Can Large Language Models Write Parallel Code?

Speaker: Jiang Chang

Author: Daniel Nichols, Joshua H. Davis, Zhaojun Xie,

Arjun Rajaram, Abhinav Bhatele

Background

Large language model (LLM) based coding tools are becoming popular in software development workflows.

1. This tool is deeply involved in all phases of software development.
2. increasing the productivity of software developers.
3. helping to debug and reducing the cost of learning new knowledge.

This makes them a promising tool for improving developer productivity and the overall quality of software.

According to these reasons, why authors did this study?

The Questions Raised By This Article

Why the author did this study?(Problems)

First,

What Is the Parallel programing?

A parallel language is a programming language designed for writing programs that can execute on multiple processors cores.

Advantages of Parallel Languages:

- **Improve performance and efficiency:** Parallel languages make tasks to run on multiple processor cores simultaneously, maximizing computational efficiency and reducing execution time.
- **Handle large-scale computation:** Parallel processing makes it possible to efficiently manage big data, machine learning, scientific simulations, and more, ensuring tasks are completed quickly and at scale.

The Questions Raised By This Article

Even with all the advantages of parallel languages, there are some issues:

Parallel code is difficult to write and optimize Any mistake can lead to bugs like race conditions and deadlocks.

Good news! The LLM, which is very hot right now, can help people solve this problem, but:

If we need to use a tool, we need to understand the performance of the tool in order to use it with confidence. So, we need a Tool for quantify LLM’s ability for generate code in complex takes, but there isn’t tool that can quantify LLM's ability to generate code in complex tasks.

Paper proposed a new Benchmarks for quantify LLM's ability to generate code in complex tasks.

Current benchmarks Still have some issue for evaluate LLM generate Parallel code or Complex Task:

1. **Focus on simple tasks**: Existing benchmarks assess LLMs' ability to generate correct code for simple tasks like arrays or strings.
2. **Lack of parallel code testing**: These benchmarks don't evaluate LLMs' ability to generate parallel code, which is very important.
3. **Does not evaluate performance** : Even correctness is important, but code performance is also important for developers working on parallel programming.

Challenges in Designing a New Benchmarks Specialized for Parallel Languages

1. **Designing new benchmarks is complex**: To fully evaluate LLMs for parallel code generation, testing is required across shared and distributed memory models, different computational problems, and parallel algorithms.
2. **Really hard in testing**: Testing parallel code requires compiling C/C++ code, linking parallel libraries, running it in parallel environments, and selecting appropriate input sizes to assess performance.

But the author overcomes these challenges.

Parallel Code Generation Evaluation (ParEval) Benchmark

Author propose the **Parallel Code Generation Evaluation (ParEval)** benchmark: a set of benchmarks designed to evaluate LLMs‘ performance in generating parallel code. These benchmarks cover seven execution models: **Serial**, **OpenMP**, **Kokkos**, **MPI**, **MPI+OpenMP**, **CUDA**, and **HIP**.

An **execution model** refers to the way a program is executed on computing devices such as CPUs, GPUs, or distributed systems, particularly in parallel computing. It defines how tasks are assigned to multiple processing units (such as threads, cores, or nodes) and how to communication

The seven exciton models:

1. **Serial:** Sequential execution where tasks run one after another on a single core.
2. **OpenMP:** A shared memory multithreading parallel model commonly used for multi-core processors.
3. **Kokkos:** An abstract parallel programming model designed for multiple hardware architectures.
4. **MPI:** A distributed memory parallel model widely used in supercomputing clusters.
5. **CUDA:** A parallel computing model for GPUs, developed primarily by NVIDIA.
6. **HIP:** A GPU parallel computing model by AMD, similar to CUDA.
7. **MPI+OpenMP:** A hybrid parallel model that combines MPI for distributed memory across multiple nodes and OpenMP for shared memory parallelism within each node, often used in high-performance computing environments to maximize resource utilization.

Parallel Code Generation Evaluation (ParEval) Benchmark

The Branchmark also cover twelve different computational problem types: **Sort, Scan, Dense Linear Algebra, Sparse Linear Algebra, Search, Reduce, Histogram, Stencil, Graph, Geometry, Fourier Transform, Transform** and author also use two a strategy for token selection for avoid repetitive, low-quality outputs

Model Temperature:

- **High Temperature**: Produces more varied and creative outputs, ideal for scenarios requiring diversity.
- **Low Temperature**: Yields more conservative and confident outputs, suited for tasks needing precision and reliability.

Nucleus Sampling:

- **Definition**: Nucleus sampling (top-p) selects the next token based on cumulative probability reaching a threshold p, instead of selecting from a fixed top-k tokens.
- **Purpose**: Ensures more diverse outputs, avoiding over-reliance on the top few likely tokens.

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.

