ABSTRACT

Title of Dissertation: ON LEARNING BEHAVIORS OF PARALLEL CODE AND SYSTEMS ACROSS MODALITIES

Daniel Nichols

Dissertation Directed by: Professor Abhinav Bhatele
Department of Computer Science

Performance modeling is an integral part of the research process for computational scientists. It enables them to understand how different factors contribute to the final runtime of an application. This understanding is crucial to developing efficient scientific applications and simulations. While important, performance modeling is difficult as there are a large number of factors that may contribute to final performance. Factors such as the algorithm, problem size, implementation, architecture, and systems software stack all impact performance in an often complex relationship. Analytical models can be employed to study these causal variables and performance, however, they are difficult to scale up to a large number of input variables. Additionally, the relationship between the causal variables and performance may be unknown or complex, making it challenging to derive an analytical model. Fortunately, machine learning (ML) can help address these challenges as ML algorithms excel at modeling unknown and complex relationships. Furthermore, ML-based performance models can handle a large number of input variables, making them ideal for modeling complex scientific codes. By training ML mod-

els on historical performance data, computational scientists can develop accurate models that can predict the performance of new applications and simulations under different scenarios. However, current ML-based modeling approaches are limited to modeling one or two sources of performance data, such as hardware counters or application features. This limitation prevents models from making use of all available causal variables that may impact performance. This thesis introduces novel approaches to modeling performance that can make use of all available data sources. Additionally, it introduces performance latent spaces that can be used to model various output metrics, such as runtime or energy consumption, in a unified manner. Finally, a method to integrate these performance models into large language models is introduced to enable modeling and improving the performance of code.

ON LEARNING BEHAVIORS OF PARALLEL CODE AND SYSTEMS ACROSS MODALITIES

by

Daniel Nichols

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2025

Dissertation Committee:

- Professor Abhinav Bhatele, Chair/Advisor
- Professor Alan Sussman
- Professor Hal Daumé
- Professor Johan Larsson
- Professor Martin Schulz
- Dr. Todd Gamblin

Acknowledgements

I was guided and supported by many people during my PhD journey and risk greatly extending the length of this already extensive document if I were to recognize them all. I am incredibly grateful to all who knowingly or unknowingly helped shape my dissertation.

Undoubtedly, the most influential person during my PhD was my advisor, Dr. Abhinav Bhatele, to whom I owe a great deal of gratitude. Alongside his professional and technical mentorship, Abhinav has been supportive of my personal well-being, making starting a PhD during a global pandemic a much less daunting task. Thank you, Abhinav, for your support, advice, and putting up with my sometimes hard-headedness. I am immensely grateful to have been a part of your first cohort of PhD students. I owe my success as a researcher to your advising and will carry your legacy of Gill Sans in my figures for the rest of my career.

I am further indebted to the support and comaraderie of my fellow PhD students — Siddharth Singh, Joy Kitson, Onur Cankur, Josh Davis, Prajwal Singhania, Dalton Hough, and Cunyang Wei — who have been both friends and colleagues. From initial virtual meetings in Gather Town to weekly happy hours and yearly trips to Supercomputing, your friendship and encouragement have been invaluable during my PhD journey and in completing this dissertation. I wish you all joy and success in your future endeavors.

My many research collaborators, particularly those at Lawrence Livermore National Laboratory, have supported me through working on a wide range of projects and assisted me in focusing them to relevant research problems. The collaborations and funding opportunities provided by LLNL allowed me to pursue the problems I was interested in and this dissertation would not have been possible without them.

The many long evenings and weekends would not have been possible without the support and understanding of my family and friends. My family has consistently encouraged me in my educational pursuits, which have led to me completing this dissertation. I promise I will get a job now. I am particularly grateful to Isaac for the many late night discussions over the last several years often scheduled around paper deadlines.

Finally, I have been overwhelmed by the support and encouragement of my soon-to-be wife, Molly. Whether it was listening to my lengthy explanations of research ideas over the phone, finding a good book to read while I finished drafts, or one of the many other ways you kindly supported me, this dissertation would not have concluded without you and I am excited you are waiting on the other side.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

LLM	Large Language Model
HPC	High Performance Computing
GNN	Graph Neural Network
ML	Machine Learning

Chapter 1: Introduction

Performance modeling has become an integral part of the scientific process. In order to develop faster, more efficient code developers need to understand the behavior of their code and how its performance might extrapolate to new run configurations. In order to gain this insight, developers often develop performance models. These are usually analytical, empirical, or simulation based and are used to predict performance metrics such as run time, energy consumption, or memory usage.

In analytical performance models an equation is derived that uses several input variables from the application to predict performance. For instance, one could derive an equation for communication time given the number of messages sent, the size of the messages sent, and the message latency. Such analytical models are great for developing a deeper understanding of the root causes of some performance phenomena. The equations are simple and it is easy to understand their behavior. However, because of this they are quite limited. Only a limited number of variables can reasonably be implemented in the model leading to them leaving out causal variables and being overly simplistic. They also require an expert to design them and this may take a considerable amount of time.

Unlike analytical models, empirical models do not try to come up with direct equations relating run configurations with performance, but rather use historical data to develop statistical

models for performance attributes. In the past decade most advancements here have come through the use of machine learning. Machine learning can model more complex relationships between variables, relationships that may not even be known to the developer. This is also a drawback as we can no longer glean any insight into the underlying behavior from the model. However, the quality and scale of predictions that ML models can make often leads to them being favored over more traditional analytical models.

