speech for natural language input.

Generally these networks are trained to model the conditional probability of observing a language token or a sequence of tokens. For instance, given a string of observed tokens $t_1 t_2 \dots t_{i-1}$ we may want the most likely next token t_i.

$$t_i = \arg\max_t P\left(t_i = t \mid t_1 t_2 \dots t_{i-1}\right)$$

Similarly we may want to know the probability of a sequence of tokens occurring given the entire observed dataset $P\left(t_1, t_2, \dots, t_N\right)$ (i.e. how likely is a given english sentence to be real given my previous knowledge of the language). Using this probability we can define a metric called *perplexity*.

$$\text{Perplexity}(T) = \left(\frac{1}{P(t_1, t_2, \dots, t_N)}\right)^{\frac{1}{N}}$$

With this metric a model that scores a lower perplexity on its test set T is better as it assigns a higher probability to the test data. The ratio is normalized to be invariant to the size of the test set. Rewriting the formula for perplexity we can see that it is equivalent to the exponential of the cross-entropy.

$$\begin{aligned}\text{Perplexity}(T) &= \left(P(t_1, t_2, \dots, t_N)\right)^{-\frac{1}{N}} \\ &= \left(\exp\log P(t_1, t_2, \dots, t_N)\right)^{-\frac{1}{N}} \\ &= \exp\left(-\frac{1}{N}\log P(t_1, t_2, \dots, t_N)\right)\end{aligned}$$

This allows us to train the language model with cross-entropy loss. Minimizing the loss will, in turn, minimize the perplexity. The perplexity is recovered by simply taking the exponential of the loss. It is important to note that perplexity measures model confidence and not accuracy. However, it has been demonstrated empirically that lower perplexity generally leads to better performance on downstream tasks.

B. Text Generation

A trained model can then be used to generate new text. Since the LLM models token probability it may seem simple to select the most probable next token, however, this can lead to poor text generation. Often a model's attention puts more focus on on the most recent tokens causing this selection method to get stuck in loops or suddenly forget context. Most recent works combat this issue by *sampling* from the model's distribution, but there are several important caveats when doing this. For instance, we want to avoid sampling from the tail as this could drastically throw off further tokens sampled. Here we discuss several of the sampling methods used later in this paper such as temperature, top-k, and nucleus sampling.

Temperature: When sampling temperature controls how *confident* the model is in the sampled token. Lower temperature leads the model to assign more confidence in the most likely tokens in the distribution. On the other end, the model will more uniformly assign confidence across the distribution when the temperature is higher. This term comes from statistical

thermodynamics where lower energy states are more frequent with a higher temperature.

Temperature is incorporated by dividing the *logits* by the temperature, $temp$, before computing the softmax output. The *logits* are the raw, un-normalized outputs of the model and the softmax is used to turn this vector into probabilities.

$$\text{softmax}\left(\frac{\boldsymbol{logits}}{temp}\right)$$

Thus, as $temp \to 0$ the output becomes the argmax and as $temp \to \infty$ it leads to a uniform sampling.

Top-k Sampling: In top-k sampling the most likely k tokens are sampled from the model. This aims to exclude the distribution's tail and prevent the model from rapidly getting off-topic. However, this can also reduce the quality of predictions if the body of the distribution is wider than k. A common choice for k is 50.

Nucleus Sampling: Nucleus, or top-p, sampling aims to solve the shortcomings of top-k sampling by choosing a more meaningful cut-off point. In this method the CDF of the distribution is computed and sampling is cut-off when the CDF exceeds p. A common choice for p is 0.9.

C. Using LLMs for Code Generation

LLMs can be trained on a variety of downstream tasks and objectives. When applied to source code data they are typically trained as left-to-right, masked, or encoder-decoder models.

Left-to-Right: Left-to-right or causal language models are trained to predict the most probable next token in a sequence. The model receives and generates text in a left-to-right fashion, which is where it gets its name. This limits the amount of context the model can see as it cannot use later tokens in its prediction even if they are present in the data. Left-to-right models are useful for text generation related tasks.

Masked: Unlike left-to-right models, masked models can predict the most probable token for any position in the text. After removing random tokens in the samples and replacing them with *mask* tokens, the model is trained to predict the most probable tokens to replace the masks with. In this configuration masked models can make use of more context in their predictions.

Encoder-Decoder: Another common approach is to train a left-to-right model to *decode* a sequence after it has been passed through an encoder. This type of model can be combined with several different objectives and is often used with sequence-to-sequence prediction.

To apply left-to-right models, which are focused on in this paper, to source code you simply need to provide the model with prior context as a sequence of tokens and then let it generate new tokens until some stopping threshold. The prior context is typically a natural language comment followed by a function declaration. Tokens are then generated until the function is complete (a closing } bracket in the case of C/C++).

Additionally, when applying language models to code it is typical to customize the training process slightly to take advantage of the syntactic differences between natural language and code. For instance, the tokenizer, which is responsible for mapping text to a sequence of integers, is often set to group whitespace into single tokens. This is not necessary in natural language inputs as multiple consecutive spaces are uncommon. However, in code this can meaningfully reduce the sequence size and a formatter can be applied after code generation to regain formatting.

III. Overview of the Proposed Methodology

Figure 1 provides an overview of the data gathering, training, and downstream application in this paper. In order to train a large HPC-specific language model we need a large dataset of HPC code. To obtain this, we gather a dataset of HPC source code and use it to fine-tune a pre-trained language model. This data gathering is described in Section IV and presents what HPC sources are used and how they are pre-processed. Following this, the model fine-tuning and selection are detailed in Section V where we explain the training setup and methodology.