Components of ParEval Benchmark

- **Task/Prompt:** A single text prompt given to the LLM to generate code. The generated code can be compiled, executed, and scored either correct or incorrect.
- **Problem:** A set of tasks or prompts designed to test the LLM's ability to generate code for the same computational task, with each task or prompt potentially using a different execution model.
- **Problem Type:** A set of problems used to test computational tasks that involve similar work or come from similar domains (e.g., sorting problems).
- **Benchmark:** A collection of prompts tested together to evaluate the performance of the LLM. We refer to the collection of all prompts we designed as the ParEval benchmark.

Hit: Every computational problem (such as sorting, scanning, etc.) has a prompt (task) that requires the LLM to generate code suitable for that specific problem. Since there are various parallel programming approaches, each problem requires different prompts to different execution models.

Experiment 1: Parallel Code Generation

- This experiment evaluates the ability of LLMs to generate sequential or parallel programming model code. By compiling and executing the code, we compare the results across different execution models and problem types.

```
#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.

Experiment 2: Parallel Code Translation

- This experiment assesses the ability of LLMs to translate code between different execution models. We focus on translating from serial → OpenMP, serial → MPI, and CUDA → Kokkos, and compare results based on the source and target execution models and problem types.

```
   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.

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 [41]	6.7B	✓	llama2	29.98	41.4
CodeLlama-13B [41]	13.0B	✓	llama2	35.07	47.0
StarCoderBase [29]	15.5B	✓	BigCode OpenRAIL-M	30.35	49.0
CodeLlama-34B [41]	32.5B	✓	llama2	45.11	55.0
Phind-CodeLlama-V2 [39]	32.5B	✓	llama2	71.95	—
GPT-3.5 [8]	—	✗	—	61.50	52.2
GPT-4 [34]	—	✗	—	84.10	—

Experiment

- **CodeLlama (CL-7B, CL-13B, CL-34B):** Variants of the Llama 2 model fine-tuned on 500 billion tokens of code data, supporting 16k context length. These models are among the top open-source LLMs and are widely accessible due to their smaller sizes and strong ecosystem.
- **StarCoderBase:** A 15.5B parameter model trained on 1 trillion tokens from The Stack, including code and natural language. It supports code filling and is one of the best-performing open-source models of its size.

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 [41]	6.7B	✓	llama2	29.98	41.4
CodeLlama-13B [41]	13.0B	✓	llama2	35.07	47.0
StarCoderBase [29]	15.5B	✓	BigCode OpenRAIL-M	30.35	49.0
CodeLlama-34B [41]	32.5B	✓	llama2	45.11	55.0
Phind-CodeLlama-V2 [39]	32.5B	✓	llama2	71.95	—
GPT-3.5 [8]	—	✗	—	61.50	52.2
GPT-4 [34]	—	✗	—	84.10	—

- **Phind-CodeLlama-V2:** A version of CodeLlama-34B fine-tuned on 1.5 billion tokens of code data, achieving a pass@1 score of 71.95
- **GPT-3.5 and GPT-4:** Closed-source LLMs from OpenAI, accessed via a paid API. And their training data is not publicly available.

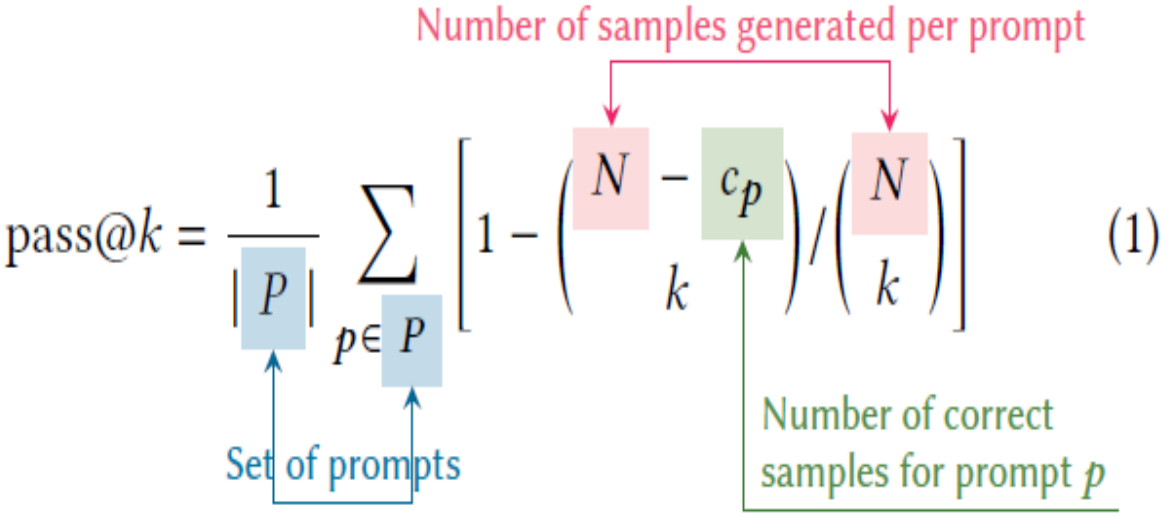
Experiment

Correctness Metric:

- **pass@*k*:** Estimates the probability that the model will generate correct code within *k* attempts.

