Modeling Parallel Programs using Large Language Models

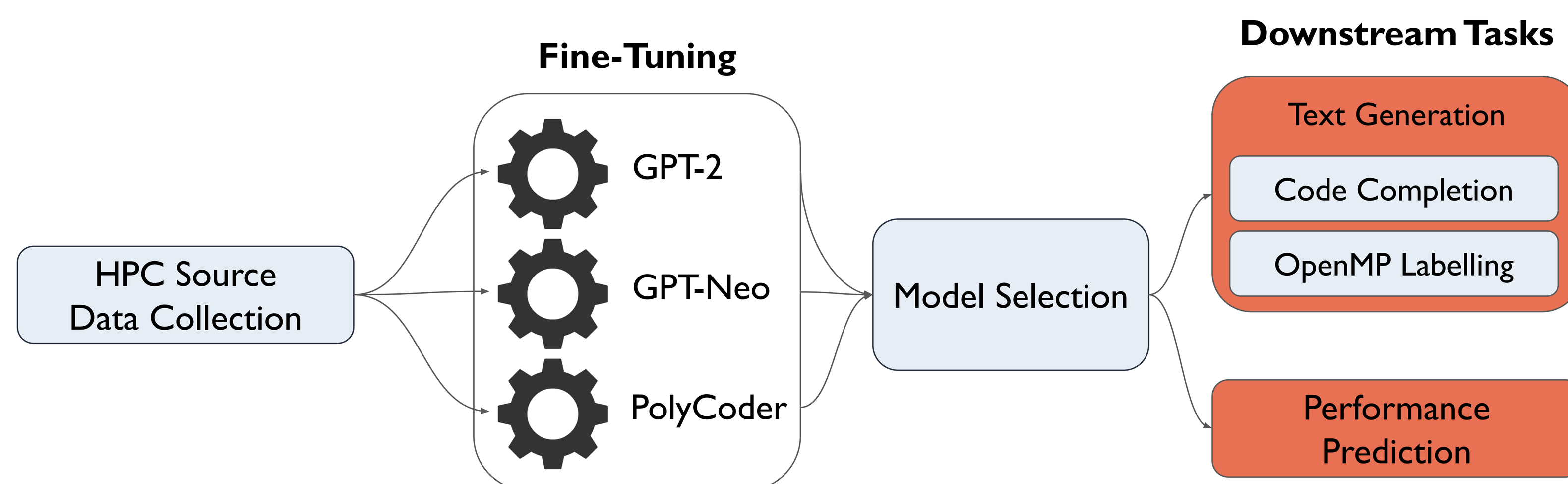
Daniel Nichols[1], Aniruddha Marathe[2], Harshitha Menon[2], Todd Gamblin[2], Abhinav Bhatele[1]

[1]Department of Computer Science, University of Maryland [2]Lawrence Livermore National Laboratory

Abstract

In the past year, a large number of large language model (LLM)-based tools for software development have been released. These tools have the capability to assist developers with many of the difficulties that arise from the ever-growing complexity in the software stack. As we enter the exascale era, with a diverse set of emerging hardware and programming paradigms, developing, optimizing, and maintaining parallel software is becoming burdensome for developers. While LLM-based coding tools have been instrumental in revolutionizing software development, mainstream models are not designed, trained, or tested on High Performance Computing (HPC) software and problems. In this poster, we present a LLM fine-tuned on HPC data and demonstrate its effectiveness in HPC code generation, OpenMP parallelization, and performance modeling.