While ML-based performance modeling is an incredibly effective approach, it comes with all the drawbacks that traditional machine learning has. One of these being *the curse of dimensionality*: the complexity and difficulty in modeling some input domain grows exponentially with the dimensionality of what is being modeled. Additionally, ML approaches that rely on deep learning often require immense amounts of data to train and their predictive performance can be bottlenecked by the amount of quality data available for training. Furthermore, up until recently, many ML methods were designed to model a single modality. For example, if performance engineers wanted to consider source code (text data), hardware counters (tabular data), and calling context trees (graph data) in their models than they needed to use separate models or design an intricate data embedding scheme.

Recent advancements in the field of ML have addressed many of the existing modality constraints and make it now possible to incorporate multiple modalities into performance models. This has the potential to greatly improve the quality and generalizability of performance models as they can consider all available data when making predictions. For example, the model can include source code, input decks, hardware counters, calling context trees, etc. in its input, which gives it full information about what it is modeling.

Building on top of the recent advancements in multi-modal modeling, this dissertation

contributes methods that can model performance metrics using all available input data. The first step to solving this problem requires developing a latent space that represents the distinct run configuration modalities of a code. These include all causal variables that may impact the final performance of a code run: source code, algorithm, inputs and problem size, hardware, resource amounts, and the underlying software stack. This latent space is created using variational auto-encoders to combine latent outputs from models for each modality in an optimal, unified latent space.

Building on top of the unified latent space, this dissertation contributes a means to model multiple output modalities at once, such as run time, calling context trees, memory traces, and energy traces. Each of these are important output modalities to study and are often modeled by themselves. Being able to model them all at once will enable faster, more informative performance studies. To accomplish this, embedding models for each output modality are aligned in a joint latent space such that latent vectors from similar runs have near distance. This latent space can be used to identify similar runs within each modality and make predictions. Such a technique enables faster and better performance modeling with less overall runs.

Finally, this dissertation contributes to training code large language models (LLMs) to align to certain performance characteristics of code. Recent advancements have introduced code LLMs that are exceptional at modeling and generating code. While there are still improvements to be made in terms of the correctness of the generated code, the best LLMs, such as GPT-4, can now solve common code generation benchmarks like HumanEval with up to an 84% pass rate. This work takes this a step further and trains LLMs that can generate *correct* and *performant* code. This is accomplished by aligning the code LLM to the performance latent space.

1.1 Outline of Dissertation

The rest of the dissertation is organized as follows. Chapter 2 provides background information on performance modeling, machine learning, and large language models. Chapter 3 discusses related work and relevant literature that this dissertation is founded upon. Chapters 4 and 5 present preliminary studies in utilizing multiple variables to model performance and integrating the learned model into a HPC batch scheduler. This is followed by Chapters 6 to 8 that present foundational work in developing LLMs that can model parallel code and its performance. Building on these chapters, Chapter 9 introduces a novel technique for aligning code LLMs with a performance latent space so that they generate faster code. Finally, Chapters 10 and 11 introduce approaches for modeling multiple code and metric modalities at once. A summary of the dissertation and future work is presented in Chapter 12.

Chapter 2: Background

This chapter provides background on performance modeling, large language models, and large language models for code.

2.1 Performance Modeling

Performance modeling is the process of predicting the performance of a system based on a set of inputs. These are useful for a variety of reasons, such as predicting the performance of a system before it is built, or for optimizing code to run faster. It can also be used to develop a better understanding of the system's behavior and to identify bottlenecks. One example use case is using sample data, such as hardware counters, from a small scale run on few nodes to predict the performance of a larger scale run on many nodes. This enables developers to optimize their code before running it on a large scale, saving time and resources.

Performance modeling is generally accomplished via analytical, empirical, or simulation-based methods. Analytical models are derivided mathetmatical models that describe the system's behavior. An example of an analytical model is the α-β model for communication, which models the time to send a message between two nodes as a linear combination of the message size, N, and the latency, α, and inverse bandwidth, β, of the network. Given these parameters, sending an N byte message takes $\alpha + \beta N$ time. From this analytical model we can further analyze

complex collective communication algorithms, such as all-to-all or broadcast, to predict their performance. Once an analytical model is derived it can be used to find values that minimize the total time. Such analytical models are extremely useful in that they often provide insight into the system's behavior that is directly interpretable by humans. However, they must be expert derived and are limited in their ability to scale with the complexity of the underlying phenomena being modeled.

Empirical models, on the other hand, are derived from data collected from previous runs. These models generally employ machine learning to use historical data to predict performance. For example, a user could run their program across many different input sizes and record the total energy usage. This data could be used to train an ML model to predict the energy usage of the program for new, unseen input sizes. ML-based performance models are capable of modeling extremely complex behaviors, but they are often considered black-box models as they are difficult to interpret. Furthermore, they require significant amounts of data to train and can be difficult to validate.

Finally, simulation-based models are used to simulate the behavior of a system under different conditions. For example, one can simulate a program running on a CPU to determine its performance. While these simulations can be extremely accurate, they are often computationally expensive and can be difficult to scale to large systems. Furthermore, a significant engineering effort is required to develop the simulation infrastructure. For these reasons, simulation-based models are typically only used to model new or theoretical hardware where empirical data is not available.

