Can Large Language Models Write Parallel Code?

Daniel Nichols
dnicho@umd.edu
Department of Computer Science,
University of Maryland
College Park, Maryland, USA

Joshua H. Davis
jhdavis@umd.edu
Department of Computer Science,
University of Maryland
College Park, Maryland, USA

Zhaojun Xie
zxie12@umd.edu
Department of Computer Science,
University of Maryland
College Park, Maryland, USA

Arjun Rajaram
arajara1@umd.edu
Department of Computer Science,
University of Maryland
College Park, Maryland, USA

Abhinav Bhatele
bhatele@cs.umd.edu
Department of Computer Science,
University of Maryland
College Park, Maryland, USA

ABSTRACT

Large language models are increasingly becoming a popular tool for software development. Their ability to model and generate source code has been demonstrated in a variety of contexts, including code completion, summarization, translation, and lookup. However, they often struggle to generate code for complex programs. In this paper, we study the capabilities of state-of-the-art language models to generate parallel code. In order to evaluate language models, we create a benchmark, ParEval, consisting of prompts that represent 420 different coding tasks related to scientific and parallel computing. We use ParEval to evaluate the effectiveness of several state-of-the-art open- and closed-source language models on these tasks. We introduce novel metrics for evaluating the performance of generated code, and use them to explore how well each large language model performs for 12 different computational problem types and six different parallel programming models.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; **Neural networks**; **Artificial intelligence**.

KEYWORDS

Large language models, Parallel code generation, Performance evaluation, Benchmarking, HPC

ACM Reference Format:
Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024. Can Large Language Models Write Parallel Code?. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3625549.3658689

HPDC '24, June 3–7, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0413-0/24/06
https://doi.org/10.1145/3625549.3658689

1 INTRODUCTION

Large language model (LLM) based coding tools are becoming popular in software development workflows. Prior work has demonstrated their effectiveness at performing a variety of tasks, including code completion, summarization, translation, and lookup [4, 5, 18, 20, 21, 26, 39]. Popular models such as StarCoder [29], span a wide range of programming languages and domains, and can be used to complete or generate code during the development process. This makes them a promising tool for improving developer productivity and the overall quality of software. However, despite the rapid advancement and scaling of LLMs in recent years, they still struggle with more complicated tasks such as reasoning and planning. One particularly complex task that LLMs struggle with is generating *parallel* code. This task involves reasoning about data distributions, parallel algorithms, and parallel programming models.

Parallel code is essential to modern software development due to the ubiquity of multi-core processors, GPGPUs, and distributed systems. However, writing parallel code is difficult and error-prone. Parallel algorithms are generally more complicated than their sequential counterparts, and parallel bugs such as race conditions and deadlocks are notoriously non-trivial to debug. Further, it can be challenging to reason about the performance of parallel code and identify "performance bugs" [25]. LLMs can potentially help developers overcome these challenges but, this requires an understanding of the current capabilities of LLMs, and in turn, a well-designed and reproducible methodology to assess these capabilities.

There are several existing benchmarks for evaluating the capabilities of LLMs to generate correct code. However, none of them test generation of *parallel* code. Most existing benchmarks focus on short, array or string manipulation tasks, and are predominantly in Python (or translated to other languages from Python [9]). Only more recent benchmarks such as DS-1000 [28], test the usage of APIs, which are critical to using parallel programming models. Further, these benchmarks do not evaluate the performance of the generated code, instead testing only correctness. While correctness is a crucial metric, performance is also vital for developers writing parallel code. Thus, it is imperative to design new benchmarks and metrics to evaluate the usefulness of LLMs for parallel code generation tasks.

Developing a set of benchmarks that fully covers the space of desired capabilities is non-trivial. Identifying the best LLM for parallel

code generation requires testing on problems that cover shared- and distributed-memory programming models, different computational problem types, and different parallel algorithms. This can become a large quantity of benchmarks that must be manually designed. Further, these benchmarks are challenging to test. Traditional Python code generation benchmarks are tested by running *eval* on the generated code for a small number of small unit tests. On the other hand, in the case of parallel code — we must compile C/C++ code, link against one or more parallel libraries, and run the code in the proper parallel environment. Additionally, if we want to test the performance of the generated code, then we must choose reasonable input sizes for each benchmark.

In order to evaluate the current capabilities and limitations of LLMs in generating parallel code, we propose the Parallel Code Generation Evaluation (ParEval) benchmark: a set of benchmarks (prompts) for evaluating how well LLMs generate parallel code. These benchmarks cover twelve different computational problem types, and seven different execution models: serial, OpenMP, Kokkos, MPI, MPI+OpenMP, CUDA, and HIP. We evaluate several state-of-the-art open- and closed-source LLMs using these benchmarks, and report metrics that represent the *correctness* and *performance* of the generated code. We introduce novel code generation evaluation metrics that assess performance and parallel scaling. We further analyze how each model performs with respect to the various programming models and computational problem types. We discuss the areas where current state-of-the-art LLMs are already performing well and the areas where they can be improved.

In this paper, we make the following important contributions:

- We design the ParEval benchmark for evaluating the ability of LLMs to generate and translate parallel code. ParEval is available online at: github.com/parallelcodefoundry/ParEval.
- We introduce two novel metrics, speedup$_n$@k and efficiency$_n$@k, for evaluating the performance and scaling of LLM generated code.
- We evaluate the effectiveness of several state-of-the-art open- and closed-source LLMs using the ParEval benchmark.
- We identify several areas where current state-of-the-art LLMs can improve their capabilities on parallel code generation.

In addition to these contributions, we explore the following research questions (answers based on our observations):

RQ1 *How well do state-of-the-art LLMs generate parallel code, and which models are the best?* We show that all tested LLMs, both open- and closed-source, struggle to generate parallel code. Of the models tested, GPT-3.5 performs the best with a pass@1 of 76.0 for serial code generation and a pass@1 of 39.6 for parallel code generation.

RQ2 *Which parallel execution models and problem types are most challenging for LLMs?* We observe that LLMs struggle most with MPI code generation, and perform best for OpenMP and Kokkos code generation. Additionally, we show that LLMs find it challenging to generate parallel code for sparse, unstructured problems.

RQ3 *How performant and scalable is the parallel code generated by LLMs?* We observe that the parallel code generated by LLMs can have poor parallel speedup and efficiency. Additionally, we show that the LLMs that most often generate correct parallel code do not necessarily generate the most performant parallel code.

RQ4 *How well can LLMs translate between execution models? How performant and scalable is the translated code?* We show that providing LLMs with correct implementations in one execution model can improve their ability to generate correct code in another execution model. This is particularly true for smaller open-source models.

2 BACKGROUND

In this section, we provide background information on large language models and how they are used for text generation. We further discuss how large language models can be used for code generation.

2.1 Large Language Models

Natural Language Processing (NLP) has largely been dominated by transformer-based models since their introduction in 2017 by Vaswani et al. [46]. Transformer networks are designed to model sequential data, such as text, relying on *self-attention* mechanisms to model the relationships between values in a sequence. Self-attention enables modeling of long-range dependencies in the data without vanishing gradient and scaling issues and allows for sequence elements to processed in parallel. Transformers learn *attention scores*, which are computed between pairs of tokens in the input. *Multi-head attention* allows for learning multiple attention representations. These large transformer models are generally trained to model the distribution of a text corpus such as the English language by predicting the next token in a sequence given previous tokens. Transformer-based models have emerged as the most effective means of modeling text data, and have been shown to be effective at a wide range of NLP tasks.

2.2 Large Language Models for Code

An LLM trained on a large corpus of code can be used to generate code by giving it a code input prompt and asking it to predict the next token. Generally, code LLMs are trained on a large corpus of code, such as The Stack [27], that covers a wide range of programming languages and application types. Sometimes the pre-training corpus includes natural language as well, such as The Pile [16, 48]. In some instances, such as CodeLlama [40], the code LLM is a natural language model that has been further fine-tuned on a corpus of code. When generating code with one of these models it is often not enough to simply select the most probable next token to construct a sequence. This often leads to repetitive, low-quality outputs [22], so we also need a strategy for token selection. We utilize *nucleus sampling* and *model temperature* in this study.