Temperature Settings:

- **High Temperature (0.8):** Used for higher *k* values in pass@*k*, allowing the model to explore a broader solution space.



Experiment

Author also use the **speedup*n@k*** and **efficiency*n@k*** serve to evaluate the performance of the parallel code generated by LLMs in terms of both speed and resource utilization.

- **speedup*n@k* (Primary Focus)Definition:** This metric measures the expected best speedup of the generated code compared to a sequential baseline. It is based on the execution time using n processes or threads.
- **efficiency*n@k* Definition:** This metric measures the expected best parallel efficiency (speedup per process or thread) of the generated code with k attempts. It ranges from 0 to 1, where 1.0 indicates perfect scalability.

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

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

Experiment Setup

LLM Setup:

- Open-source models are loaded using HuggingFace and PyTorch, running on an NVIDIA A100 80GB GPU.
- Nucleus sampling is used with p=0.95, limiting the maximum generated tokens to 1024.
- Two sets of outputs are generated per model: 20 samples (temperature 0.2) for k=1 evaluation, and 200 samples (temperature 0.8) for larger k values.
- GPT-3.5 and GPT-4 did not undergo 200-sample evaluations due to cost constraints.

Code Evaluation:

- The ParEval test compiles and runs the generated code, recording compile status, correctness, and runtime.
- GCC is used as the main compiler with adjusted flags for different execution models.
- Generated code is compared against hand-written sequential baselines.
- It is marked incorrect if it fails to compile, exceeds a 3-minute runtime, or doesn’t use the specified parallel mode

Experiment 1: Parallel Code Generation Result for How well do state-of-the-art LLMs generate parallel code, and which models perform the best?

- **Point 1**: All LLMs score lower on pass@1 for parallel code than for sequential code. (See Figure 1)
- **Point 2**: GPT-3.5 and GPT-4 drop from around 76 (sequential) to 39.6 and 37.8 (parallel). (See Figure 1)
- **Point 3**: Phind-CodeLlama-V2 is the best open-source model for parallel code, scoring 32, but still trails closed-source models by 8 points. (See Figure 1)
- **Point 4**: Larger models like CodeLlama-34B and GPT-4 score lower due to overconfidence in incorrect outputs. (See Figure 1)
- **Point 5**: Increasing the number of attempts (k) improves scores, but gains plateau quickly. (See Figure 2)

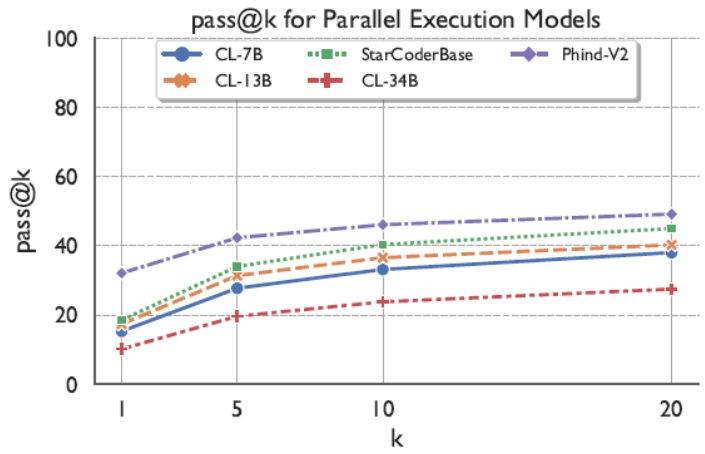


Figure 2: The pass@k for various values of k. The relative order of the LLMs is the same for all values of k with Phind-V2 leading the group.

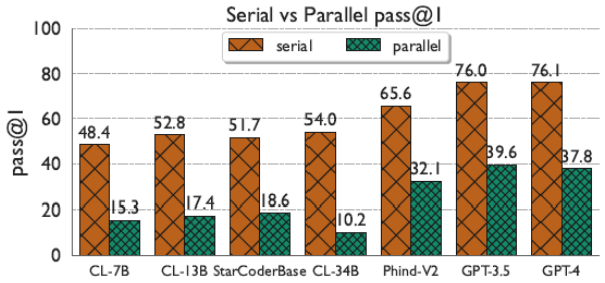


Figure 1: Each LLM's pass@1 score over ParEval. All of the LLMs score significantly worse in generating parallel code than serial code.

Which parallel execution models and problem types are the most challenging for LLMs?