2.2 Large Language Models

When applying machine learning to textual data we need a model that takes text as input and, through the process of training on previous data, learns how to predict some property of that text. In recent years such models have been mostly dominated by large transformer-based models. Transformers were first introduced by Vaswani et al. [137]. They are designed to work with sequential data much like recurrent and long short-term memory neural networks. However, they differ in their use of a self-attention mechanism to attribute importance weights to inputs into the model. Due to this mechanism transformers also process entire sequences at once unlike recurrent neural networks.

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

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

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

Generally these networks are trained to model the conditional probability of observing a language token or a sequence of tokens. For instance, given a string of observed tokens

$t_1 t_2 \ldots t_{i-1}$ we may want to find the most likely next token t_i.

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

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

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

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

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

This allows us to train the language model with cross-entropy loss. Minimizing the loss will, in turn, minimize the perplexity. The perplexity is recovered by simply taking the exponential of the loss. It is important to note that perplexity measures model confidence and not

accuracy. However, it has been demonstrated empirically that lower perplexity generally leads to better performance on downstream tasks.

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

Temperature

When sampling temperature controls how *confident* the model is in the sampled token. Lower temperature leads the model to assign more confidence in the most likely tokens in the distribution. On the other end, the model will more uniformly assign confidence across the distribution when the temperature is higher. This term comes from statistical thermodynamics where lower energy states are more frequent with a higher temperature.

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

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

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

Top-k Sampling

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

Nucleus Sampling

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

2.3 Applying LLMs to Code

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

Left-to-Right

Left-to-right or causal language models are trained to predict the most probable next token in a sequence. The model receives and generates text in a left-to-right fashion, which is where it gets its name. This limits the amount of context the model can see as it cannot use later tokens

in its prediction even if they are present in the data. Left-to-right models are useful for text generation related tasks.

Masked

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

Encoder-Decoder

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

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

Additionally, when applying language models to code it is typical to customize the training process slightly to take advantage of the syntactic differences between natural language and code. For instance, the tokenizer, which is responsible for mapping text to a sequence of integers, is often set to group whitespace into single tokens. This is not necessary in natural language inputs

as multiple consecutive spaces are uncommon. However, in code this can meaningfully reduce the sequence size and a formatter can be applied after code generation to regain formatting.

Chapter 3: Related Work

In this section I highlight existing literature in performance modeling using both analytical and machine learning approaches. I present the state-of-the-art in multi-modal machine learning and the use of Large Language Models for code.

3.1 Machine Learning for Performance Modeling

Performance modeling is a well studied research area with lots of literature surrounding analytical and statistical models. Recently, with the increase in machine learning innovations, there has been a large focus on the latter. Machine learning can help model complex relationships between applications and their final performance. It has been used to model job runtimes [152, 145], variability [105], power consumption [26], and many other things [93].

These models are often used to study and understand complex relationships between applications and their performance. Malakar et al. [93] compare the capability of various different machine learning methods on modeling performance. Furthermore, Zhou et al. [153] demonstrate how to extrapolate models from small scale runs to larger scale runs. Many works also use these models in downstream tasks to improve performance. In [65] standard machine learning techniques such as k-Nearest-Neighbors and XGBoost are used to model MPI collective performance and inform auto-tuning decisions. This fits into the broader study of using machine

learning models to more efficiently explore the combinatorial search space in auto-tuning [94, 18, 34]. There are other works that use machine learning models to make informed scheduling decisions on HPC systems such as to reduce variability [105] or avoid IO bottlenecks [144].

There are few works that take advantage of multiple modalities when modeling performance. Dutta et al [43] combine IR and data-flow embeddings to model the performance of HPC applications. The work proves to be very effective, particularly when applied to auto-tuning. However, it fails to make use of all available modalities in its modeling, which limits its applicability to instances where IR is available.

3.2 Large Language Models for Code

A large number of works have looked at applying LLMs to code. One of the first seminal works in this area introduced Codex [31], which is a LLM trained on a large corpus of code from GitHub. Codex is able to generate code from natural language descriptions and is the basis for GitHub's Copilot. Many works have built on top of this by creating their own models [83, 121, 115] or extending techniques to more efficiently generate code with them [63]. Surrounding this work there has been a large number of benchmarks introduced to evaluate the ability of LLMs to generate code [31, 16, 79, 28, 49, 147, 87, 42, 131].

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 [103, 70] or applying existing LLMs to HPC tasks [99, 29, 30]. Nichols et al. [103] 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. [70] 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. [99] evaluate the ability of LLMs to generate compiler verification tests for parallel OpenACC code. Chen et al. [29] use LLMs to identify data races in parallel code and propose the DRB-ML data set, which is integrated into the LM4HPC framework [30].

Currently, there is little work on applying LLMs to performance of code. Garg et al [50] train a masked language model to suggest performance optimization edits to C# code. The model is trained on git commits and is, thus, quite noisy. Furthermore, the model used for training is no longer near the capabilities of current state-of-the-art code LLMs. Another study by Nichols et al [103] attempts to model the performance of code changes using an LLM. While this work is successful in modeling the performance of code changes, it is unable to write or suggest code changes that will improve performance.

Chapter 4: Resource Utilization Aware Scheduling (RUSH)

In this chapter I present work on the RUSH algorithm, which is a novel algorithm for scheduling jobs on a cluster that avoids congestion by using machine learning models that predict job runtime variability. The first main contribution of this chapter is a novel technique for collecting network counters and developing an ML model to predict if jobs are likely to experience variability. The final contribution is a novel batch scheduling algorithm that makes use of the ML model to schedule jobs optimally to prevent variation. It is shown that this algorithm in combination with the trained ML model can reduce the expected variability in jobs and improve the overall throughput of the system scheduler. The work presented in this chapter is published in [105].