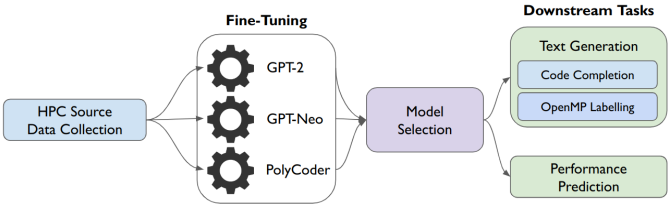

Fig. 1. Overview of the steps described in this paper to train an HPC specific model and run it on several downstream tasks. After collecting a large dataset of HPC code we fine-tune several pre-trained language models and select the best one. The selected model is then used to generate code, label OpenMP pragmas, and predict relative performance as part of several downstream tasks.

We need several realistic tests to study the performance of the language model on relevant metrics. We present three main downstream tasks for evaluation in Section VI. The first two, code generation and OpenMP pragma labeling, test the model on its ability to generate correct and meaningful code. The last test, relative performance prediction, shows how this trained model can be used for useful tasks that require language comprehension. Results from each of these tests are presented and discussed in Section VII.

IV. Data Gathering and Pre-processing

In order to train a large language model to understand and generate HPC code, we need to show it lots of examples. We must first build a dataset to accomplish this. In this section, we detail our collected dataset and how it is processed. We present two additional code datasets paired with performance data for further fine-tuning model performance.

A. HPC Source Code Data

We first collect a sufficiently large dataset of source code to train the model on HPC and scientific code. The HPC source

dataset is collected from GitHub repositories. The source files are pulled from repositories with C/C++ marked as the primary language and with ≥ 3 stars. The repositories are additionally filtered by HPC related GitHub *topics*. Once cloned, we collect all the C/C++ source files based on their file extension.

This dataset is collected and structured in the same manner as the C/C++ source dataset from Xu et al. [14]. Their dataset is scraped from GitHub in a similar manner with the exception of only including repositories with ≥ 5 stars. Figure 2 shows the distribution of lines of code (LOC) by file types in the HPC source dataset. There are roughly the same number of LOC in both C and C++ files. The distribution of actual file counts follows the same trend.

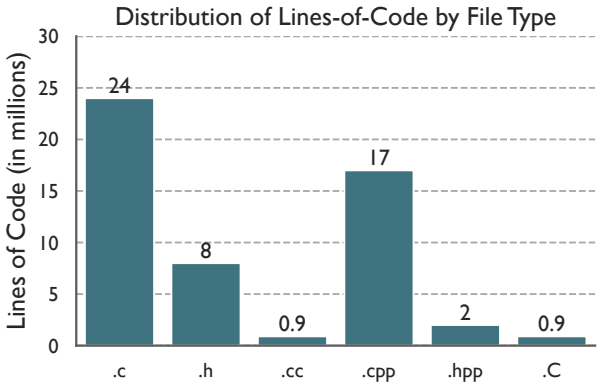

Fig. 2. Distribution of no. of lines of code in each file type. *.cxx*, *.hh*, *.H*, and *.hxx* files are included in the dataset, but omitted here due to small counts.

B. Data Pre-processing

Allamanis [15] shows how duplicate source data, which is prevalent across GitHub repositories, can adversely bias LLMs during training. To prevent this we filter our datasets by removing duplicate files based on the hash of their contents. We use sha256 to hash the contents of the file.

In addition to deduplicating we also filter out small and large files. Source files larger than 1 MB are designated as large files and removed. These are generally entire libraries in a single source file or contain raw data within the code. Additionally, files containing less than 15 tokens, as defined by the language vocab, are not included. The reduced dataset sizes after deduplication and filtering are listed in Table I. Approximately 18% of the files are removed during this processing. Table I shows the properties of the dataset after each step of deduplication and filtering.

TABLE I
PROPERTIES OF THE HPC SOURCE CODE DATASET.

Filter	# Files	# LOC	Size (GB)
None	239,469	61,585,704	2.02
Deduplicate	198,958	53,043,265	1.74
Deduplicate + remove small/large files	196,140	50,017,351	1.62

After filtering source files, we tokenize the dataset to obtain integer values for the text that can be used as input into the model. We use the pre-trained tokenizers for each of our selected models (see Section V). These are all GPT-2 [16] based Byte-Pair Encoding (BPE) tokenizers.

C. Performance Datasets

In addition to the large HPC source code dataset, we create two datasets of code paired with performance data. These datasets contain code pairs with performance data for both codes in the pair, and can be used to train an LLM to model performance characteristics between them.

We create two datasets – one with pairs of code that are functionally different and one where they are the same. The first dataset is created by using version control history to capture performance regressions. We run each commit for the Kripke [17] and Laghos [18] applications. These are small HPC apps meant to mimic the computational behavior of larger scientific applications. We automate building and running each commit to the best of our ability and collect performance results for 830 commits in total.

The second dataset is a set of programming competition solutions from the *code_contests* dataset [19]. These are aggregated from several online programming competitions: Aizu, AtCoder, CodeChef, CodeForces, and HackerEarth. This dataset allows us to create pairs of code that solve the same problem (the contest problem), but may be different in implementation. We run every correct solution for each problem in the dataset, with the corresponding problem's test cases as inputs, and record the run time. Using all the C++ solutions in the dataset we create ~1.7 million samples of code. Using the run times, we group the solutions into pairs and label them as *slower* and *faster* pairs.

V. FINE-TUNING METHODOLOGY

In this section, we describe the models used and how they were selected. We also discuss the methods used to fine-tune them on our collected dataset.

A. Models Selected For Fine-tuning

Recent years have seen the introduction of a significant number of large language models. These models can range in size from 100 million to more than 100 billion parameters. Such large models have been shown to work well for language modeling, but pose significant hurdles to train and use in practice. They can take months to train on large GPU clusters and typically cannot feasibly run inference on consumer-grade hardware. Thus, choosing the right model requires selecting one that can sufficiently model the language data, but also be reasonably deployed for downstream tasks.