- **Point 1**: LLMs perform best on OpenMP as it closely resembles sequential code. (See Figure 3)
- **Point 2**: LLMs struggle most with MPI and MPI+OpenMP due to their complexity compared to sequential code. (See Figure 3)

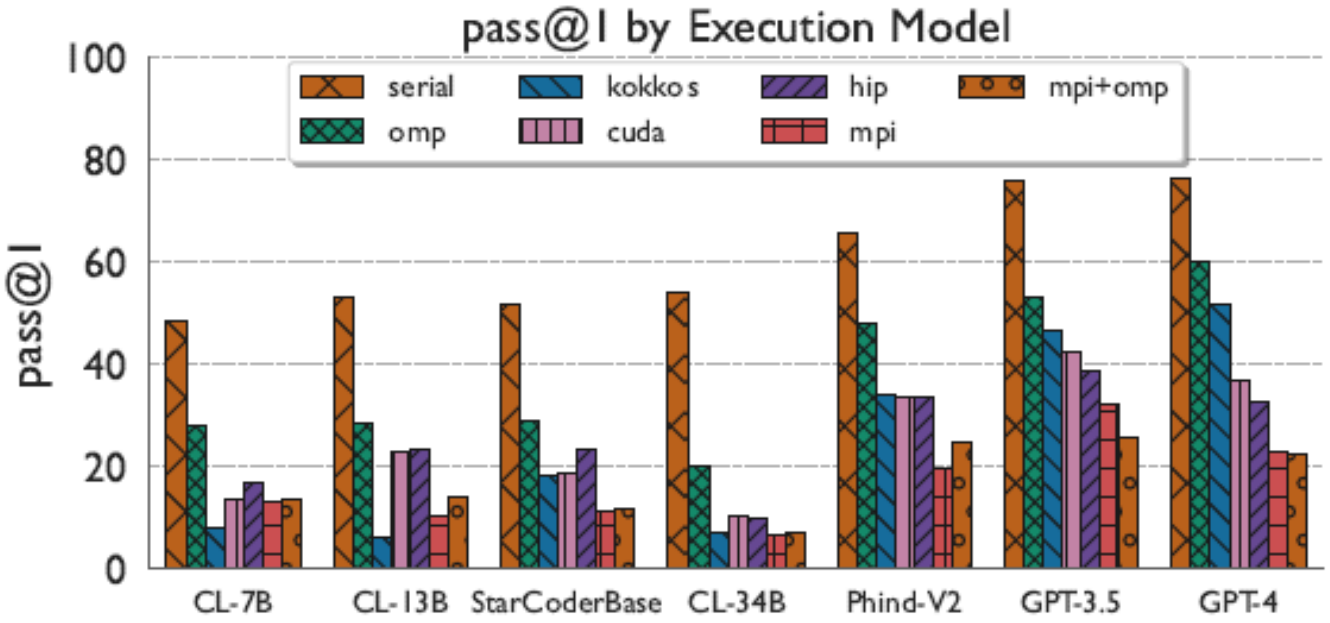


Figure 3: pass@1 for each execution model. The LLMs generally follow the same distribution of scores across the execution models: serial (best), OpenMP, CUDA/HIP, and MPI/MPI+OpenMP (worst) with Kokkos varying between LLMs.

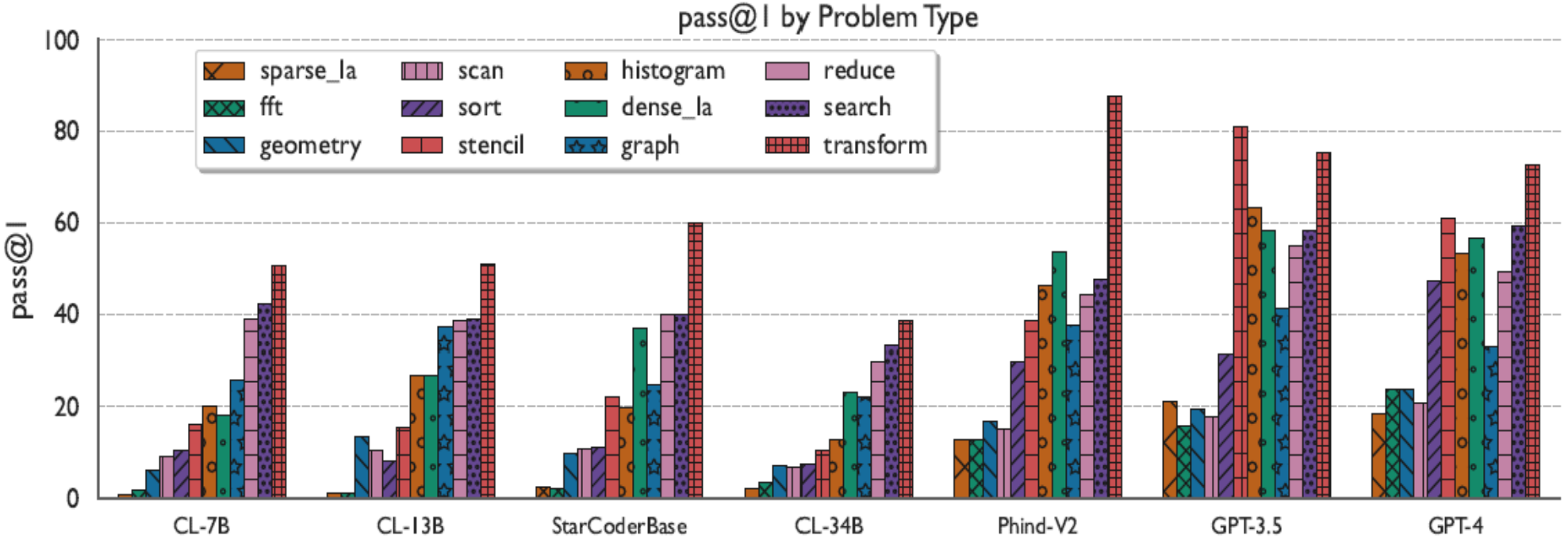


Figure 4: pass@1 for each problem type. The LLMs are best at transform problems, while they are worst at sparse linear algebra problems.