4.1 Introduction

Performance variability has become a significant problem for end users, especially as high performance computing (HPC) systems grow in scale and complexity. It refers to the variation in performance (execution time) observed when a given executable is run with the same input parameters multiple times on an HPC system. Users may observe several times worse performance than expected for jobs submitted at different times that are otherwise identical. This can happen due to operating system (OS) noise or contention for shared resources such as the net-

work or filesystem [37, 22]. Performance degradation negatively affects the end user as well as the operational efficiency of the system.

When faced with performance variability, users are unable to estimate run times for their jobs accurately, and hence may request nodes for longer times than may be required. Most HPC systems use batch schedulers such as Slurm [127] and LSF [66] to run jobs and assign allocated resources to them. These batch schedulers require a run time limit provided by the user that serves as an upper bound on the duration the job will be allocated resources. When the end user cannot predict the total run time of their application due to large variances, they will often over-estimate total job time input to the scheduler, which may result in longer queue wait times [76]. On the other hand, if the user underestimates performance degradation, their application may be terminated prematurely resulting in loss of job progress. Both these situations adversely affect resource utilization as well as job throughput. In addition, variability also makes it challenging to analyze the performance bottlenecks in a parallel application, and study the impact of performance improvements made to a code.

The overall operational efficiency of the HPC system also suffers due to performance variability as jobs take longer to complete on average. This results in fewer jobs being completed over time and causes the system's throughput to diminish. Additionally, the scheduler receives less realistic time estimates from users which inhibits its scheduling capability. Hence, it is important to tackle the performance variability problem not just at the individual user level, but at the system level.

With storage becoming relatively inexpensive, the amount of system related data being logged has increased considerably. Software such as the Lightweight Distributed Metric Service (LDMS) [4] are being used to collect and aggregate multiple data streams from system hardware

and software on HPC systems. We believe that such system data holds clues about the performance variability of individual jobs. Moreover, we hypothesize that past historical data can give us a reasonable indication of the performance of jobs in the near future.

In this paper, we use historical job information and system monitoring data to accurately predict if a job in the scheduler queue will experience variation if scheduled right away. We observe that ML models trained on historical data for control jobs perform exceedingly well in predicting if a job in the queue will experience variation. Our models obtain an F_1 score of 0.95 in cross-validation. We use these trained models as an input to the job scheduler to influence scheduling decisions with a goal to reduce variability. Predictions from the trained ML models are used by the scheduling algorithm to delay scheduling of certain jobs in the queue if their run time may vary significantly. We design and implement an end-to-end system, which we call the Resource Utilization aware Scheduler for HPC (`RUSH` in short), for collecting application and system data, accurately modeling and predicting application variation, and intelligent adaptive scheduling based on such predictions.

Using real workloads and an implementation of our scheduling algorithm on a large allocation of the Quartz cluster at LLNL, we show that `RUSH` effectively reduces the variation and maximum run time of applications without significantly affecting makespan or mean queue time. We see up to 5.8% improvement in maximum run time and no performance outliers. In our experiments, we see the average number of runs experiencing variation drop from 17 to 4 using `RUSH`. Additionally, we show that our ML model and scheduler can generalize to applications not included in its training data as well as different inputs for the same applications.

4.2 Data Collection and Modeling

Previous work [144] that used I/O performance predictors has shown that using current information about the file system to delay scheduling of I/O-intensive jobs can improve resource utilization. This shows that the relative health of shared resources is a meaningful predictor in determining if an application will experience performance variation in the near future. It also shows that delaying the execution of an application when shared resources are congested can lead to less variation and higher resource utilization. Thus, we hypothesize that deploying a resource utilization aware scheduler will also improve these two metrics.

An adaptive job scheduler would require *online* knowledge of system health and its relationship with application performance. Existing works have shown that system monitoring data can provide this meaningful insight into the health of shared resources. Moreso, [2] shows that this data in conjunction with historical runs from proxy applications can accurately predict their relative performance. With this information available *apriori*, the job scheduler can alter its queue order to prevent variation and further congestion.

Thus, we collect system data, shared resource benchmarks, and proxy application profiles over time to build statistical models to be used in our scheduling algorithm. Below, we present the data used in our ML pipeline as well as our collection methodology. All of the data was collected on the Quartz system at LLNL. Quartz is a fat-tree cluster with 2,988 Intel Xeon E5-2695 compute nodes, connected by a Cornelis Networks Omni-Path fabric.

4.2.1 System Monitoring Data from the HPC Cluster

Recent years have seen the growth of software stacks to collect and analyze system data. We utilize these to gather information about the state of the machine as proxy applications run. With this data we can infer causes of performance anomalies and predict future occurrences with statistical models.

We include two sources of system counters in our data: sysclassib and lustre_client. Sysclassib is a table of counters containing values for the endpoint traffic such as the xmit rate and recv rate. The lustre_client table of counters contains the number of system calls to the Lustre parallel filesystem as well as the amount of data being written/read. These data are consistently collected by LDMS, which writes the aggregated data into Cassandra tables on the LLNL Sonar system. Each sample in the table is indexed on the hostname of its source node and the timestamp from when it was recorded.