Keeping the above mentioned requirements in mind, we select several models for fine-tuning and/or testing. These are listed in Table II. All of these are based on GPT-2 [16] and/or GPT-3 [23] architectures with slight variations in size, configuration, and pre-training data. GPT-2, the smallest in our experiments, is pre-trained on the WebText [20] dataset, which

TABLE II
DESCRIPTION OF THE MODELS USED FOR FINE-TUNING.

Model	# Params.	# Layers	Hidden Size	Window Size	Pre-Training Set
GPT-2 [16]	1.5B	48	1600	1024	WebText [20]
GPT-Neo [21]	2.7B	32	2560	256	Pile [22]
PolyCoder [14]	2.7B	32	2560	2048	Source Code

is a collection of language data scraped from the internet. We use the 1.5 billion parameter GPT-2 model variant in this paper. PolyCoder [14] is pre-trained on a collection of solely source code data from GitHub that contains a mixture of 12 popular programming languages [14]. Between these two is GPT-Neo [21] that is pre-trained on the Pile dataset [22]. This dataset contains a collection of approximately 800GB of text data from the internet, academic articles, source code, etc. Notably this dataset has a mixture of natural language and code. It has been demonstrated that pre-training over *both* natural language and code can improve the performance of the model.

We exclude models such as GPT-4 [24], the state-of-the-art model that powers GitHub CoPilot, from our experiments due to the model and its dataset being closed source. It is currently only accessible for inference via a non-free API. GPT-4's dataset being closed source is significant as we cannot remove data it has trained on from the dataset we use to evaluate its performance, so its results would be overly optimistic. This prevents a realistic evaluation and comparison.

B. Fine-tuning Setup and Hyperparameters

We rely on the functionality provided in the HuggingFace [25] Python library for fine-tuning the models. This library automates many of the tasks related to loading and pre-processing datasets, and running language models on the datasets. In particular, we use the `Trainer` interface with DeepSpeed [26] as the backend to optimize fine-tuning. DeepSpeed is a framework that provides distributed training functionality and several memory optimizations to enable large models to fit in GPU memory.

Starting with the pre-trained models, we fine-tune them on a single node with an AMD EPYC 7763 CPU, 512 GB memory, and four 40 GB NVIDIA A100 GPUs. With DeepSpeed's ZeRO memory optimizations [27], all of the models fit entirely within a single A100 GPU and are, thus, fine-tuned using pure data parallelism. We refer the reader to [28], [29] for a comprehensive overview of training deep neural networks in parallel.

We use the AdamW [30] optimizer for all the models to update model weights and minimize the loss. We set the learning rate to 5×10^{-5} and Adam parameters β_1 and β_2 to 0.9 and 0.999, respectively. These hyperparameters are consistent with typical values in the literature. 16-bit floating point precision is used to accelerate fine-tuning and reduce model size on the A100s. We record the perplexity of the model on the training data during fine-tuning. This is calculated as the exponential of the training loss (see Section II-A). Every 1000 optimizer steps, we also test the model using the validation dataset, and record the perplexity and accuracy at predicting tokens. The validation dataset is 5% of the full dataset, separate from the training dataset.

VI. DOWNSTREAM INFERENCE TASKS AND EVALUATION METRICS

In this section, we introduce the benchmarks and metrics used to evaluate the performance of the language models.

A. Code Completion

A standard benchmark for code generation tasks is the HumanEval benchmark [31]. This is comprised of 164 sample Python problems, where the input to the model is a natural language description of a function and function header. The model generates code for the function implementation, and is scored on functional correctness rather than textual similarity or equivalence.

We introduce our own adaptation of this benchmark for HPC C/C++ programs. Our benchmark consists of 25 custom HPC code generation problems including simple numerics, OpenMP parallel code, and MPI routines. Table III lists the tests used in our evaluation. Figure 3 shows a sample prompt (top) and output (bottom) for a shared-memory parallel implementation of `saxpy`. The prompt is provided as input to the model and it is expected to generate text functionally equivalent to the text on the bottom.

TABLE III
CODE GENERATION TESTS. OPENMP AND MPI COLUMNS DENOTE IF THE TEST INCLUDES A VERSION WITH THAT PARALLEL BACKEND.

Name	Description	Seq.	OpenMP	MPI
Average	Average of an array of doubles	✓	✓	✓
Reduce	Reduce by generic function foo	✓	✓	✓
Saxpy	Saxpy	✓	✓	✓
Daxpy	Daxpy	✓	✓	✓
Matmul	Double-precision matrix multiply	✓	✓	✓
Simple Send	Send MPI message			✓
Simple Receive	Receive MPI message			✓
FFT	Double-precision FFT	✓	✓	✓
Cholesky	Single-precision Cholesky factorization	✓	✓	✓
Ping-pong	MPI ping-pong			✓
Ring pass	MPI ring pass			✓

Evaluation Metric: We first record the ratio of generated samples that build correctly to those that do not. This indicates the model's ability to generate syntactically correct code. For those that compile we compute the pass@k metric that denotes the probability that at least one of k samples out of N_p code samples is correct. We do N_p trials with each prompt p to generate N_p code samples, compile/run the samples, and record the number that are functionally correct (c_p). To estimate the probability that at least one of k samples chosen from N_p samples is correct for a particular prompt, p, we

(a) Prompt

```
/*
 multiply scalar float a by vector x and add to y
 vectors x and y are length N
 use OpenMP to compute in parallel
*/
void saxpy(float *x, float *y, float a, int N) {
```

(b) Output

```
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] += a * x[i];
    }
}
```

Fig. 3. An example prompt asking the model to generate a parallel version of saxpy. The comment and function header make up the prompt. The function body on the bottom shows a potential model output.