- **Point 3**: LLMs handle structured, dense problems (e.g., transform, reduce, search) well, as they are easier to parallelize. (See Figure 4)
- **Point 4**: LLMs perform worst on sparse linear algebra, scan, FFT, geometry, and sort due to their complexity. (See Figure 4)
- **Point 5**: GPT-4's performance varies significantly across models and problem types, highlighting its limitations with more complex parallel tasks. (See Figure 5)

- **Point 5**: GPT-4's performance varies significantly across models and problem types, highlighting its limitations with more complex parallel tasks. (See Figure 5)

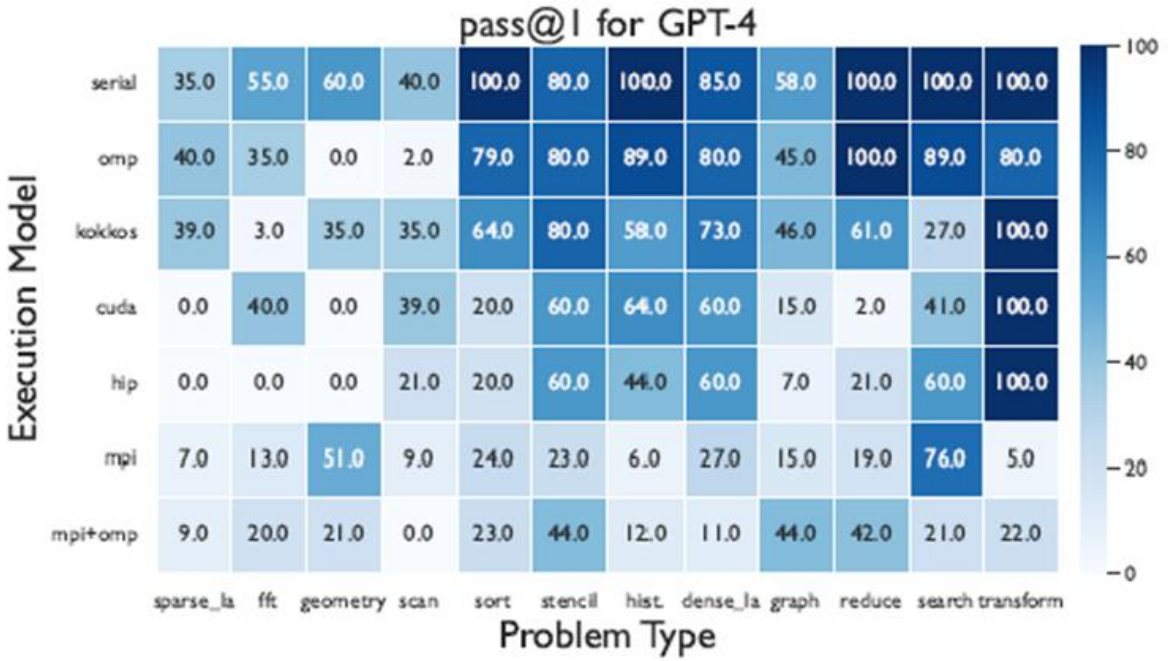


Figure 5: pass@1 for GPT-4 across all execution models and problem types. GPT-4 excels with the Kokkos and OpenMP execution models, while getting more problems correct for transform, search, and reduce problems.

Result for How do LLMs perform in terms of parallel code efficiency and scalability?

- **Point 1**: Some LLMs generate correct parallel code, but the resource efficiency is generally low. (See Figure 6)
- **Point 2**: GPT-4 achieves an average speedup of 20.28x but only 13% efficiency, indicating poor resource utilization. (See Figure 6)