We utilize aggregate data points in training rather than temporal data (see Section 4.2.4). These aggregate data points are calculated by aggregating counter values over some duration before a job is run. In our training data we use five minutes as the duration. The counters are aggregated via minimum, maximum, and mean and, thus, each counter column becomes 3. For example, the `xmit_rate` counter in sysclassib becomes `max_xmit_rate`, `min_xmit_rate`, and `mean_xmit_rate`. This data is also aggregated over compute nodes as it is recorded on each node. In our dataset we include the aggregates over both all compute nodes and the nodes exclusive to the job being run, so that we can compare the results from training the ML models over data from the entire machine versus the nodes exclusive to each data sample.

4.2.2 Proxy Applications Used in Control Jobs

This system data provides insights into the state of the machine, while proxy applications can provide insight into how applications perform on that machine. Proxy applications are simpler programs that mimic the typical workload of a larger scientific code. As a result, they are ideal for generating data that is representative of historical workloads of HPC systems. We run seven proxy applications at frequent intervals to collect performance data: Kripke [78], AMG [61], Laghos [41], SWFFT [10], PENNANT [46], sw4lite [130], and LBANN [45]. These applications represent a range of computational and communication patterns in a variety of scientific domains. Each was compiled with the default build settings in their build documentation using the system intel compiler.

We submitted jobs for each proxy application two to three times a day on the cluster from August 2020 to February 2021 with each job being run at various times in the day. Each application ran on 16 nodes using 512 cores in total. All of the applications use MPI for distributed memory parallelism and run in CPU only mode. Each run was profiled with HPCToolkit [3]. We use Hatchet [21] to read in the HPCToolkit profiles, and extract the inclusive run time of the main compute region in each code. Figure 4.1 shows the variation experienced by each application between November 12th, 2020 and December 31st, 2020 relative to each application's minimum running time. In mid-December, there was a significant spike in variation in all applications.

While all of the applications experienced some degree of variation, they may have different sources of this variation from their types of workload. Thus, the type of workload is included in the dataset as a one-hot encoded vector over compute, network, and I/O intensive. For the training data we hand selected these values. However, in production this data needs to be provided

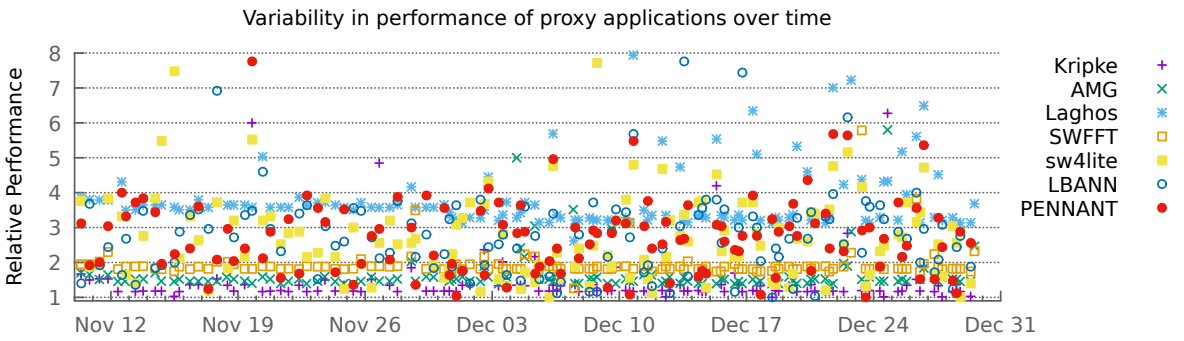


Figure 4.1: Observed variability in the performance of proxy applications run in the production batch queue of Quartz at LLNL over a period of two months in 2020. Performance is relative to the lowest execution time per application. Several applications see over 2× performance variation and some even up to 14×. (Several data points over 8× not shown in the plot.)

accurately to the scheduler. This can be given by the user, empirical methods, or binary analysis.

4.2.3 Benchmarks Used to Monitor System Health

Before and during proxy application runs we collected several metrics related to the health of the system as well as the performance of the job. Right as each job is scheduled we ran two MPI benchmarks with mpiP to gather information about the network health. These benchmarks are used to offer some information to the ML model as to how congestion is currently affecting running applications.

The first benchmark is a simple ring routine with send/recv that passes around a 100MB token for ten iterations. The second calls AllReduce on 100MB of random data for five iterations. Message sizes and iteration counts for these benchmarks were picked empirically such that there was sufficient variance for the ML models to learn from, but not enough to cause significant communication overhead.

Using mpiP we record the time spent waiting on the blocking Send, Recv, and AllReduce calls on each node. For the dataset we record the minimum, maximum, and mean of each of these

values across used nodes. This becomes nine features in each data point.

4.2.4 Input to the Machine Learning Models

Each of these application runs becomes a sample in the final dataset. The input features for each sample consist of the minimum, maximum, and mean of every counter in the sysclassib and lustre_client tables, the user provided application type label, and the nine aggregated benchmark results. Finally, each sample has its run time and z-score as output labels. The resulting dataset and its features are presented in Table 4.1.

Table 4.1: Description of data sources and the number of counters/features derived from them for training the ML models.

Input source	**# Counters**	**# Features**	**Description**
sysclassib	22	66	InfiniBand counters
opa_info	34	102	Omni-Path switch counters
lustre2_client	34	102	Lustre client metrics
MPI benchmarks	3	9	Execution time
Proxy applications	-	1	Compute Intensive
	-	1	Network Intensive
	-	1	I/O Intensive

This collected data is designed to encapsulate the machine state during an application run and the relative performance of that run. The goal of the ML models is to analyze the machine state data and basic information about a job and predict if it will experience variation. Several recent works have explored using various models to learn over system data [123]. We find static models trained on aggregate statistics to work well for our purposes.