can use the number of generated samples that are functionally correct, c_p, out of the N_p total samples generated to calculate pass@k for a given k as,

$$\text{pass@}k = 1 - \binom{N_p - c_p}{k} \Big/ \binom{N_p}{k} \tag{1}$$

For each model, we report the average_pass@k metric as the average pass@k over all P prompts as shown below:

$$\text{average\_pass@}k = \frac{1}{P}\sum_{i=1}^{P}\left[1 - \frac{\binom{N_i - c_i}{k}}{\binom{N_i}{k}}\right] \tag{2}$$

This metric provides insight into the probability of a model generating functionally correct code. In our experiments, we calculate the pass@k score for several temperatures, namely 0.1, 0.2, 0.4, 0.6, and 0.8, and select the best one. This is in line with experiments in related literature [14]. For each temperature and prompt, we generate $N_p = 100$ samples. The code is generated with nucleus sampling using 0.93 as the cutoff value in the CDF (see Section II).

To compile the generated code samples, we use `g++` with the "`-O2 -std=c++17 -fopenmp`" flags. For tests that need MPI we use the OpenMPI `mpicxx` compiler. If the build is successful, then a corresponding driver binary is called that will call and test the generated function for correctness. These are run on a AMD EPYC 7763 CPUs with 64 physical cores at 2.45 GHz each. For tests that require OpenMP or MPI we only denote them as correct if they used the corresponding parallel framework to compute their result.

B. Predicting OpenMP Pragmas

A common HPC coding task is decorating `for` loops with OpenMP pragmas. Every pragma starts with `#pragma omp parallel for` and is followed by a list of optional clauses that modify the behavior of the parallel `for`. We test the model's ability to write OpenMP pragmas for arbitrary `for` loops.

Further Fine-tuning: We cannot directly use the existing models to generate pragmas *before* a `for` loop, since they are all left-to-right and can only append tokens to sequences. Thus, we need to further fine-tune the models on a smaller dataset that puts the `for` loop before the pragma. To accomplish this, we first create a dataset of every `for` loop with an OpenMP pragma from our HPC code dataset. 500 tokens of context from before the `for` loop are also included. This results in a dataset with 13,900 samples.

Since our model is left-to-right, we format each sample by moving the pragma to directly after the loop and a unique separating token `<begin-omp>`. This allows us to use the model by providing a `for` loop plus some context and the model will generate an OpenMP pragma for the `for` loop.

Each model is fine-tuned on this smaller dataset for three epochs (passes over the entire dataset). To prevent overfitting we use a starting learning rate of 3×10^{-5}. During training 10% of the dataset is set aside for validation.

Evaluation Metric: To measure the success of this test, we use the accuracy of generating correct pragmas. This is calculated as shown in Equation 3.

$$\text{accuracy} = \frac{\text{\# correct pragmas}}{\text{total pragmas tested}} \tag{3}$$

For this problem, we define a *correct* pragma in two ways: syntactic and functional. To measure syntactic correctness we compare the generated pragma with the actual pragma for textual equivalence. Since it is impossible to automate the running and evaluation of arbitrary `for` loops from our dataset we measure functional correctness by comparing the generated pragmas with the actual ones while ignoring differences that do not contribute to functionality. For instance we ignore reordering of variables and clauses where these do not matter. Additionally, clauses such as *schedule* are ignored. This correctness check is done using a custom Python script that parses the pragmas and compares them. We record accuracy from both of these correctness metrics for each model.

C. Relative Performance Prediction

In addition to text generation, we can also use the LLMs for classification. Here we use them to predict performance slowdowns between two pairs of code.

Further Fine-tuning: In order to use the models for relative performance classification we need to first fine-tune them on new data for this output task. Using the Git commit data from Section IV-C we give the model text for a region of code before and after a Git commit. The codes are concatenated with a unique token separating them, namely `<COMMIT>`. We repeat a similar process for the code contest dataset, but instead separate pairs by the token `<PAIR>`. With this data the model is fine-tuned to predict whether the second code will be slower (*positive*) or the same/faster (*negative*).

For each dataset we fine-tune the model on 90% of the data with the other 10% set aside for evaluation. The model takes the concatenated sequences of the two versions of the code implementation and is fine-tuned for the binary classification problem of predicting relative performance. The

training objective is classification accuracy, which we also use to measure success for this task.

Evaluation Metric: To evaluate the performance on this task we measure the model's classification accuracy. This is calculated as shown in Equation 4.

$$\text{accuracy} = \frac{\text{\# correct performance predictions}}{\text{total performance predictions}} \quad (4)$$

For this metric higher is better and a classification accuracy of 100% signifies a perfect score.

VII. Results

We now present the fine-tuning and evaluation results using the downstream tasks discussed in Section VI.

A. Fine-tuning on HPC Source Code Data

We first show the results of fine-tuning the three models selected in Table II. Table IV shows the validation perplexity at the end of fine-tuning. Here perplexity is calculated as the exponential of the loss as described in Section II. Each model converges to a low perplexity score over the separate testing set (between 2 and 4). GPT-Neo and PolyCoder achieve comparable perplexity scores (within 0.01) while GPT2 achieves a higher perplexity. All three have different pre-training datasets and the former two are of a larger size than GPT2 (see Table II). From this we can conclude that for this problem the pre-training dataset had less of an impact on validation perplexity than the model size. The lower perplexity of the larger models means that they model the language better.

TABLE IV
Final validation perplexities for each model after fine-tuning on the HPC source code dataset.

Model	GPT-2	GPT-Neo	PolyCoder
Final Validation Perplexity	4.47	2.23	2.24

For the rest of the results presented in this section we will use PolyCoder+HPC, GPT-Neo+HPC, and GPT2+HPC to refer to the respective models fine-tuned on the HPC dataset.