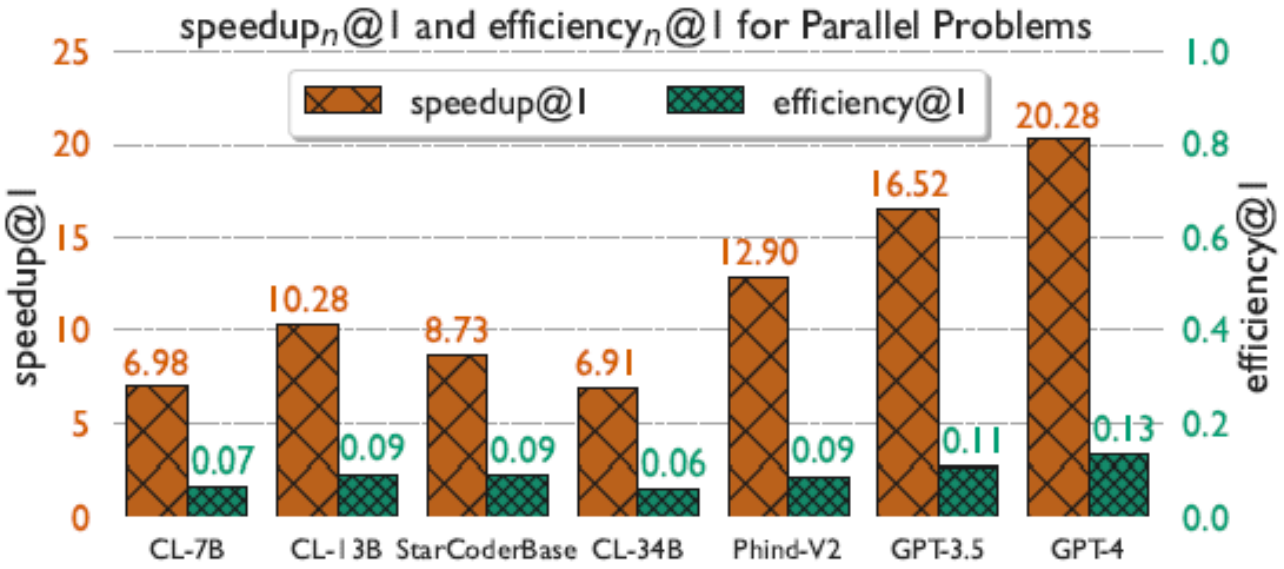


Figure 6: speedup$_n$@1 and efficiency$_n$@1 for parallel prompts. Results are shown for $n = 32$ threads for OpenMP and Kokkos, $n = 512$ ranks for MPI, and $n = (4 \text{ ranks}) \times (64 \text{ threads})$ for MPI+OpenMP. For CUDA/HIP n is set to the number of kernel threads, which varies across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

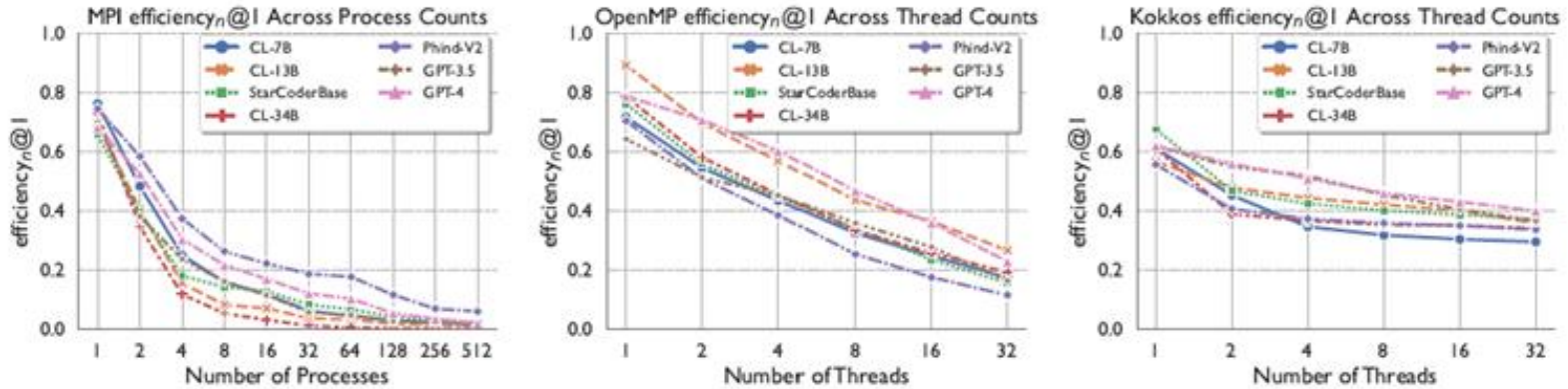


Figure 7: efficiency@1 for MPI (left), OpenMP (middle), and Kokkos (right) prompts across rank and thread counts. Phind-V2 is most efficient for MPI prompts, but is one of the least efficient for OpenMP and Kokkos. GPT-4 is the most efficient for OpenMP and Kokkos prompts. [1]

Point 3: As resource count (n) increases, efficiency declines for all LLMs, especially in OpenMP, where it drops to around 0.2. Kokkos shows more stable efficiency. (See Figure 7)