Nucleus Sampling. Nucleus sampling [22], also called *top-p sampling*, samples the next token from the token probability distribution up to some cut-off p in the cumulative distribution function. Compared to sampling from a fixed number of top tokens in the distribution (called *top-k sampling*), this ensures the selection of a more representative sample of tokens from the distribution. Nucleus sampling is often used in code generation tasks with a value of $p = 0.95$ and is sometimes combined with top-k sampling.

Model Temperature. Generation *temperature* is a scaling value applied to the raw model outputs, or *logits*, before they are converted to a probability distribution. The value is applied by first dividing the logits vector by the scalar temperature before computing the softmax of the logits. Higher temperatures make the probability distribution more peaked, upweighting the most probably tokens, while lower temperatures make the distribution more uniform. Intuitively, lower temperatures yield more conservative generations that the model is more *confident* in. Conversely, higher temperatures will lead to more varied and *creative* generations. For code generation tasks, a low temperature value of 0.2 is often used.

3 RELATED WORK

Below, we describe related work in benchmarking LLMs for code-related tasks and applying LLMs to parallel and HPC code.

3.1 Benchmarking LLMs for Code-related Tasks

Since the introduction of the Codex model and HumanEval benchmark [13], many works have proposed new LLMs for code and evaluated them on a variety of tasks. The number of code-specific models has grown rapidly as open-source models and data sets become more available and low-rank training techniques, such as LoRA [23], make training large models more feasible. These models are usually evaluated on code generation tasks such as HumanEval [13], MBPP [7], and DS-1000 [28].

The first of these, HumanEval [13], is a set of 164 code generation tasks that are designed to evaluate the ability of LLMs to write short Python functions that solve a variety of problems, given a docstring and function signature. Similar to HumanEval is the Mostly Basic Python Problems (MBPP) [7] benchmark which is a set of 1000 simple Python problems. MBPP is often evaluated with few-shot prompts, where example correct solutions to other problems are included in the prompts. A common extension of these benchmarks is MultiPL-E [9] which extends the set of HumanEval and MBPP tests to 18 programming languages.

The DS-1000 benchmark [28] tests the ability of the model to generate more complex, data science-related code, for 1000 tasks making use of common data science libraries. Other similar benchmarks that evaluate coding LLMs on more complex tasks are GSM8K [14] and GSM-HARD [17], which use PAL [17] to evaluate the ability of LLMs to generate Python code snippets to assist in chains of reasoning. The CoderEval benchmarks [49] are a set of 230 Java and 230 Python code generation tasks that require the model to write context-dependent functions, rather than standalone functions as in HumanEval and MBPP.

Additionally, there have been several domain specific benchmarks that evaluate more narrow uses of LLM code generation [15, 30, 42]. All of these benchmarks make use of tasks manually created by experts to test more specific use cases of LLMs.

3.2 Applying LLMs to Parallel and HPC Code

Recently there has been a growing interest in applying LLMs to parallel and High Performance Computing (HPC) code. Several works have looked at creating smaller specialized HPC models [24**?**] or applying existing LLMs to HPC tasks [10, 11, 31]. Nichols et al. [**?**] introduce HPCCoder, a model fine-tuned on HPC code, and evaluate its ability to generate HPC code, label OpenMP pragmas, and predict performance. Kadosh et al. [24] introduce TOKOMPILER, an HPC specific tokenizer for LLMs, and use it to train COMPCODER, a model trained on C, C++, and Fortran code.

Other works have looked at applying existing LLMs to HPC tasks. Munley et al. [31] evaluate the ability of LLMs to generate compiler verification tests for parallel OpenACC code. Chen et al. [10] use LLMs to identify data races in parallel code and propose the DRB-ML data set, which is integrated into the LM4HPC framework [11]. Godoy et al. [19] and Valero-Lara et al. [45] both evaluate the capabilities of LLMs on generating HPC kernels, but use a limited set of problems and LLMs and do not prompt or evaluate the LLMs using standard practices. None of these works comprehensively evaluate and compare the ability of LLMs to generate parallel code across a large number of problems, execution models, and LLMs using state-of-the-art evaluation techniques, which is the focus of this work.

4 PAREVAL: PROMPTS FOR PARALLEL CODE GENERATION

In order to evaluate the ability of LLMs to generate parallel code, we propose the Parallel Code Generation Evaluation (ParEval) benchmark. Below, we discuss the design of ParEval, and its various components that lead to the creation of concrete prompts for LLMs.

To disambiguate the use of the terms *prompt*, *task*, *problem*, *problem type*, and *benchmark* we define them as follows.

- *Task/Prompt*: An individual text prompt that is given to the LLM to generate code. The output can be compiled, executed, and scored as either correct or incorrect code.
- *Problem*: A set of tasks or prompts that test the ability of the LLM to generate code for the same computational work, but each task or prompt may use a different execution model.
- *Problem Type*: A set of problems that test computational problems with similar work or from similar domains (for example, *sorting* problems).
- *Benchmark*: A set of prompts that are all tested together to evaluate the performance of the LLM. We name the collection of all the prompts we have designed as the ParEval benchmark.

Benchmark Requirements. The goal of ParEval is to evaluate the ability of LLMs to generate parallel code. To do this, the prompts should be such that:

(1) The prompts should cover a wide variety of computational problem types, and parallel programming models.
(2) The prompts should be simple enough that they can be generated as a standalone function, but complex enough that they are not too trivial to solve.
(3) The prompts should not exist within any of the LLMs' training datasets, to prevent the LLMs from simply copying solutions from their training data.
(4) The prompts and corresponding outputs should be able to be evaluated automatically, since there will be many different tasks and LLM outputs.

In order to fulfill the requirements above, we propose ParEval, a set of 420 prompts that cover twelve different computational

problem types and seven different execution models. Each problem type has five different problems, and each problem has a prompt for each of the seven execution models, resulting in 420 total prompts. Each prompt in ParEval is a standalone function that requires the LLM to generate code that solves the problem either sequentially or in parallel.

Problem Types. The problem types are listed and described in Table 1. These were hand-selected by us, and represent a wide variety of common computational problems that are often parallelized. Each requires different strategies or APIs to solve in parallel. For instance, the problems in the *Sort* problem type require the LLM to generate code that sorts an array of values.

Problems. The five problems within each problem type are designed to test the core functionality of the problem type. To prevent prompting the model for a solution that is already in its training dataset, the five problems are small variations of the usual problem type. For example, one of the scan problems is to compute the *reverse* prefix sum of an array, rather than directly computing the prefix sum. These variations still test the model's understanding of the core computational problem, but mitigate the likelihood of it simply copying code from its training dataset. Listing 1 shows another example of these problem variations. Another benefit of having five problems per problem type is that it provides more data points for evaluating the LLM's performance on that problem type, but not so many that it becomes infeasible to implement and maintain.

Prompts. Each problem has a prompt for each of the seven execution models that the LLM is required to generate code for. The seven execution models we test are: serial, OpenMP [36], MPI [41], MPI+OpenMP, Kokkos [44], CUDA [32], and HIP [2]. All the prompts are in C++, CUDA, or HIP. These represent both shared and distributed memory programming models, as well as GPU programming models. The prompts for each execution model are designed to be as similar to the other prompts for that problem as possible, while still being idiomatic for the programming model. For serial, OpenMP, MPI, and MPI+OpenMP prompts, we use STL data structures such as `std::vector` and `std::array`. For Kokkos, we utilize the `Kokkos::View` data structure (as shown in Listing 1). The CUDA and HIP prompts use raw pointers to represent array structures.

We list an example prompt in Listing 1 for a variant of a scan problem to generate Kokkos code. The goal of this problem is to compute the minimum value of the array up to each index. We include example inputs and outputs in the prompt as this can significantly improve the quality of the generated code [7]. The necessary `#include` statements are also prepended to the prompt as we found that this improves the likelihood of the LLM correctly using the required programming model.