After fine-tuning each of the models and evaluating them on the downstream tasks we noticed that the perplexity would keep improving with more fine-tuning, but the downstream evaluation performance would start to decrease. This is likely because LLMs are subject to *catastrophic forgetting* during fine-tuning. *Catastrophic forgetting* is the phenomenon where previously learned information is lost or forgotten as the model continues training and updating its weights. It is typically prevented by minimizing the amount of fine-tuning and using a sufficiently low learning rate.

To explore this phenomenon we ran the code generation tasks every 1000 samples during fine-tuning of the PolyCoder model. Figure 4 presents the results from our evaluation tests during fine-tuning on the PolyCoder model. After seeing about 45,000 samples during fine-tuning the model starts to decrease in evaluation performance. This is in contrast to the perplexity which keeps improving past 45,000 samples. Based on this result we stop fine-tuning at 45,000 samples and use these weights for the rest of the evaluations. Additionally, due to the computation time needed to run this test we use the 45,000 samples stopping point for fine-tuning all the models.

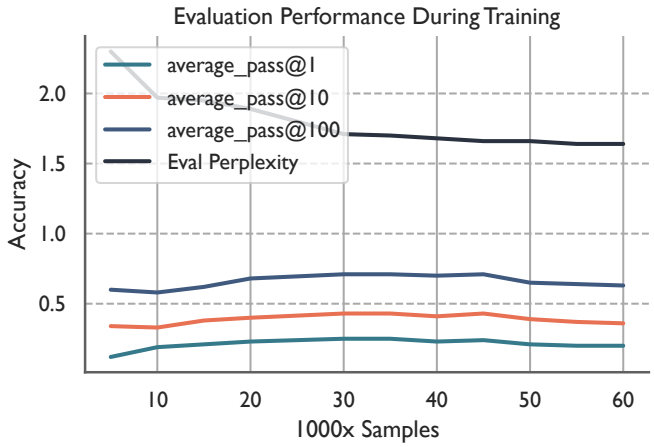

Fig. 4. Downstream evaluation performance across training iterations for PolyCoder+HPC. The model starts to perform worse around 45,000 samples even though the perplexity keeps improving.

B. Code Completion

Having fine-tuned the three models, we now start using them for the different downstream tasks described in Section VI. The first downstream task is code generation, described in Section VI-A. Figure 5 shows the average_pass@k rates for the code generation tests. The average_pass@k values are computed according to Equation 2. We use PolyCoder as a baseline for comparison since it is a state-of-the-art LLM for code generation. PolyCoder+HPC scores the best for average pass@1, pass@10, and pass@100. For each value of k the models score in the order of PolyCoder+HPC, PolyCoder, GPT-Neo+HPC, and GPT2+HPC. PolyCoder+HPC gains the slight edge over the original PolyCoder by successfully generating code for the HPC-specific tasks (see Figure 6).

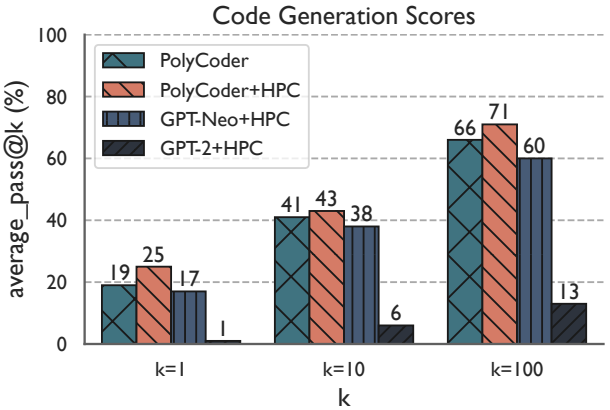

Fig. 5. Comparison of models on code generation. The clusters represent the average pass@k scores for $k = 1, 10$ and 100. Higher percentage is better.

In Figure 5 we see that GPT2+HPC scores significantly lower than the other models. This is likely due to the smaller model size and the fact that there is no source code in its pre-training dataset. In this instance fine-tuning is not enough to enable GPT-2 to generate correct C++ HPC code.

Altogether, the scores are indicative that PolyCoder+HPC and GPT-Neo+HPC has learned how to generate valid C++ code. For instance, if the best model, PolyCoder+HPC, is permitted to generate 100 samples, then 71% of them are correct on average across all the tests. Similarly for 1 sample generated this is 25%. These numbers roughly align with results from [14] on the HumanEval Python tests. However, the results are not directly comparable since they are a different set of tests in a different programming language.

To demonstrate the generative capabilities of the specialized models we reduce the code generation tasks to those that are specific to HPC. This includes code that uses OpenMP and/or MPI parallelism. Figure 6 shows the performance when restricted to these tests. We see that PolyCoder is unable to generate OpenMP and MPI code as it scores significantly lower than the rest. GPT2+HPC still performs fairly low, however, its score has actually improved slightly over Figure 5. This is due to the fact that it has only seen HPC-specific code during training and that is what is being tested here.

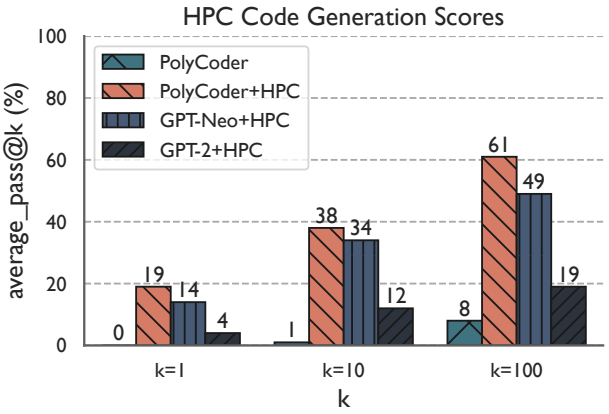

Fig. 6. Comparison of models on code generation for HPC-specific functions. The clusters represent the average pass@k scores for $k = 1, 10$ and 100. Higher percentage is better.