4.3 RUSH: Resource Utilization aware Scheduler for HPC

We now present the two main components of RUSH: an ML-based variability predictor, and a model-based adaptive job scheduling algorithm, and also describe the design of the entire pipeline. Figure 4.2 highlights how each component of the pipeline fits together and their input/outputs.

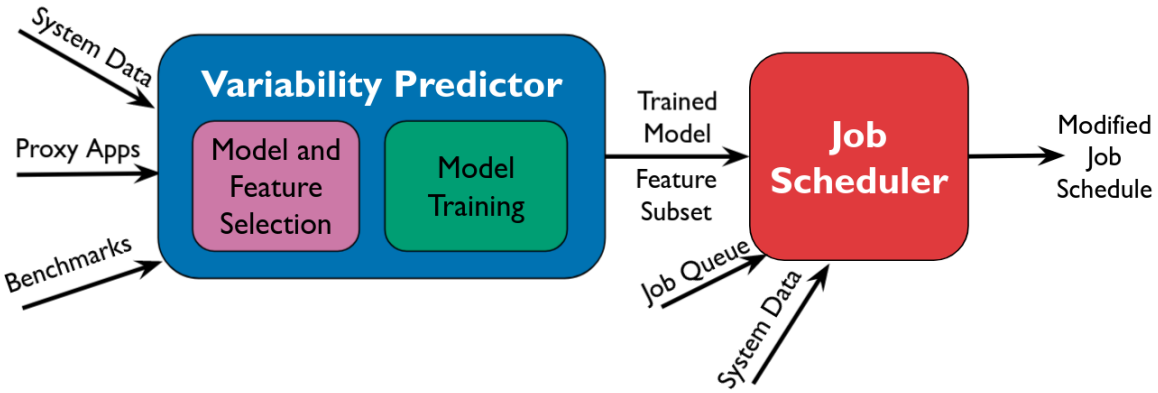


Figure 4.2: Pipeline Overview. The ML model is trained offline on historical jobs and system data. Optimal features are selected and a trained model is exported. This trained ML model, current system data, and submitted jobs are provided as input to the job scheduler that, in turn, decides a new order for scheduling jobs and mapping them to system resources over time.

4.3.1 Variability Predictor Module

The first module in the RUSH pipeline uses system and control job data to predict if variation will occur from running a job on the current system state. There are three inputs to this module: system counters from Sonar, profiles from longitudinal runs of proxy applications, and timings from the MPI benchmarks. Within this module feature and model selection are done first followed by training and exporting the chosen model and features.

The ML models in the first component use the input data as described in Section 4.2. We

set up the ML problem as a classification task with the goal of classifying the occurrence of variation given the system and benchmark data. The input to this model consists of the 282 features listed in Table 4.1. For model and feature selection we use binary classification and set the label of each data as 0 or 1. The first label, 0, is assigned when an application's run time is less than 1.5 standard deviations of its mean run time. This signifies no variation. On the other hand, we assign a label of 1 when the run time is greater than 1.5 standard deviations from its mean. These variations are computed per-application using the mean and standard deviation for each application's run times, but the model is trained on data from all applications. Instead of arbitrarily selecting an ML model we train a variety of models and use their F_1 scores to to compare their performance (see Section 4.4.2).

The set of classifiers used are standard models and we use the best performing in the pipeline (see Section 4.3.3) based on F_1 score. The models used are Extra Trees, Decision Forest, K-Nearest Neighbors, and AdaBoost. Each is trained using stratified cross validation to preserve the imbalance of the data. To cross validate we split the data using six applications for training and one for validation. This is performed over every possible partitioning.

Features are selected after model selection using recursive feature elimination. Features are eliminated recursively and the set with the highest F_1 score are kept. For the Extra Trees and Decision Forest models, which have metrics for feature importance, the least import features are removed first during feature elimination.

After selecting the model and feature set the second component outputs the trained ML model that can be used offline. The chosen model is trained using the same data and k-fold cross-validation. However, this model is trained on three output classes: no variation, little variation, and variation. Here the variation label stays the same while the no variation label is assigned

when an application's run time is less than 1.2 standard deviations from its mean run time. Little variation is when the application's run time is between 1.2 and 1.5 standard deviations of its mean run time. These labels are chosen based on our observations of application performance behavior.

4.3.2 Model-based Adaptive Job Scheduler

The second component of `RUSH` is an intelligent job scheduler that uses the models trained by the variability predictor as input. The scheduler has three inputs: the trained ML model, job queue, and systems data. It uses the ML model with the systems data as input to implement a scheduling policy and map jobs from the queue to system resources.

The proposed scheduler utilizes predictions provided by the ML model to delay scheduling of jobs that will experience variation. We do this by running the ML model on the current system counters whenever a new job is about to be scheduled. If the model predicts variation for this job we skip over it and look at the next one in the queue. The delayed job remains at the top of the queue and will be the first to be considered for scheduling next time resources become available.

Our job scheduling modifications are general enough that they can be used to modify other existing policies. For example, we show that we can easily modify the FCFS+EASY scheduling algorithm presented in Algorithm 3. The main and backfilling policies can be replaced with other queue ordering policies. One common example is Shortest Job First or SJF. This allows `RUSH` to utilize the benefits from other optimal queue ordering policies assuming they work by statically re-ordering the queue.