5 DESCRIPTION OF EVALUATION EXPERIMENTS

Now that we have described the prompts in the previous section, we describe how we can use them to evaluate the performance of LLMs on two different tasks – code generation and translation.

```
#include <Kokkos_Core.hpp>

/* Replace the i-th element of the array x with the minimum
   value from indices 0 through i.
   Use Kokkos to compute in parallel. Assume Kokkos has
   already been initialized.
   Examples:

   input: [8, 6, -1, 7, 3, 4, 4]
   output: [8, 6, -1, -1, -1, -1, -1]

   input: [5, 4, 6, 4, 3, 6, 1, 1]
   output: [5, 4, 4, 4, 3, 3, 1, 1]
*/
void partialMinimums(Kokkos::View<float*> &x) {
```

Listing 1: An example *Scan* prompt for Kokkos. The LLM will be tasked with completing the function body.

5.1 Experiment 1: Parallel Code Generation

The first experiment studies the ability of LLMs to *generate* code, either sequential or in a specific parallel programming model, given a simple description in a prompt (see Listing 1). We evaluate LLMs on how well they can generate code for all the prompts in ParEval. We do so by asking the model to complete the function started in the prompt, and then evaluating the generated code. By compiling and executing the generated code, we report different metrics that will be described in Section 7. The metrics are computed over the combined results from the OpenMP, MPI, MPI+OpenMP, Kokkos, CUDA, and HIP execution models, and compared with the same metrics computed over the serial results. These results will provide insight into how well the model can write parallel code based on natural language descriptions. The results can also be compared along the axes of execution model and problem type.

5.2 Experiment 2: Parallel Code Translation

The second experiment studies the ability of LLMs to effectively *translate* code provided in one execution model to another execution model. To accomplish this, we prompt the LLM with a correct version of the code in one execution model and ask it to translate it to another execution model. An example of this prompt format is shown in Listing 2. We evaluated several prompting formats for translation, such as giving examples of other successful translations, but found the format in Listing 2 to be the most effective.

In theory, we could have evaluated translation capabilities between each pair of execution models for each problem. However, to limit the quadratic increase in the number of prompts, we only evaluate translations for these pairs: serial → OpenMP, serial → MPI, and CUDA → Kokkos. We identify these as some of the most relevant translation tasks for HPC developers. We compute the same metrics as for Experiment 1. These results will provide insight into how well the model can translate between different execution models. The results can also be compared along the axes of source and target execution model and problem type.

Table 1: Descriptions of the twelve problem types in ParEval. Each problem type has five concrete problems, and each problem has a prompt for all seven execution models.

Problem Type	Description
Sort	Sort an array or sub-array of values; in-place and out-of-place.
Scan	Scan operations, such as prefix sum, over an array of values.
Dense Linear Algebra	Dense linear algebra functions from all three levels of BLAS.
Sparse Linear Algebra	Sparse linear algebra functions from all three levels of BLAS.
Search	Search for an element or property in an array of values.
Reduce	Reduction operation over an array dimension, such as computing a sum.
Histogram	Binning values based on a property of the data.
Stencil	One iteration of 1D and 2D stencil problems, such as Jacobi relaxation.
Graph	Graph algorithms, such as component counting.
Geometry	Compute geometric properties, such as convex hull.
Fourier Transform	Compute standard and inverse Fourier transforms.
Transform	Map a constant function to each element of an array.

Table 2: The models compared in our evaluation. CodeLlama and its variants currently represent state-of-the-art open-source LLMs and GPT represents closed-source LLMs. OpenAI does not publish the numbers of parameters in their models.

Model Name	No. of Parameters	Open-source Weights	License	HumanEval† (pass@1)	MBPP‡ (pass@1)
CodeLlama-7B [40]	6.7B	✓	llama2	29.98	41.4
CodeLlama-13B [40]	13.0B	✓	llama2	35.07	47.0
StarCoderBase [29]	15.5B	✓	BigCode OpenRAIL-M	30.35	49.0
CodeLlama-34B [40]	32.5B	✓	llama2	45.11	55.0
Phind-CodeLlama-V2 [38]	32.5B	✓	llama2	71.95	—
GPT-3.5 [8]	—	✗	—	61.50	52.2
GPT-4 [33]	—	✗	—	84.10	—

†HumanEval results are from the BigCode Models Leaderboard [1], except for GPT-3.5 and GPT-4 which are from [3].

‡MBPP results are from [40].

6 MODELS USED FOR COMPARISON

We choose to compare several state-of-the-art open-source and closed-source LLMs, as well as smaller LLMs that are more practical for use in production. We provide brief descriptions of the LLMs used in our evaluation, and their properties below. Table 2 provides a summary and some salient properties of the models used.

CodeLlama (CL-7B, CL-13B, and CL-34B). Rozière et al. originally introduced CodeLlama models in [40] as variants of the Llama 2 model [43], fine-tuned for code. All three models started with Llama 2 weights and were then fine-tuned on 500 billion tokens from a dataset of predominantly code. The Llama 2 models were also extended to support longer context lengths of 16k and infilling to generate code in the middle of sequences. We select these models as they are amongst the top performing open-source LLMs. Additionally, the CodeLlama models are very accessible as there are small model sizes available and there exists a thriving software ecosystem surrounding Llama 2 based models.

StarCoderBase. The StarCoderBase model [29] is a 15.5B parameter model trained on 1 trillion tokens from The Stack [27]. In addition to code from 80+ programming languages, its data set includes natural language in git commits and Jupyter notebooks. StarCoderBase supports infilling as well as a multitude of custom tokens specific to code text data. The model architecture is based on the SantaCoder model [6], and it supports a context length of 8K tokens. We select StarCoderBase as it is one of the best performing open-source models around its size, and is frequently used for comparisons in related literature.

Phind-CodeLlama-V2. The Phind-CodeLlama-V2 model [38] is a CodeLlama-34B model fine-tuned on over 1.5 billion tokens of code data. At the time we were selecting models for comparison it topped the BigCode Models Leaderboard [1] among open-access models on HumanEval with a pass@1 score of 71.95. However, the fine-tuning dataset for this model is not publicly available, so it is not possible to ensure that the BigCode benchmarks themselves are not included in Phind's fine-tuning dataset.

GPT-3.5 and GPT-4. GPT-3.5 and GPT-4 are closed-source LLMs from OpenAI [8, 33]. Most information about these models is not publicly available, however, they can be used for inference via a paid API. We use the most up-to-date versions of these models available at the time of writing, the *gpt-3.5-turbo-1106* and *gpt-4-1106-preview* models. Unlike the other models tested, these are instruction-tuned and aligned to human preferences. Rather than using them for

```
// A serial implementation of sumOfMinimumElements
/* Return the sum of the minimum value at each index of vectors
   x and y for all indices.
   i.e. sum = min(x_0, y_0) + min(x_1, y_1) + min(x_2, y_2) + ...
   Example:

   input: x=[3, 4, 0, 2, 3], y=[2, 5, 3, 1, 7]
   output: 10
*/
double sumOfMinimumElements(std::vector<double> const& x,
    std::vector<double> const& y) {
    double sum = 0.0;
    for (size_t i = 0; i < x.size(); ++i) {
        sum += std::min(x[i], y[i]);
    }
    return sum;
}

// An OpenMP implementation of sumOfMinimumElements
/* Return the sum of the minimum value at each index of vectors
   x and y for all indices.
   i.e. sum = min(x_0, y_0) + min(x_1, y_1) + min(x_2, y_2) + ...
   Use OpenMP to sum in parallel.
   Example:

   input: x=[3, 4, 0, 2, 3], y=[2, 5, 3, 1, 7]
   output: 10
*/
double sumOfMinimumElements(std::vector<double> const& x,
   std::vector<double> const& y) {
```

Listing 2: An example prompt given to the model for code translation. The model is given a sequential implementation of sumOfMinimumElements and tasked with translating it to OpenMP.

direct code generation, we have to interact with them via a chat interface. As with the Phind-CodeLlama-V2 model, the data used to train these models is not publicly available, so it is difficult to fairly compare them with the other models as they might have seen some prompts during training.

7 EVALUATION METRICS

It is important to be able to meaningfully compare the performance of the selected LLMs at generating correct and efficient code for the prompts in ParEval. This section details how we accomplish this by adopting a popular correctness metric for code LLMs, and defining two new performance-related metrics.