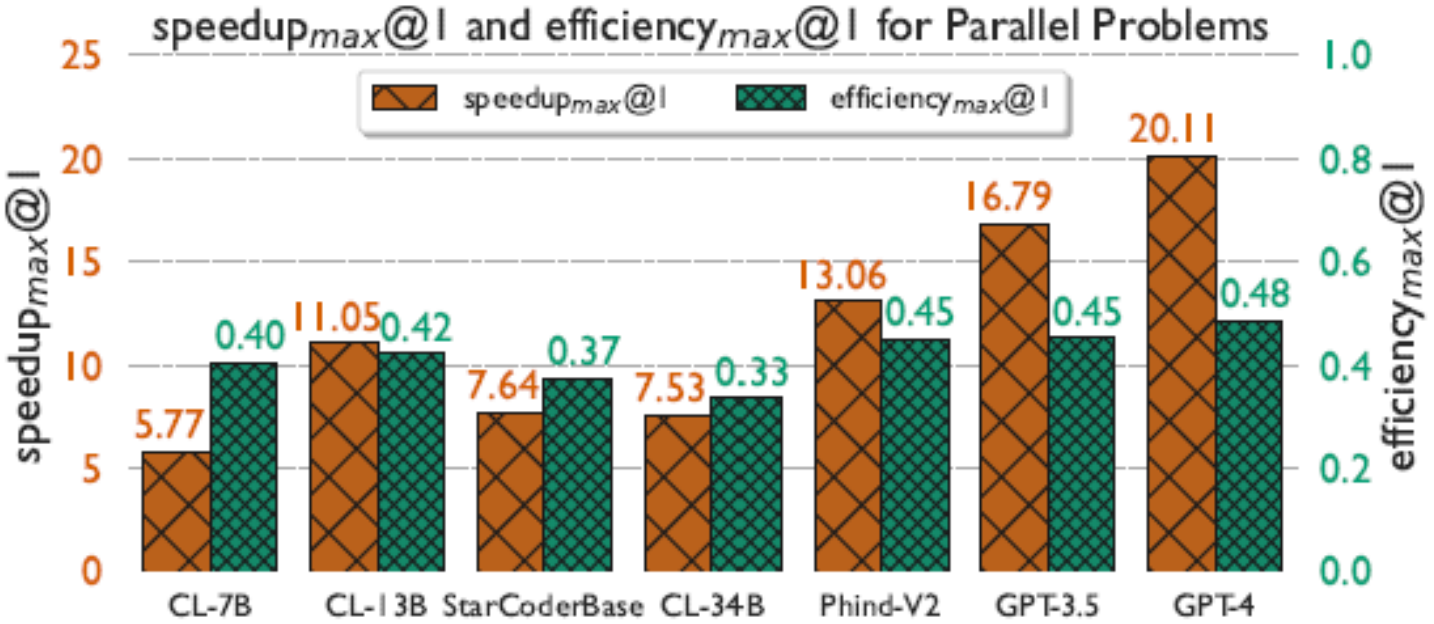


Figure 8: The expected max speedup and efficiency across all resource counts n.

Point 4: Metrics like speedup_max@k and efficiency_max@k suggest better efficiency with smaller n, showing LLMs struggle with large-scale parallelism. (See Figure 8)

Experiment 2 result for Parallel Code Translation

- Better translation performance: LLMs perform better in translating sequential to parallel code than generating parallel code from scratch. (See Figure 9)
- Significant improvement in smaller models: Models like CodeLlama-7B show major gains in translation, with pass@1 increasing from 20 to 52. (See Figure 9)

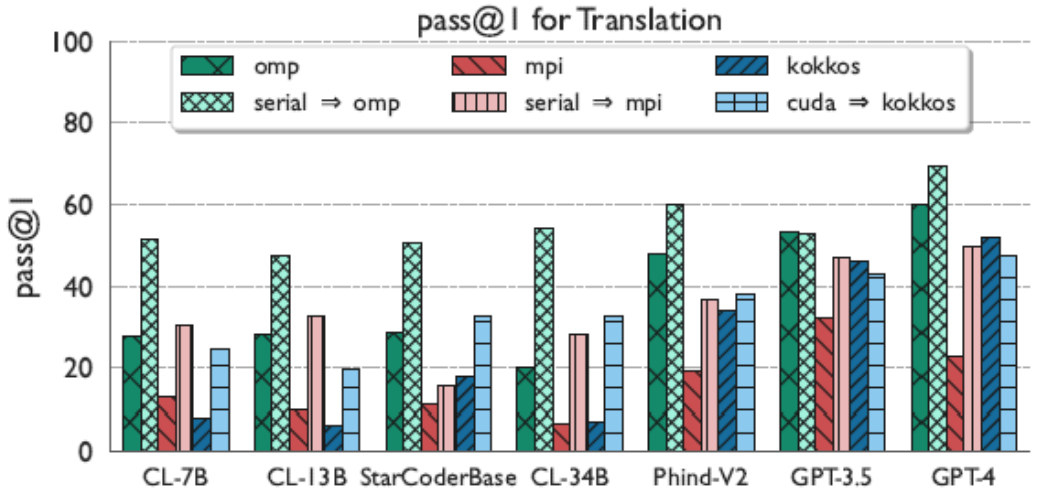


Figure 9: pass@1 for each LLM when translating serial to OpenMP, serial to MPI, and CUDA to Kokkos compared to the pass@1 score for generating code in the destination execution model. The smaller LLMs see a significant improvement when shown an example correct implementation.