This algorithmic change only affects the job queue ordering. It is agnostic towards resource mappings and network topology. These can be accounted for in the start function when jobs are

Algorithm 1 Scheduling Algorithm. This standard algorithm queues jobs using policy $\mathcal{R}_1$ and uses EASY to backfill smaller jobs. $Start(\cdot)$ is used to launch jobs when resources become available.

Input $Q \leftarrow$ queue of jobs
$M \leftarrow$ ML model
$S \leftarrow$ current machine state
SkipTable $\leftarrow$ Count of times skipped for each job
$\mathcal{R}_1 \leftarrow$ Queue ordering policy
$\mathcal{R}_2 \leftarrow$ Backfill ordering policy

```
1  sort Q according to R_1
2  for job j ∈ Q do
3     if j can be started currently then
4        pop j from Q
5        Start(j, Q, M, S, SkipTable)
6     else
7        Reserve j at earliest possible time
8        L ← Q \ {j}
9        sort L according to R_2
10       for job j' ∈ L do
11          if j' can be started currently without delaying reservation of j then
12             pop j' from Q
13             Start(j', Q, M, S, SkipTable)
14          end if
15       end for
16       break
17    end if
18 end for
```

being launched or by a separate software system. Therefore, the proposed algorithm only needs to modify the $Start(\cdot)$ function as shown in Algorithm 2. This function takes care of putting jobs back on the queue when they are being delayed.

However, continually delaying jobs can lead to starvation. To prevent job starvation the modified schedule also includes a hard limit on the number of times a job can be skipped over. In our experiments we set this to 10, but the threshold was never met. This parameter could be extended to be per-job and used to enforce priorities or even ignore the scheduling delay entirely for certain jobs.

This leads us to the following design of the $Start(\cdot)$ function in Algorithm 2. It first checks

Algorithm 2 Modified $Start(\cdot)$ Function. This is called to launch jobs when resources are available. This modified version in RUSH puts jobs back on the queue if they will vary in performance significantly.

Input $j \leftarrow$ job
$Q \leftarrow$ scheduler queue
$M \leftarrow$ ML model
$S \leftarrow$ current machine state
SkipTable $\leftarrow$ Count of times skipped for each job

1 **if** SkipTable$[j] < j.skip_threshold$ **and**
$M(j, S) \in$ variation labels **then**
2 SkipTable$[j] \leftarrow$ SkipTable$[j] + 1$
3 push j after front of Q
4 **else**
5 launch job j
6 **end if**

if a job j is past its skip threshold (line 1). When j is past the threshold, then the **and** is short-circuited and j will be run (line 5). If j is within its skip threshold, then RUSH will evaluate the ML model $M(j, S)$ (line 1). Variation being predicted will lead to j being put back on the queue (lines 2-3). Otherwise, j will be run (line 5).

4.3.3 Implementation

In its entirety, RUSH requires recording large amounts of system data, training ML models, and modifying an existing batch scheduler implementation. This section discusses how these components of the pipeline were implemented.

4.3.4 Variability Predictor Implementation

To facilitate our scheduler, we utilize a data pipeline (Figure 4.2) that controls running the proxy jobs, collecting the performance and run time system data, and training the ML models. This pipeline needs to be portable and efficient, so that the same experiments and scheduler

adaptations can be used on other machines.

To make our pipeline portable we designed it entirely using bash to control job launches and Python to collect and analyze data. Jobs are launched using configurations from environment variables and can launch using either LSF or Slurm based job schedulers. Once the jobs run the data from these jobs are aggregated and analyzed using Python.

In addition to portability the pipeline needs to be efficient in its storage and analysis of large amounts of data. Collecting 32 HPCToolkit profiles per day can create several million files in a short amount of time. To alleviate this storage burden we only store database files from `hpcprof-mpi` in addition to the hatchet dataframes of them.

Given a set of application runs we collect a unified dataset depicted in Table 4.1. To build contained datasets we query the Sonar tables for aggregated LDMS data. We collect counter information for the duration prior to a job running. In our tests this was the five minutes prior to each proxy application's run. The counters were reduced over this interval with the minimum, maximum, and mean of each being included as a column in the dataset. Next the profiling information, including the wall clock time, is added into our table. This table is stored in a Pandas dataframe, which is pickled and compressed for easier use in the rest of the pipeline.

Prior to running experiments we train the ML models over the collected data sets and select the best one based on their F_1 score and accuracy. At the end of the pipeline the models are pickled and exported for use in the scheduler.

4.3.5 Job Scheduler Implementation

Using this exported model we modify the Flux [7] framework to implement our scheduler. Flux is a job scheduling software designed for HPC that integrates graph-based resource modeling with traditional batch scheduling. This section details how we implement our algorithm in Flux.

`RUSH` adds a scheduling policy within Flux to implement its algorithm. This is done by adding a new "scheduling policy" class to Flux. We extend the class `queue_policy_fcfs_t`, which in turn extends the general `queue_policy_base_t` class. Our implemented subclass `queue_policy_rush_t` implements the scheduling algorithm detailed in Section 4.3.2.

The `RUSH` implementation provides a modified function for ordering the queue. It first orders the queue with $\mathcal{R}_1$ as FCFS. When jobs are about to be run a Python script is first executed that runs the ML model with the next job as input.