7.1 Metric for Correctness

We adopt the pass@k metric from [13] to quantify correctness of the generated code. For a given prompt, pass@k estimates the probability that the model will generate a correct solution given k attempts. Often the average pass@k over all prompts in a benchmark is reported. To estimate the pass@k over a set of prompts, we first generate N samples for each prompt using the model, where $N > k$. These samples are then evaluated for correctness. The number of correct samples can be used to estimate the pass@k value as shown in Equation (1).

$$\text{pass@}k = \frac{1}{|P|} \sum_{p \in P} \left[1 - \binom{N - c_p}{k} \Big/ \binom{N}{k} \right] \tag{1}$$

Number of samples generated per prompt; Number of correct samples for prompt p; Set of prompts

This metric provides insight into how often do models generate correct code. The probability that the model will generate a correct solution in one attempt, pass@1, is the most useful metric for end-users as it aligns with how LLMs are used in practice. In this paper, we report 100×pass@k as is common in related literature and online leaderboards [1, 12]. Additionally, as models have become more capable, studies have shifted toward only reporting pass@1 values. However, pass@k values for $k > 1$ are still useful for understanding how models perform on more difficult prompts. Commonly reported values of k are 1, 5, 10, 20, and 100. It is also common to report pass@1 values using a generation temperature of 0.2 and pass@k for higher values of k using a generation temperature of 0.8. This higher temperature allows the model to more extensively explore the solution space when generating a larger number of attempts.

7.2 Performance Metrics

For parallel and HPC code, it is important to consider both the correctness and performance of the generated code. To analyze and compare the runtime performance of LLM generated code, we introduce two new metrics: speedup$_n$@k and efficiency$_n$@k.

speedup$_n$@k. The first metric, speedup$_n$@k, measures the expected best performance speedup of the generated code relative to the performance of a sequential baseline (see Section 8.2) if the model is given k attempts to generate the code. The relative speedup is computed based on the execution time obtained using n processes or threads. For a given prompt p, the expected best speedup relative to a sequential baseline, T_p^*, is given by Equation (2).

$$\mathbb{E}\left[\max\left\{\frac{T_p^*}{T_{p,s_1,n}}, \ldots, \frac{T_p^*}{T_{p,s_k,n}}\right\}\right] = \sum_{j=1}^{N} \frac{\binom{j-1}{k-1}}{\binom{N}{k}} \frac{T_p^*}{T_{p,j,n}} \tag{2}$$

runtime of sequential baseline for prompt p; runtime of sample j of prompt p on n resources

To demonstrate that Equation (2) represents the desired quantity, consider the set of N generated samples is in order from slowest to fastest. This is without loss of generality as we assume the k samples are selected uniformly and, thus, all size k permutations are equally likely. The probability that the max is the jth sample is given by $\binom{j-1}{k-1}/\binom{N}{k}$, as there must be $j-1$ elements before j and, thus, $\binom{j-1}{k-1}$ ways to select the remaining elements. The sum of these probabilities, each weighted by their respective speedups, gives the expected max speedup over k samples. Taking the average of Equation (2) over all prompts we can define the speedup$_n$@k metric as shown in Equation (3).

$$\text{speedup}_n\text{@}k = \frac{1}{|P|} \sum_{p \in P} \sum_{j=1}^{N} \frac{\binom{j-1}{k-1}}{\binom{N}{k}} \frac{T_p^*}{T_{p,j,n}} \tag{3}$$

For a single LLM, the speedup$_n$@k metric can be used to understand how well its generated code performs compared to sequential baselines. A value greater than 1 indicates that the generated code

is faster than the baseline on average, while a value less than 1 indicates that the generated code is generally slower than the baseline. When comparing multiple LLMs, a higher value of speedup$_n$@k signifies more performant code. It is important to note that this metric is hardware dependent and, thus, to compare models fairly all the run times need to be collected on the same hardware.

The speedup$_n$@k metric also gives insight into how well the generated code makes use of parallelism in its computation. It is fixed to a given number of resources, n, which can either be threads or processes, depending on the model of parallelism being used. It also adds another axis to vary when comparing models. When studying a single model, the speedup$_n$@k metric can be compared at different values of n to understand the complete scaling behavior of that model. When comparing multiple models, it is typically most useful to fix n to a single value. One could also average over many values of n, but this risks hiding too much information to be useful.

speedup$_{\text{max}}$@k. We also define a variant of the speedup$_n$@k metric, speedup$_{\text{max}}$@k, as shown in Equation (4), which estimates the maximum speedup over all n and not a fixed resource count.

$$\text{speedup}_{\text{max}}@k = \frac{1}{|P|} \sum_{p \in P} \sum_{\substack{j=1 \\ n \in \text{procs}}}^{N \cdot |\text{procs}|} \frac{\binom{j-1}{k-1}}{\binom{N \cdot |\text{procs}|}{k}} \frac{T_p^*}{T_{p,j,n}} \tag{4}$$

Here procs is the set of resource counts over which the experiments can be performed. For example, if there are 128 hardware cores, procs = 1, 2, 4, 8, 16, 32, 64, 128 processes or threads.

efficiency$_{\text{n}}$@k. To further understand the parallel performance of the generated code, we define the efficiency$_n$@k metric. This metric measures the expected best performance efficiency (speedup per process or thread) if the model is given k attempts to generate the code. This is easily defined by modifying Equation (3) to divide by n as shown in Equation (5). The possible values of this metric range between 0 and 1.0, with 1.0 representing a model that generates code that scales perfectly with the number of processes or threads. This metric is useful for understanding how well the generated code makes use of parallel resources. In addition to efficiency$_n$@k, we also define efficiency$_{\text{max}}$@k in the same fashion as Equation (4).

$$\text{efficiency}_n@k = \frac{1}{|P|} \sum_{p \in P} \sum_{j=1}^{N} \frac{\binom{j-1}{k-1}}{\binom{N}{k}} \frac{T_p^*}{n \cdot T_{p,j,n}} \tag{5}$$

Even though we explore parallel code generation in this paper, these metrics can be used to consider the performance of sequential code generation as well. For example, examining speedup$_1$@k for the HumanEval, MBPP, or DS-1000 benchmarks will lead to a better understanding of how efficient the generated Python code is compared to a human created baseline. Additionally, both performance metrics could be modified to be parameterized by problem size instead of number of processes/threads in order to study the computational complexity of the generated code.

8 EXPERIMENTAL SETUP

This section describes how we generate outputs using each of the LLMs (Section 6) and the prompts in ParEval, and how we evaluated the generated code using the ParEval test harness.

8.1 LLM Inference: Generating Code Output

To generate outputs with the open-source models, we use the HuggingFace library [47] with PyTorch [37] as the backend to load the LLM weights and use them for inference. Specifically, we create a PyTorch Dataset object that wraps the set of prompts and we pass this as input to a Huggingface Pipeline object, which then runs the models in inference mode and generates the outputs. We do these runs on a single NVIDIA A100 80GB GPU using 16-bit floating point precision. Since the prompt workloads are fairly regular, we get the best inference performance for larger batch sizes. So for each model, we use the largest batch size that fits in GPU memory. To generate the GPT-3.5 and GPT-4 outputs we use the OpenAI API [34] via OpenAI's Python client [35].

For all of the tasks, we use nucleus sampling with a value of $p = 0.95$. Additionally, we limit the maximum number of new tokens generated to 1024. We experimentally found this to be long enough for all of the tasks to be completed, but short enough to limit long, repetitive outputs. Using this configuration, we create two sets of outputs for each model: one with 20 samples per prompt and a temperature of 0.2, and the other with 200 samples per prompt and a temperature of 0.8. The former is used to calculate the metrics at $k = 1$ (such as pass@1) and the latter for larger values of k. This is in line with the generation configurations in related literature [29, 40]. Note that we exclude the evaluation of GPT-3.5 and GPT-4 with 200 samples per prompt and a temperature of 0.8 due to the high monetary cost of generating these outputs.

8.2 Evaluating the Generated Code

To evaluate the generated code, we use the ParEval test harness. The test harness is a set of scripts that compile and run the generated code using manually written test drivers for each problem. The scripts handle recording the compile status, correctness, and execution time of the generated code.