Another point of interest besides functional correctness is syntactic correctness. This can be measured by the total number of generated samples that compile successfully. This is how often the model generates valid code, whether it is functionally correct or not. This data is presented in Figure 7. PolyCoder and PolyCoder+HPC both perform the best compared to the other models with 84% and 86% of samples compiling correctly, respectively. GPT-Neo+HPC performs slightly worse at 74% and GPT2-HPC has only 30% of samples compile. The worse performance of the latter two can likely be attribute to their pre-training datasets having less code. We also observe that for all models there is a visual correlation between build and correctness rates, which is expected as a model needs to compile in order to be functionally correct.

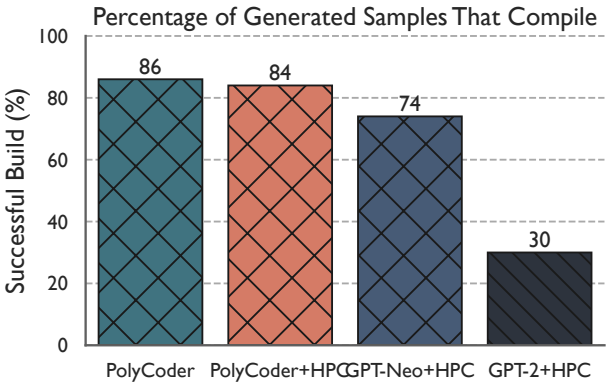

Fig. 7. Comparison of the models' build rate. Both PolyCoder and PolyCoder+HPC have the best percentage of total samples that successfully compile. Higher percentage is better.

The code in Figure 8 shows example output from PolyCoder and PolyCoder+HPC on generating OpenMP code to compute a sum in parallel. We see that PolyCoder is able to produce correct sequential code, however, it fails to add any OpenMP pragmas. The PolyCoder+HPC model is able to correctly tag the for loop with an OpenMP pragma.

(a) Prompt

```
/*
 Compute the sum of the array X and return the sum.
 X has N elements.
 Use OpenMP to compute the sum in parallel.
*/
float sum(float *X, int N) {
```

(b) PolyCoder output

```
    float sum = 0.0f;
    for (int i = 0; i < N; i++)
        sum += X[i];
    return sum;
}
```

(c) PolyCoder+HPC output

```
    float sum = 0.0f;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < N; i++)
        sum += X[i];
    return sum;
}
```

Fig. 8. Example OpenMP output from (b) PolyCoder and (c) PolyCoder+HPC. The comment and function description (top) make up the prompt that is given to the model, while the bottom two blocks are the generated text. We see that PolyCoder is unable to generate OpenMP pragmas for the reduction in this example.

A similar example with distributed memory is shown in Figure 9. The PolyCoder+HPC is able to generate correct MPI code to compute an average in parallel across ranks. PolyCoder often generated long and incorrect code that demonstrated little understanding of how to write MPI routines. The example presented is indicative of the rest of the MPI samples generated

by PolyCoder. Notably, some instances did contain MPI calls, so PolyCoder must have some MPI code in its pre-training dataset. However, it lacked the ability to properly use MPI.

(a) Prompt

```
/*
    Use MPI to compute the average of X across ranks
    and return the result on all ranks.
    Use the default communicator.
*/
double mpiAverage(double X) {
```

(b) PolyCoder output

```
    int rank;
    int nprocs;
    double average;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    if (rank == 0) {
        MPI_Recv(&status, 1, MPI_INT, 0, 0,
            MPI_COMM_WORLD,
                &request);
        MPI_Send(&X, 1, MPI_INT, 0, 0,
            MPI_COMM_WORLD);
        MPI_Send(&average, 1, MPI_DOUBLE, 0, 0,
                MPI_COMM_WORLD);
    }
    return average;
}
```

(c) PolyCoder+HPC output

```
    int size;
    double sum;

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Allreduce(&X, &sum, 1, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD);

    return sum / size;
}
```

Fig. 9. Example MPI output from (b) PolyCoder and (c) PolyCoder+HPC. The highlighted region is code generated by the model (reformatted to fit the column). PolyCoder results varied significantly, however, the above example demonstrates the general lack of understanding it had for MPI.

Figure 10 shows the speedups for the code generated by PolyCoder+HPC over sequential baselines. These are hand-written efficient, sequential implementations for each test. We see that PolyCoder+HPC is able to generate code that is faster than the sequential baseline. This demonstrates that it is not generating very poor performing parallel code and is likely using the parallelism correctly.

Since PolyCoder+HPC scores the highest in training and these code generation tests we select it for further comparisons in the rest of the paper. PolyCoder+HPC is the fine-tuned model we present as HPC-Coder. We continue to use PolyCoder as a baseline.

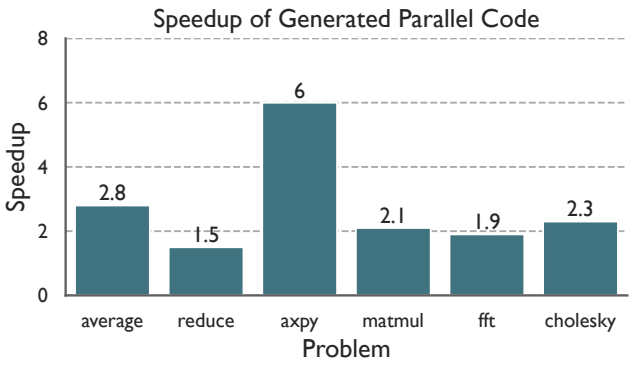

Fig. 10. Comparison of the speedups for the code generation tests over sequential baselines. They are all above 1 demonstrating that the model is not generating very poor performing parallel code.