Overview of Our Approach



HPC Source Data Collection

- Create a dataset of HPC source code
- All GitHub repos with ≥ 3 stars, C/C++/Fortran, and HPC-related tags

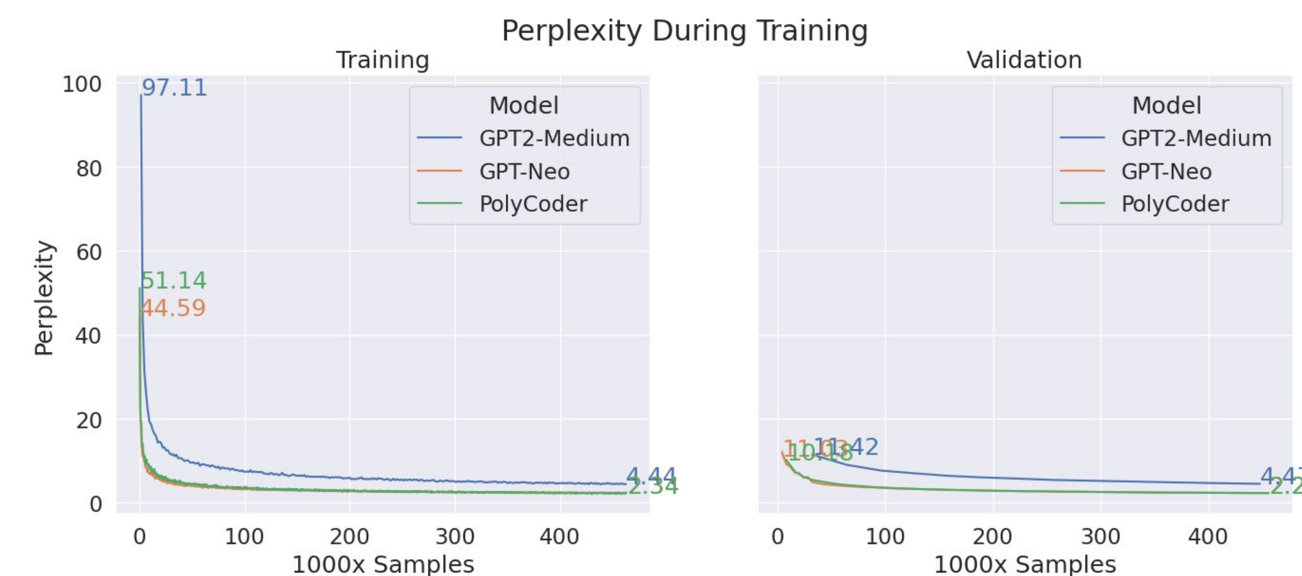
Filter	# Files	LOC	Size (GB)
None	239,469	61,585,704	2.02
Deduplicate	198,958	53,043,265	1.74
Deduplicate + remove small/large files	196,140	50,017,351	1.62

- Deduplicate dataset to improve training performance and prevent overfitting
- Deduplicate by sha256 hash of file contents
- Remove files > 1MB or < 25 tokens
- 18% of files or 0.4 GB are removed

Fine-tuning Large Language Models with HPC data

- Select three LLMs that can run on a single consumer GPU and have a variety of pre-training data: natural language, natural language+code, and code.
- Train models on the HPC dataset for next token prediction.
- Compare models by their validation perplexity after one epoch of training.