To compile the generated code, we use the GNU Compiler Collection (GCC) version 9.4.0. For serial, OpenMP, and Kokkos versions, we use GCC as the primary compiler, whereas we use it as the backend to the respective frontend compiler for the other models (i.e. the backend compiler to *mpicxx*). All compilations use the flags `-O3 -std=c++17` and the OpenMP tasks add the `-fopenmp` flag. We use version 4.1.0 of Kokkos, and the *threads* execution space, which uses C++ threads for parallelism. MPI codes are compiled with OpenMPI version 4.1.1. CUDA programs are compiled with *nvcc* and CUDA version 12.1.1. Likewise, HIP programs are compiled with *hipcc* and ROCm version 5.7.0.

Before compiling an output, the prompt and generated code are written to a header file that is included by the driver script for that task. Once compiled, the generated binary is run by the test harness. The test harness checks if the generated code produces the same results as the sequential baseline. The sequential baselines are handwritten, optimal implementations of the prompt that are used to test correctness and to calculate the performance metrics (see Section 7.2). Additionally, a code can be labeled as incorrect for the following reasons:

- The code does not compile or it takes longer than three minutes to run. We choose the problem sizes for each prompt such that

any reasonable implementations execute in much less than three minutes.
- The code does not use its respective parallel programming model. For example, if the model generates a sequential implementation rather than using OpenMP when prompted to do so, it is labeled as incorrect. We utilize several string matching criteria to implement this check.

The output of the program includes the result of the correctness check of the generated code, the average runtime of the generated code, and that of the sequential baseline over ten runs. We use the default timer for each execution model to measure its run time.

The CPU runs are conducted on an AMD EPYC 7763, 2.45 GHz CPU with 64 physical cores and 512 GB of RAM. We run with 1, 2, 4, . . . , 32 threads for OpenMP and Kokkos. For MPI, we run with 1, 2, 4, . . . , 512 processes across multiple nodes with one process per physical core. For MPI+OpenMP we run on 1, 2, 3, and 4 nodes with 1 process per node and 1, 2, 4, . . . , 64 threads per node. The CUDA runs are completed on an NVIDIA A100 80GB GPU and the AMD runs on an AMD MI50 GPU. Kernels are launched with the number of threads indicated in the prompt text (i.e. *at least as many threads as values in the array*).

9 EVALUATION RESULTS

We now present detailed results from evaluating the LLMs described in Section 6 using the ParEval prompts and test harness.

9.1 Experiment 1: Parallel Code Generation

> **RQ1** *How well do state-of-the-art LLMs generate parallel code, and which models are the best?*

To evaluate the correctness of the code generated by the LLMs we first look at the pass@1 scores over ParEval. Figure 1 shows the pass@1 score for each LLM for generating the serial code versus the average over the six parallel execution models. As defined in Equation (1), these values are aggregated over all the prompts including problem types and execution models. Notably, all of the LLMs score significantly worse for parallel code generation than they do for serial code generation. The best performing models, GPT-3.5 and GPT-4, both achieve ~76 pass@1 on the serial prompts. This is a strong score in the context of other benchmarks, such as HumanEval, where GPT-4 gets 84.1 (see Table 2). Despite the strong serial scores, GPT-3.5 and GPT-4 only achieve 39.6 and 37.8 pass@1, respectively, on the parallel prompts.

The open-source models show a significant decrease in performance for parallel code generation with all of them except Phind-V2 (Phind-CodeLlama-V2) scoring between 10.2 and 18.6. Phind-V2 does much better than the other open-source models, achieving 32 pass@1 on the parallel prompts. This suggests that further fine-tuning of the open-source code models can improve their performance on parallel code generation. Additionally, it is significant that an open-source model performs near to the closed-source models on parallel code generation. Open-source models are more accessible and, thus, having a strong open-source model for parallel code generation would be beneficial to the community.

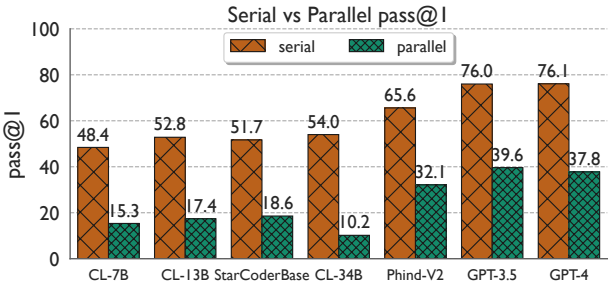

Figure 1: Each LLM's pass@1 score over ParEval. All of the LLMs score significantly worse in generating parallel code than serial code.

Another interesting trend we observe in Figure 1 is that CodeLlama-34B and GPT-4 both score worse than their smaller counterparts on parallel code generation. The reasons for this decrease in performance are not immediately obvious. However, we observe that CodeLlama-34B and GPT-4 often generate the same output for a given prompt for most or all of the 20 samples. This is due to the larger models being more "confident" in their outputs, but this can have an adverse effect on the pass@1 score when the output is incorrect.

Ultimately, the closed-source models are better than the open-source models at parallel code generation. Interestingly, GPT-3.5 beats GPT-4 on the parallel prompts by almost 2 percentage points, suggesting it may be better suited for parallel code generation tasks. This is interesting since GPT-4 is bigger and newer than GPT-3.5 and generally obtains better results on other code and natural language benchmarks. Amongst the open-source models, Phind-V2 has the best results, but still lags behind the closed-source models by almost 8 percentage points.

In addition to pass@1 it is also useful to consider pass@k for $k > 1$ to understand how the LLMs perform provided more attempts at a problem. Figure 2 shows the pass@k for each LLM for $k = 1, 5, 10, 20$ with 200 samples and a temperature of 0.8 for $k \neq 1$. The GPT models are omitted for $k > 1$ due to the monetary cost of generating a large number of samples with these models. We observe the same relative ordering as in Figure 1 is maintained for all values of k with Phind-V2 leading the open-source LLMs. At $k = 20$ Phind-V2 achieves a pass@k of 46 meaning that on average it is able to generate a correct answer to one of the parallel prompts in 20 attempts 46% of the time. The scores of each LLM improving with an increase in k is expected due to the nature of the pass@k metric. The fact that each LLM begins to plateau suggests that there is an upper limit to their ability to generate correct parallel code and giving them more attempts does not significantly improve their performance.

> **RQ2** *Which parallel execution models and problem types are most challenging for LLMs?*

9.1.1 **Breakdowns by Execution Models.** We further break down the pass@1 results by each execution model in Figure 3.

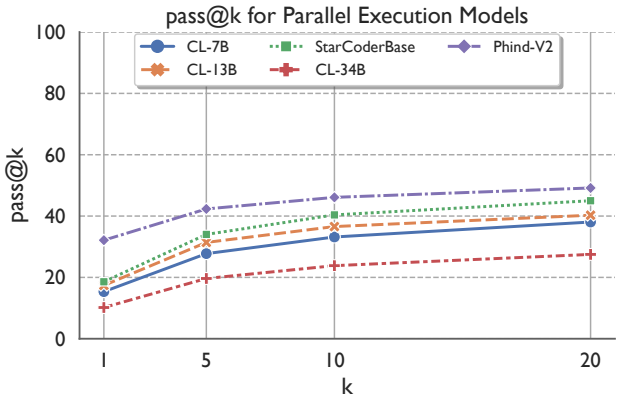

Figure 2: The pass@k for various values of k. The relative order of the LLMs is the same for all values of k with Phind-V2 leading the group.

From this data we observe that every LLM follows a similar distribution of scores across the execution models: serial (best), OpenMP, CUDA/HIP, and MPI/MPI+OpenMP (worst) with Kokkos varying between LLMs.

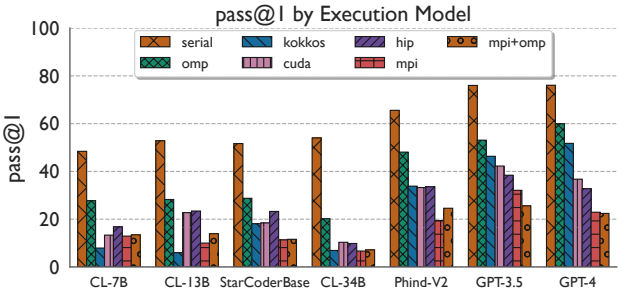

Figure 3: pass@1 for each execution model. The LLMs generally follow the same distribution of scores across the execution models: serial (best), OpenMP, CUDA/HIP, and MPI/MPI+OpenMP (worst) with Kokkos varying between LLMs.