C. Predicting OpenMP Pragmas

Next, we examine the result from the OpenMP prediction tests described in Section VI-B. Figure 11 shows the results from the OpenMP experiments detailed in Section VI-B. We see that both models are able to generate functionally correct OpenMP pragmas with high accuracy (right plot). PolyCoder+HPC is able to do this with 97% accuracy and PolyCoder 94%. The LLMs are exemplary at understanding the dependencies of the `for` loop and what clauses are required to correctly parallelize them. We see that the model that has seen large amounts of OpenMP code performs better.

We can also look at how well the models reproduce the pragmas exactly. This means all the clauses and variables within those clauses are in the same order in the dataset and in the output from the model. These results are shown in the left plot in Figure 11. While less meaningful than functional correctness, it is interesting that the model is able to exactly reproduce pragmas it has not seen before with relatively high accuracy (67% and 61%). This is likely due to certain trends in the construction and ordering of OpenMP clauses that the LLMs are learning as they train.

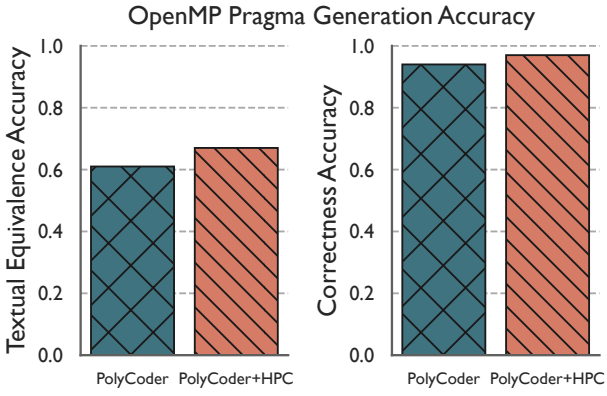

Fig. 11. Comparison of models on predicting OpenMP pragmas. The left plot presents accuracy in predicting OpenMP pragmas exactly as they appear in the dataset. The right plot shows the accuracy in predicting functionally correct OpenMP pragmas. Higher accuracy is better.

D. Relative Performance Prediction

Finally, we look at the results from the relative performance prediction tests described in Section VI-C. Figure 12 shows the results from the relative performance prediction tests (see Section VI-C). Both models achieve high classification accuracy with PolyCoder+HPC being slightly better for the two proxy applications at 88% and PolyCoder at 86%. This means that for 88% of the code changes in the two repositories version control history PolyCoder+HPC is able to correctly identify if there will be a performance slowdown. Likewise for the programming competition dataset we see that PolyCoder+HPC outperforms the PolyCoder baseline with an accuracy of 92% vs 86%. This is a higher accuracy improvement than the proxy applications by 4 percentage points. This is likely due to the fact that the programming competition dataset is larger and PolyCoder+HPC has been trained on more C/C++ code.

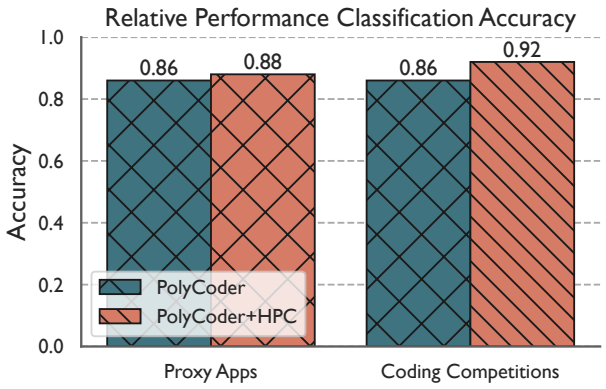

Fig. 12. Comparison of models on predicting relative performance of code changes. Both models achieve similarly high accuracy. The PolyCoder+HPC model performs slightly better on both datasets. Higher accuracy is better.

The success of this test demonstrates that the models are able to correlate their prior language understanding with performance related properties of code. This means we can leverage LLMs and fine-tuning to model code performance without the need to collect large amounts data.

VIII. Related Work

In this section we detail related work that uses LLMs to study source code and work that uses machine learning to model the performance of source code.

A. LLMs for Code Generation

With the explosion in research in transformer models and LLMs there have been a large number of papers applying these techniques to source code. Most of these methods have extended GPT-2 [16], GPT-3 [23], or BERT [32], [33] models and trained them on code. A notable instance is Codex [2], which is a modification of GPT-3 that is targeted for source code generation. Following Codex's introduction there have been several other works that have introduced state-of-the-art large language models [3], [4], [34]. While some of these are open source, the best, such as GPT-4 [24], keep their architecture, weights, and training data closed source and only inference is available via a paid API.

A large amount of this recent research has focused on code generation. These usually take a mix of code and natural language and learn how to meaningfully finish the code. While seminal works have continued to improve code generation with better and bigger models [2], [23], [33], other works have explored how to better utilize these tools in software engineering workflows [35]–[37]. Some flip code generation around and learn to generate natural language code summaries from code snippets [7]–[10].

These models can even be trained for tasks such bug and malware detection [11], [12]. LLMs can also be used to suggest fixes in these cases rather than just identify problematic code. Many other previously difficult to automate software development tasks have since been tackled by applying LLMs [6]. More recently some of these tasks have included HPC development tasks such as race detection [38] and OpenACC compiler validation [39].

B. Machine Learning Applied to Source Code Performance

However, one important problem in software development that has not received much research with LLMs is that of performance. Many of the reasons listed in Section I have prevented meaningful studies from being accomplished. Previously approaches used code2vec [40], ir2vec [41], or a similar method to first map source code to an embedded space that could then be learned on. These were successfully used for some performance related analytical modeling such as OpenCL kernel device placement [41], but never leveraged LLMs for a full performance study.