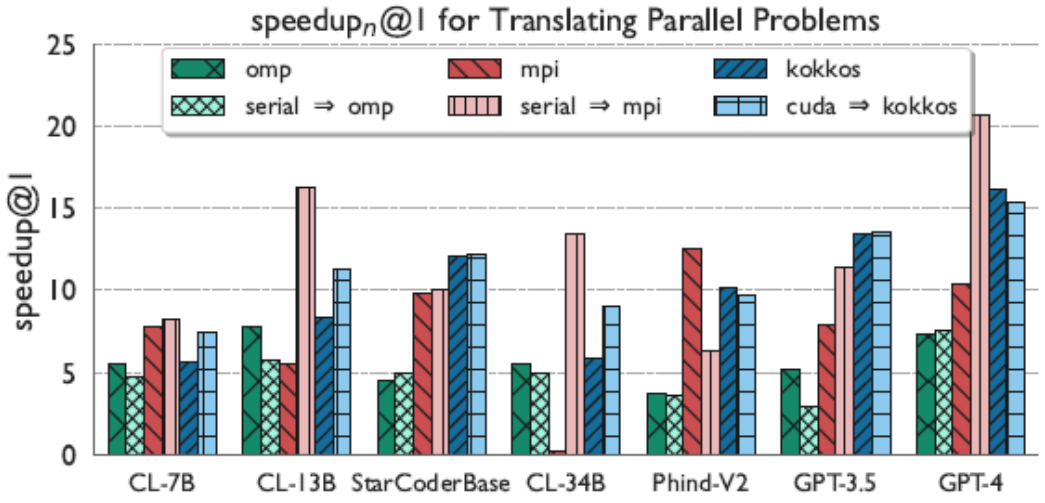


Figure 11: speedup@1 translation scores compared to generation scores. The LLMs generally perform similarly for translation and generation with the exception of MPI.[1]

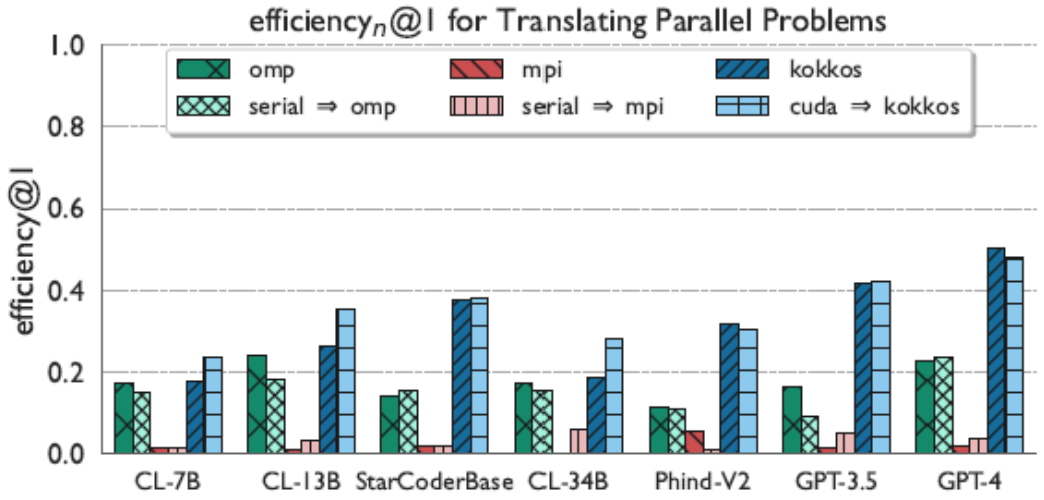


Figure 10: efficiency@1 translation scores compared to generation scores. The LLMs generally score similarly for translation and generation.[1]

Experiment 2 result for Parallel Code Translation

- Limited performance improvement: Translation improves correctness but doesn't significantly enhance performance. (See Figure 10)
- MPI exception: CodeLlama-13B, CodeLlama-34B, and GPT-4 perform better in MPI translation than generation. (See Figure 11)

Conclusion

1. **ParEval Benchmark**: The paper introduces the **Parallel Code Generation Evaluation (ParEval)** benchmark to evaluate LLMs' ability to generate parallel code, along with two new metrics for assessing performance and scalability.
2. **Poor Parallel Code Generation**: LLMs perform significantly worse at generating parallel code compared to sequential code, especially struggling with **MPI code** and **sparse, unstructured problems**.
3. **Closed-Source Models Outperform Open-Source**: **Closed-source models** like GPT-3.5 and GPT-4 outperform all tested open-source models in generating parallel code.
4. **Limited Improvement from Correct Implementations**: Providing correct sequential code helps LLMs generate parallel code but doesn't significantly improve performance or scalability.
5. **Importance of Benchmarks**: Benchmarks like ParEval are important for improving LLMs' ability to generate parallel code, and iterative improvements on these benchmarks and metrics will drive advancements in this field.