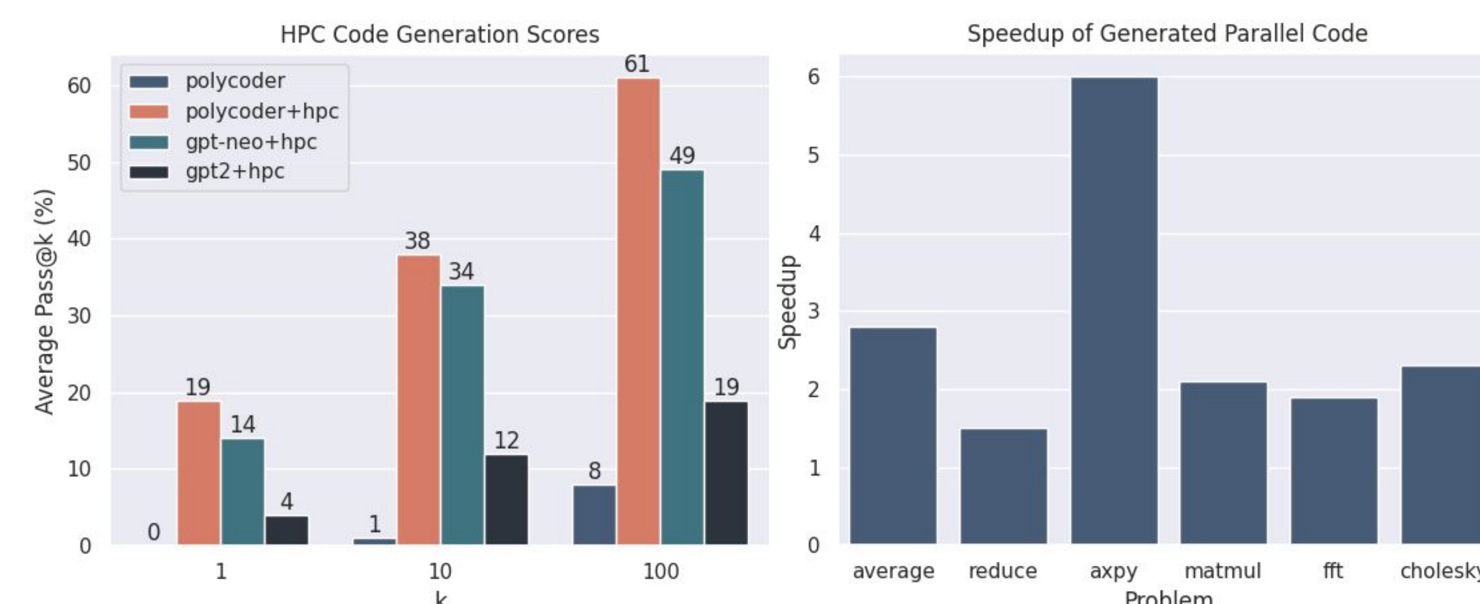
Model	# Param	Layers	Hidden Size	Window Size	Pre-Training Set
GPT-2	1.5B	48	1600	1024	WebText
GPT-Neo	2.7B	32	2560	256	Pile
PolyCoder	2.7B	32	2560	2048	Source Code



- Lower perplexity means the LLM models the language's underlying distribution better
- All three models train to low perplexity after one epoch
- GPT-2 is the worst due to smaller model size (1.5 billion) and a natural language only pre-training task
- PolyCoder performs slightly better than GPT-Neo, however, their training results are comparable
- Results are comparable when pre-trained on source code or source code + natural language

Task 1: Text Generation

- Generate function bodies for 25 custom HPC kernels
- Sequential, OpenMP and MPI kernels
- Compare pass rate: the percentage of functionally correct solutions
- PolyCoder trained on HPC data has the best pass rate
- Standard PolyCoder performs poorly on HPC tasks
- Source code pre-training gives better performance than mixed natural language and code
- Generated parallel code is efficient



Prompt:

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

PolyCoder:

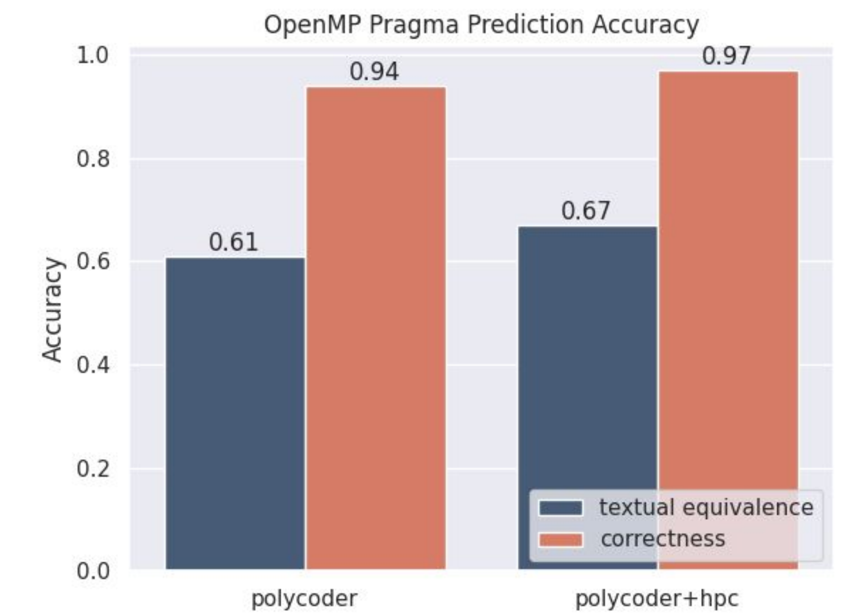
```
    float sum = 0.0f;
    for (int i = 0; i < N; i++)
        sum += X[i];
    return sum;
}
```