The pass@1 of LLMs being better with OpenMP than other parallel execution models is likely due to the fact that OpenMP code is the most similar to serial code. For many problems it only requires adding an OpenMP pragma, and occasionally a reduction clause. GPT-4 gets nearly as many OpenMP problems correct as serial problems, with an OpenMP pass@1 of 60 vs a 76 serial pass@1. The other top LLMs, GPT-3.5 and Phind-V2, are also nearly as efficient on OpenMP problems as serial problems. StarCoderBase and the CodeLlama models have a larger gap between their serial and OpenMP pass@1 scores, but still have better results on OpenMP than the other parallel execution models.

With the larger LLMs, Kokkos is consistently just behind OpenMP in its pass@1 results. Like OpenMP, Kokkos is a shared memory parallel programming model that relies mostly on high-level abstract constructs to parallelize code. These high-level abstractions make it simpler for the LLM to translate the prompt text to code. The smaller LLMs struggle with Kokkos, likely due to the fact that Kokkos is more verbose than OpenMP and is more niche than the other parallel execution models leading to less inclusion in their training data. With fewer Kokkos examples in the dataset the smaller LLMs likely struggle to learn how to model Kokkos code well.

Following Kokkos, we observe that all the LLMs are next most efficient for CUDA/HIP. These two always have a similar pass@1 score, which is likely due to the similarity of CUDA and HIP. All of the open-source LLMs have a slightly better pass@1 with HIP than CUDA, while the closed-source LLMs are slightly better with CUDA than HIP. CUDA/HIP kernels are more complex than OpenMP and Kokkos, but the parallelism is intrinsic to the kernel making it easier than MPI, since the LLM does not need to reason about large changes to the underlying algorithm.

MPI and MPI+OpenMP are generally the worst parallel execution models for all the LLMs (except for CodeLlama 7B and 13B where they are second and third worst). Compared to the other execution models in our testing, MPI implementations often differ the most from their sequential counterparts. This complexity makes it difficult for the LLMs to generate correct MPI code. Based on the results for all the execution models, we hypothesize that this trend generalizes to all parallel execution models: the more different a parallel programming model's code is from the corresponding serial code, the more difficult it is for the LLMs to generate correct code in that programming model.

9.1.2 **Breakdowns by Problem Types.** In addition to execution models it is also important to understand what types of computational problems LLMs struggle to parallelize. Figure 4 shows the pass@1 score for each problem type across all the LLMs. As a general trend, we observe that all LLMs are better at generating parallel solutions for structured, dense problems and worse for unstructured, sparse problems.

All of the LLMs get their best pass@1 scores for transform problems with the exception of GPT-3.5 where it is the second best. Transform problems are the simplest as they are completely data parallel. In addition to transform, all of the LLMs generally score well on reduction and search. These are also fairly simple to parallelize as searching requires little to no communication and reductions are often offered as high-level constructs in parallel programming models.

Phind-V2 and the GPT LLMs score well on stencil, histogram, and dense linear algebra problems. These problems are all structured and dense, which makes them easier for the LLMs to parallelize. These three problems are in the middle of the group for StarCoderBase and the CodeLlama LLMs coming after transform, search, and reduce. This suggests that the larger LLMs are better at parallelizing these types of problems. Interestingly, StarCoderBase and the CodeLlama LLMs all have graph problems in their top four to five problem types, which is not the case for Phind-V2 and the GPTs.

The bottom five problem types for all of the LLMs are sparse linear algebra, scan, fft, geometry, and sort. GPT-4 is the exception with graph instead of sort as the fifth-worst problem type. Sparse

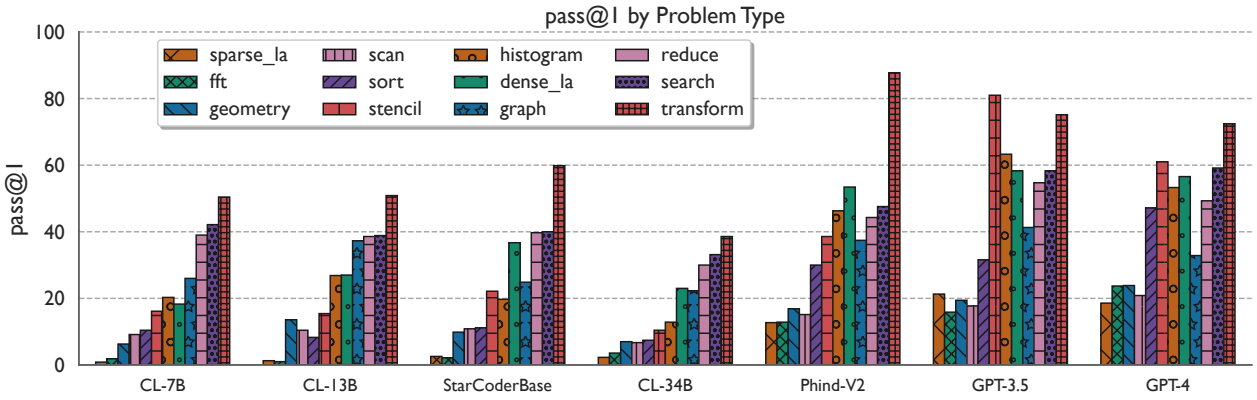

Figure 4: pass@1 for each problem type. The LLMs are best at transform problems, while they are worst at sparse linear algebra problems.

linear algebra is generally the worst problem type, which is likely due to the difficulty in parallelizing sparse computations. FFT and geometry problems are also generally more difficult to parallelize so it readily follows that the LLMs would struggle with them. The sorting and scan results are more surprising. Parallel implementations for sort and scan are well known and certain execution models like OpenMP and MPI even offer high-level abstractions for scan.

Figure 5 provides an even more detailed view of the pass@1 metric across both execution models and problem types for GPT-4. We see the same trends as in Figures 3 and 4 for GPT-4, however, we can also see where certain trends do not hold. For example, despite being the best LLM for search problems and the best LLM at Kokkos, GPT-4 does not do well on Kokkos search problems. We also see that MPI and MPI+OpenMP scores on a particular problem type are not always the same. This suggests that the model has difficulty dealing with these dual execution models.

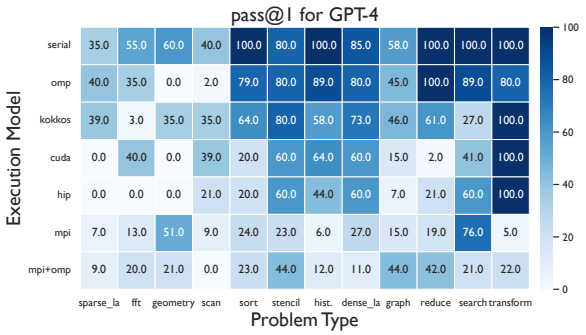

Figure 5: pass@1 for GPT-4 across all execution models and problem types. GPT-4 excels with the Kokkos and OpenMP execution models, while getting more problems correct for transform, search, and reduce problems.

> **RQ3** *How performant and scalable is the parallel code generated by LLMs?*

9.1.3 **Speedup and Efficiency.** When writing parallel code, it is important to consider performance in addition to correctness. A parallel implementation that is correct, but makes inefficient use of resources is not useful in practice. Hence, we compare the speedup$_n$@k and efficiency$_n$@k metrics for each LLM.