Garg et al. [42] recently introduced DeepDevPERF, which is a BART-based [43] LLM designed to suggest performance improvements to arbitrary C# code. They overcome the issue of data collection by using code changes from Git commits that have performance related keywords in their commit message, albeit, this dataset is still noisy. This work is different than that presented in this paper as it suggests code transformations rather than learn relative performance. The latter being useful in cases where two versions of a code already exist, such as with Git commits. Additionally, our model is trained on real performance data and can be used for HPC and parallel code generation tasks.

IX. Conclusion and Future Work

In this paper, we have demonstrated the fine-tuning of an LLM using HPC code, and its ability to outperform other LLMs in HPC related tasks such as HPC code generation and performance modeling. We have accomplished this by fine-tuning a model, and showing that it can generate functionally correct HPC code at up to a 53% higher pass@k rate and can accurately label `for` loops with OpenMP pragmas with 97% success. We have further demonstrated how this fine-tuned model can be utilized to study performance properties of source code with little data. These results demonstrate the need for and usefulness of HPC-specific language models. The

best model in our experiments, PolyCoder+HPC, we present as *HPC-Coder*.

In the future, we plan to explore further analyses that can be accomplished using our language model. We also plan on exploring how to tune the model to generate not just correct but performant code. Additionally, we plan to investigate how to engineer these innovations into practical tools that can be easily used by computational scientists and HPC developers to enable them to produce better code more efficiently.

ACKNOWLEDGMENT

This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-844549).

REFERENCES

[1] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2023.

[2] M. Chen and et al, "Evaluating large language models trained on code," 2021.

[3] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[4] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2023.

[5] J. Senanayake, H. Kalutarage, and M. O. Al-Kadri, "Android mobile malware detection using machine learning: A systematic review," *Electronics*, vol. 10, no. 13, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/13/1606

[6] "Ml4code," https://ml4code.github.io/, accessed: 2022.

[7] J. Gu, P. Salza, and H. C. Gall, "Assemble foundation models for automatic code summarization," *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 935–946, 2022.

[8] T. Ahmed and P. Devanbu, "Learning code summarization from a small and local dataset," *ArXiv*, vol. abs/2206.00804, 2022.

[9] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, "Semantic similarity metrics for evaluating source code summarization," *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, pp. 36–47, 2022.

[10] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *ArXiv*, vol. abs/2005.00653, 2020.

[11] C. Richter and H. Wehrheim, "Can we learn from developer mistakes? learning to localize and repair real bugs from real bug fixes," *ArXiv*, vol. abs/2207.00301, 2022.

[12] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. B. Clement, and N. Sundaresan, "Learning to reduce false positives in analytic bug detectors," *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1307–1316, 2022.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[14] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A Systematic Evaluation of Large Language Models of Code," Feb. 2022, https://arxiv.org/abs/2202.13169. [Online]. Available: https://doi.org/10.5281/zenodo.6363556

[15] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 143–153. [Online]. Available: https://doi.org/10.1145/3359591.3359735

[16] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[17] A. Kunen, T. Bailey, and P. Brown, "KRIPKE-a massively parallel transport mini-app," *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2015.

[18] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-order curvilinear finite element methods for lagrangian hydrodynamics," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. B606–B641, 2012. [Online]. Available: https://doi.org/10.1137/120864672

[19] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," 2022. [Online]. Available: https://arxiv.org/abs/2203.07814

[20] A. Gokaslan and V. Cohen, "Openwebtext corpus," http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[21] S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," Mar. 2021, If you use this software, please cite it using these metadata. [Online]. Available: https://doi.org/10.5281/zenodo.5297715

[22] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," *CoRR*, vol. abs/2101.00027, 2021. [Online]. Available: https://arxiv.org/abs/2101.00027

[23] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[24] OpenAI, "Gpt-4 technical report," 2023.

[25] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-Art Natural Language Processing." Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[26] Microsoft, "Deepspeed: Extreme-scale model training for everyone," https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/.

[27] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training," *CoRR*, vol. abs/2101.06840, 2021. [Online]. Available: https://arxiv.org/abs/2101.06840

[28] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3320060

[29] D. Nichols, S. Singh, S.-H. Lin, and A. Bhatele, "A survey and empirical evaluation of parallel deep learning frameworks," 2022.

[30] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," *CoRR*, vol. abs/1711.05101, 2017. [Online]. Available: http://arxiv.org/abs/1711.05101

[31] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri,

G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://www.aclweb.org/anthology/N19-1423

[33] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[34] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv preprint arXiv:2312.02120*, 2023.

[35] J.-B. Döderlein, M. Acher, D. E. Khelladi, and B. Combemale, "Piloting copilot and codex: Hot temperature, cold prompts, or black magic?" *ArXiv*, vol. abs/2210.14699, 2022.

[36] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *ArXiv*, vol. abs/2206.15000, 2022.

[37] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. G. Zorn, "What is it like to program with artificial intelligence?" *ArXiv*, vol. abs/2208.06213, 2022.

[38] L. Chen, X. Ding, M. Emani, T. Vanderbruggen, P. hung Lin, and C. Liao, "Data race detection using large language models," 2023.

[39] C. Munley, A. Jarmusch, and S. Chandrasekaran, "Llm4vv: Developing llm-driven testsuite for compiler validation," 2023.

[40] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," 2018. [Online]. Available: https://arxiv.org/abs/1803.09473

[41] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant, "Ir2v¡span class="smallcaps smallercapital"¿ec¡/span¿: Llvm ir based scalable program embeddings," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, dec 2020. [Online]. Available: https://doi.org/10.1145/3418463

[42] S. Garg, R. Z. Moghaddam, C. B. Clement, N. Sundaresan, and C. Wu, "Deepdev-perf: a deep learning-based approach for improving software performance," *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[43] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019. [Online]. Available: https://arxiv.org/abs/1910.13461