PolyCoder+HPC:

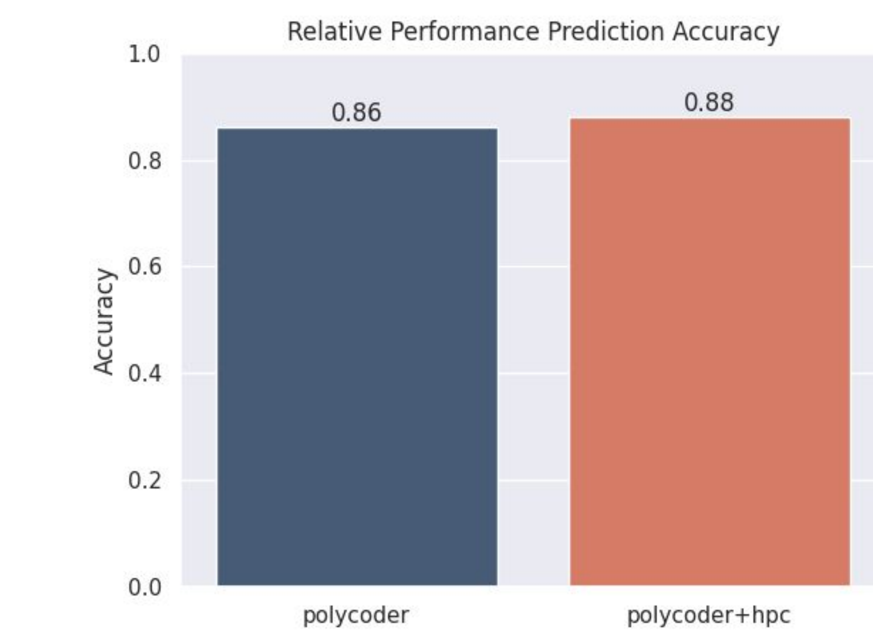
```
    float sum = 0.0f;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < N; i++)
        sum += X[i];
    return sum;
}
```

Task 2: OpenMP Pragma Labeling

- Scrape all OpenMP-labeled `for` loops in dataset
- Fine-tune models to add OpenMP pragmas to `for` loops
- Compare textual equivalence and functional correctness
- PolyCoder fine-tuned on HPC data gets up to 97% of samples correct
- Standard PolyCoder only gets up to 67% correct



Task 3: Performance Modeling



- Build and run every commit in Kripke and Laghos git repositories
- Train models to predict relative performance across code changes
- Classification prediction: positive for improved performance and negative for degraded or same performance
- Both models get high accuracy
- PolyCoder fine-tuned on HPC data scores slightly better than standard PolyCoder

Conclusion and Future Work

- State-of-the-art LLMs that can run on consumer GPUs are bad at HPC tasks
- Fine-tuning LLMs on HPC data can improve parallel code generation
- LLMs understand parallel data movement enough to correctly label OpenMP data clauses
- LLMs can be used to model code performance and perform best at this task when fine-tuned on HPC data
- In the future, we will
  - study larger and better LLMs as they are released rapidly
  - study how well LLMs perform at writing more complicated parallel structures
  - train LLMs to write *faster* code

Acknowledgements

This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-POST-855166)

References

[1] Frank F. Xu, et al., 2022. A systematic evaluation of large language models of code. In International Symposium on Machine Programming (MAPS 2022).
[2] L. Gao et al., The Pile: An 800GB Dataset of Diverse Text for Language Modeling. 2020.
[3] S. Black, et al., GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. Zenodo, 2021. doi: 10.5281/zenodo.5297715.
[4] A. Radford, et al., "Language Models are Unsupervised Multitask Learners," 2019.