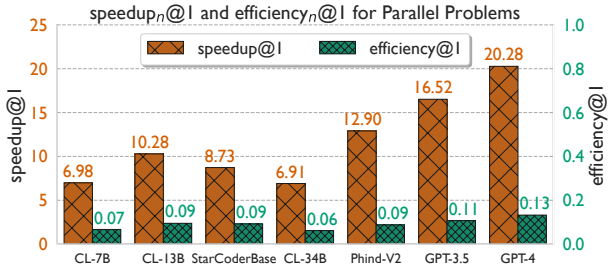

Figure 6: speedup$_n$@1 **and** efficiency$_n$@1 **for parallel prompts. Results are shown for** $n = 32$ threads **for OpenMP and Kokkos,** $n = 512$ ranks **for MPI, and** $n = (4 \text{ ranks}) \times (64 \text{ threads})$ **for MPI+OpenMP. For CUDA/HIP** n **is set to the number of kernel threads, which varies across prompts.** [1]

Figure 6 shows the speedup$_n$@1 and efficiency$_n$@1 scores for each LLM, averaged across the parallel execution models. For comparison, we use the highest value of n for each execution model that we ran in our experimentation: $n = 32$ threads for OpenMP and Kokkos, $n = 512$ processes for MPI, and $n = (4 \text{ processes}) \times (64 \text{ threads})$ for MPI+OpenMP. For CUDA/HIP, n is set to the number of kernel threads, which varies across prompts.[1]

[1] *Search* problems are omitted from speedup$_n$@k and efficiency$_n$@k results due to their high super-linear speedups preventing a meaningful analysis of the performance results for other problem types.

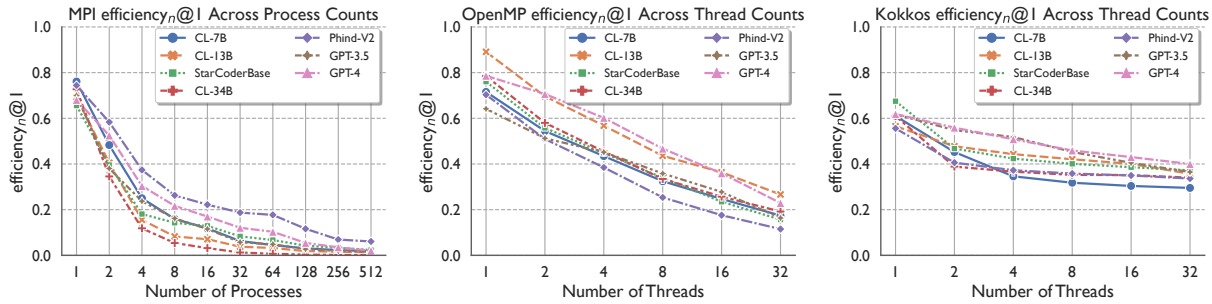

Figure 7: efficiency@1 for MPI (left), OpenMP (middle), and Kokkos (right) prompts across rank and thread counts. Phind-V2 is most efficient for MPI prompts, but is one of the least efficient for OpenMP and Kokkos. GPT-4 is the most efficient for OpenMP and Kokkos prompts. [1]

In Figure 6, we see a trend similar to the pass@1 scores in Figure 1, with the GPT models scoring the highest and the CodeLlama models scoring the lowest. Despite GPT-3.5 having the highest pass@1 for parallel prompts, GPT-4 has the highest speedup$_n$@1 for all parallel execution models at 20.28. This means that on average GPT-4's parallel code achieves a 20.28x speedup over the sequential baseline. To help interpret this result, we also show the efficiency$_n$@1 for each LLM for the parallel prompts in Figure 6. From this we see that none of the LLMs use parallel resources efficiently. The best efficiency$_n$@1 is 0.13 for GPT-4 meaning that on average GPT-4's parallel code achieves 13% of the maximum possible speedup. CodeLlama-34B has the worst efficiency$_n$@1 at 0.06. From the results in Figure 6 we can conclude that the parallel code produced by LLMs is generally inefficient even when correct.

It is also important to consider how efficiency$_n$@1 varies across n. Figure 7 compares the efficiency$_n$@1 curves for MPI, OpenMP, and Kokkos. We see Phind-V2 is the most efficient at MPI prompts, while the least efficient at OpenMP and second to least for Kokkos. GPT-4 produces the most efficient code on average as it is one of the top two most efficient for all three execution models. All of the models start with better efficiency$_n$@1 for OpenMP than Kokkos, but rapidly decline towards an efficiency$_n$@1 of ~0.2. On the other hand, the Kokkos efficiency$_n$@1 values stay roughly consistent across n, showing efficient use of threads.

Figure 8 further shows the expected maximum speedup and efficiency across all resource counts n. We see the same trends as in Figure 6 with the speedups at similar values and the efficiencies higher. This is likely due to a number of the generated code samples plateauing at a certain n, so choosing a smaller n can give a better efficiency with the same speedup.

9.2 Experiment 2: Parallel Code Translation

> **RQ4** *How well can LLMs translate between execution models? How performant and scalable is the translated code?*

In addition to generating parallel code from scratch, we also evaluate the LLMs ability to translate between execution models (see Section 5.2). Figure 9 shows the pass@1 scores for each LLM for translating serial to OpenMP, serial to MPI, and CUDA to Kokkos.

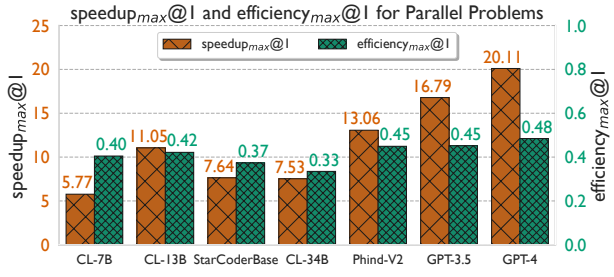

Figure 8: The expected max speedup and efficiency across all resource counts n.

We also include the generation pass@1 scores from Figure 3 for each LLM for OpenMP, MPI, and Kokkos.

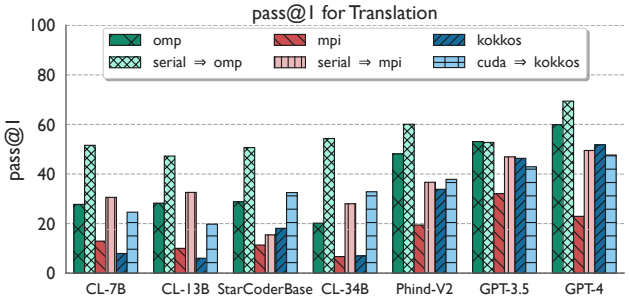

Figure 9: pass@1 for each LLM when translating serial to OpenMP, serial to MPI, and CUDA to Kokkos compared to the pass@1 score for generating code in the destination execution model. The smaller LLMs see a significant improvement when shown an example correct implementation.

Several LLMs score significantly better when given a correct example implementation in a different execution model i.e. translation. All LLMs, except for GPT-3.5, have a higher pass@1 score for translating to OpenMP than they do for generating OpenMP code from scratch. We observe that the LLMs are able to correctly

parallelize the provided serial code with OpenMP. A similar trend emerges with the serial to MPI translation. All of the LLMs score better when translating serial code to MPI than they do when generating MPI code from scratch. Likewise, all of the LLMs see an improvement translating from CUDA to Kokkos over native Kokkos generation with the exception of the GPT models.

It is expected that the pass@1 scores would either increase or stay the same, since the LLM is given more information during translation than when generating code from scratch. It is surprising, however, the magnitude of improvement that the smaller LLMs experience. For example, CodeLlama-7B has a pass@1 of 20 for generating OpenMP code from scratch, but a pass@1 of 52 for translating serial code to OpenMP. This suggests that providing LLMs with correct implementations can improve their ability to generate correct parallel code.

9.2.1 **Speedup and Efficiency.** While translating between execution models improves the pass@1 score it does not generally improve the performance of the generated code as shown in Figure 10. Most LLMs see a similar efficiency$_n$@1 for OpenMP, MPI, and Kokkos whether generating from scratch or translating between execution models. A number of LLMs actually perform worse when translating from serial to OpenMP.

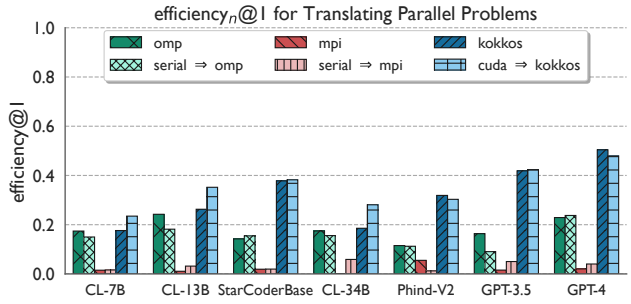

Figure 10: efficiency@1 translation scores compared to generation scores. The LLMs generally score similarly for translation and generation.[1]

We observe similar trends with OpenMP and Kokkos for speedup$_n$@1 as shown in Figure 11. The LLMs generally perform similarly for translation and generation. The exception is MPI where CodeLlama-13B, CodeLlama-34B, and GPT-4 all get significantly better speedup$_n$@1 when translating from serial to MPI code. From the results in Figures 9 to 11 we conclude that providing LLMs with correct implementations in one execution model helps them generate correct code in another execution model, but does not necessarily improve the performance of the generated code.

10 CONCLUSION

In this paper, we proposed a Parallel Code Generation Evaluation (ParEval) benchmark for evaluating the ability of LLMs to generate parallel code. We additionally introduced two novel metrics for evaluating the runtime performance and scaling behavior of the generated parallel code. Using ParEval and these metrics, we have evaluated the ability of state-of-the-art open- and closed-source

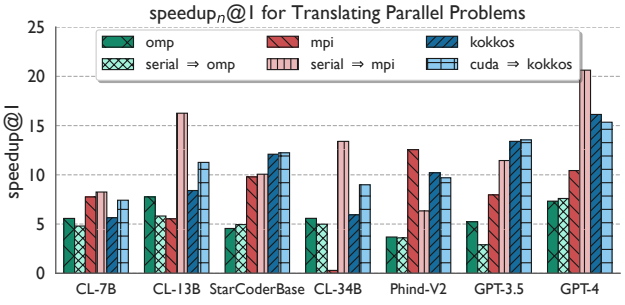

Figure 11: speedup@1 translation scores compared to generation scores. The LLMs generally perform similarly for translation and generation with the exception of MPI.[1]

LLMs to generate parallel code. We find that LLMs are significantly worse at generating parallel code than they are at generating serial code. In particular, we find that LLMs struggle most with MPI code and sparse, unstructured problems. Further, we observe that closed-source models outperform all the open-source models we tested, and that even when LLMs generate correct parallel code, it is often not performant or scalable. Providing correct implementations in one execution model (i.e. serial) helps LLMs generate correct parallel code, but does not necessarily improve the performance or scalability of the generated parallel code.

The poor performance of LLMs on ParEval indicates that further efforts are necessary to improve the ability of LLMs to model parallel code and/or create new LLMs that are specialized for parallel code generation. These LLMs will need to improve both the correctness and runtime performance of their outputs. Benchmarks, such as ParEval, are vital to creating and improving LLMs for parallel code generation. By iterating on ParEval and the metrics we have proposed, we can continue to improve the ability of LLMs in this domain and create state-of-the-art open-source LLMs for different parallel code development tasks.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120, and by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 2236417. This research used resources of the National Energy Research Scientific Computing Center, a U.S. Department of Energy Office of Science User Facility using NERSC award DDR-ERCAP0025593. We spent ~80 dollars for the use of the paid API of GPT-3.5 and GPT-4 for the evaluation in this paper.

REFERENCES

[1] 2023. Big Code Models Leaderboard - a Hugging Face Space by bigcode. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard
[2] 2023. HIP Documentation. https://rocm.docs.amd.com/projects/HIP/en/latest/
[3] 2023. Zero-Shot Replication Framework. https://github.com/emrgnt-cmplxty/zero-shot-replication.
[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. *ArXiv* abs/2005.00653 (2020).
[5] Toufique Ahmed and Prem Devanbu. 2022. Learning code summarization from a small and local dataset. *ArXiv* abs/2206.00804 (2022).

[6] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[7] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[8] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[9] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691. https://doi.org/10.1109/TSE.2023.3267446

[10] Le Chen, Xianzhong Ding, Murali Emani, Tristan Vanderbruggen, Pei hung Lin, and Chuanhua Liao. 2023. Data Race Detection Using Large Language Models. arXiv:2308.07505 [cs.LG]

[11] Le Chen, Pei-Hung Lin, Tristan Vanderbruggen, Chunhua Liao, Murali Emani, and Bronis de Supinski. 2023. LM4HPC: Towards Effective Language Model Application in High-Performance Computing. In *OpenMP: Advanced Task-Based, Device and Compiler Programming*, Simon McIntosh-Smith, Michael Klemm, Bronis R. de Supinski, Tom Deakin, and Jannis Klinkenberg (Eds.). Springer Nature Switzerland, Cham, 18–33.

[12] Mark Chen and et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:arXiv:2107.03374

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:arXiv:2107.03374

[14] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168* (2021).

[15] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. arXiv:2308.01861 [cs.CL]

[16] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2021. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *CoRR* abs/2101.00027 (2021). arXiv:2101.00027 https://arxiv.org/abs/2101.00027

[17] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. PAL: Program-aided Language Models. *arXiv preprint arXiv:2211.10435* (2022).

[18] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. 2022. 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).

[19] William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. 2023. Evaluation of OpenAI Codex for HPC Parallel Programming Models Kernel Generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops (ICPP-W 2023)*. ACM. https://doi.org/10.1145/3605731.3605886

[20] Jian Gu, Pasquale Salza, and Harald C. Gall. 2022. Assemble Foundation Models for Automatic Code Summarization. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2022), 935–946.

[21] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic Similarity Metrics for Evaluating Source Code Summarization. *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)* (2022), 36–47.

[22] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rygGQyrFvH

[23] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. *CoRR* abs/2106.09685 (2021). arXiv:2106.09685 https://arxiv.org/abs/2106.09685

[24] Tal Kadosh, Niranjan Hasabnis, Vy A. Vo, Nadav Schneider, Neva Krien, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, Yuval Pinter, Timothy Mattson, and Gal Oren. 2023. Scope is all you need: Transforming LLMs for HPC Code. arXiv:2308.09440 [cs.CL]

[25] Md Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, and Probir Roy. 2023. An Empirical Study of High Performance Computing (HPC) Performance Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 194–206. https://doi.org/10.1109/MSR59073.2023.00037

[26] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin B. Clement, and Neel Sundaresan. 2022. Learning to Reduce False Positives in Analytic Bug Detectors. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), 1307–1316.

[27] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).

[28] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. arXiv:2211.11501 [cs.SE]

[29] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! (2023). arXiv:2305.06161 [cs.CL]

[30] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. arXiv:2309.07544 [cs.LG]

[31] Christian Munley, Aaron Jarmusch, and Sunita Chandrasekaran. 2023. LLM4VV: Developing LLM-Driven Testsuite for Compiler Validation. arXiv:2310.04963 [cs.AI]

[32] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[33] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[34] OpenAI. 2023. *OpenAI API.* https://platform.openai.com/docs/api-reference/

[35] OpenAI. 2023. *OpenAI Python API library.* https://github.com/openai/openai-python

[36] OpenMP4 2013. OpenMP Application Program Interface. Version 4.0. July 2013.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG]

[38] Phind. 2023. *Phind-CodeLlama-34B-v2.* https://huggingface.co/Phind/Phind-CodeLlama-34B-v2

[39] Cedric Richter and Heike Wehrheim. 2022. Can we learn from developer mistakes? Learning to localize and repair real bugs from real bug fixes. *ArXiv* abs/2207.00301 (2022).

[40] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[41] M. Snir. 1998. *MPI–the Complete Reference: The MPI core.* Mass. https://books.google.com/books?id=x79puJ2YkroC

[42] Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark Gerstein. 2023. BioCoder: A Benchmark for Bioinformatics Code Generation with Contextual Pragmatic Knowledge. arXiv:2308.16458 [cs.LG]

[43] Hugo Touvron et al. 2023. *Llama 2: Open Foundation and Fine-Tuned Chat Models.* Technical Report. arXiv:2307.09288 [cs.CL]

[44] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell,

Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[45] Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F. Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter. 2023. Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation. arXiv:2309.07103 [cs.SE]

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[47] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. Association for Computational Linguistics, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[48] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. *A Systematic Evaluation of Large Language Models of Code.* https://doi.org/10.5281/zenodo.6363556 https://arxiv.org/abs/2202.13169.

[49] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *arXiv preprint arXiv:2302.00288* (2023).