This Python script then reads the collected counter data, runs the ML models, and provides its prediction to standard output. Our implementation uses this to make a scheduling determination as defined in Algorithm 2.

Jobs are matched to resources using Flux's default algorithm. Information about this mapping is captured implicitly in the system counters. Thus `RUSH` can be utilized with any resource mapping algorithm.

4.4 Experimental Setup

In this section, we describe the experiments and metrics used to evaluate the ML models and the new job scheduler.

Table 4.2: Description of experiments run in a system reservation of 512 nodes of Quartz to compare RUSH to the baseline.

Experiment	Name	Applications	# of Jobs	Description
ADAA	All Data All Applications	All	190	ML model trained on data from all running applications
ADPA	All Data Partial Applications	Laghos, LBANN, PENNANT	150	Subset of 3 applications running
PDPA	Partial Data Partial Applications	Laghos, LBANN, PENNANT	150	ML model trained on AMG, Kripke, sw4lite, SWFFT
WS	Weak Scaling	All	190	Jobs run on 8, 16, and 32 nodes; weak scaling
SS	Strong Scaling	All	190	Jobs run on 8, 16, and 32 nodes; strong scaling

4.4.1 Scheduling Experiments

To test the effectiveness of our scheduler we designed experiments to mimic typical workloads on an HPC system. We then compare the proposed scheduling policy on this workload with the default FCFS+backfilling scheduler as a control.

In order to create an HPC system-like environment we ran all of the experiments within a fixed set of 512 nodes on Quartz. These nodes lie in the same pod of the fat-tree cluster. The nodes are allocated by the system Slurm scheduler as a single job and we run Flux within this allocation to handle scheduling jobs in the experiments.

To mimic a typical HPC workload we design several experiments using the seven proxy applications listed in Sec 4.2.2. We setup a queue of jobs that takes between 30 and 50 minutes for all of them to run to completion. Each job runs on 16 nodes with 512 processes. At the beginning of the experiment we submit 20% of the jobs to the Flux queue immediately and submit the rest uniformly over 20 minutes. This mimics normal scheduling behavior where knowledge of every job to be scheduled is not known apriori.

Since we ran on a single pod on the fat-tree, we used a noise job that runs on 1/16th of the nodes in the experiment that continuously sends variable amounts of all-to-all traffic on the

network. This allowed us to run fewer experiments as we observed variation more frequently with the noise. To account for other system noise we run ten trials of each experiment: five with FCFS+EASY and five with `RUSH`.

We ran several different experiments to test how the scheduling policy performed under different circumstances. Table 4.2 highlights the experiments conducted within each 512-node reservation.

We first test the scheduling policy on all of the applications in "ADAA". This experiment runs all seven proxy applications and uses an ML model trained on a dataset containing runs from all seven applications.

To test the generalizability of the scheduler experiment "PDPA" only runs three applications and uses the ML model trained on the other four applications exclusively. We use Laghos, LBANN, and PENNANT as the applications to run and AMG, Kripke, sw4lite, and SWFFT to train the ML model. Experiment "ADPA" runs the same applications, but uses the full dataset for training. This serves as a control for "PDPA".

The final two experiments, "WS" and "SS", test how the policy generalizes to different scales of the jobs. Both run all of the applications and use an ML model trained on all of the data. However, they run each application on 8, 16, and 32 nodes. "WS" uses weak scaling to change the input parameters and "SS" strong scaling.

4.4.2 Metrics for Evaluating the ML Models

Before the scheduler is run in these experiments the ML models need to be trained and exported. This section discusses the metrics used to evaluate the success of the models in predicting

variation.

Performance variation is rare and, thus, the dataset is imbalanced. There are significantly more samples with little or no variation than there are samples with variation. This means testing accuracy is not a useful performance metric. A model that always predicts that no variation will occur would still yield an accuracy greater than 90%, but not provide any meaningful information to the scheduler. Due to this limitation, we use precision and recall related metrics to evaluate the success of our models. In particular, we use the F-measure (F_1 score) to compare and find the best performing model.

$$F_1 = \frac{\text{tp}}{\text{tp} + \frac{1}{2}(\text{fp} + \text{fn})}$$

where tp is the number of true positive predictions, fp the false positives, and fn the false negatives. F_1 score is a standard measure for how well models predict imbalanced labels.

When comparing different models, we use the average F_1 score from cross-validation. The F_1 score was calculated for the binary classification problem of variation vs no variation.

4.4.3 Metrics for Evaluating the Job Scheduler

We record different metrics that help us evaluate our new job scheduler across multiple different axes of improvement. Schedulers can provide improvement in several different areas, each of interest to different parties in the supercomputing eco-system. Providing reliability and high resource utilization is important to system administrators, while end-users may be more concerned with wait queue time and ease-of-use. Additionally, the efficiency of the scheduler is typically crucial to everyone.

Scheduler efficiency can be measured in terms of the *makespan*, which is defined as the

duration from the submission of the first job to the end of the last job. The makespan describes the amount of time it takes a scheduling policy to complete a workload on a certain system.

However, some policies with better makespans may see adverse performance in other areas. So we also record the *mean time in queue* as well as the *mean job variation per application*. The mean time in queue will show how delaying jobs impacts the average time spent waiting in the queue. The job variation will indicate to what degree `RUSH` successfully mitigates run time variation.

4.5 Results

We now present our results from training the machine learning models and the job scheduling experiments.

4.5.1 Prediction Accuracy of ML Models

Figure 4.3 presents the performance of different ML models we experimented with based on their F_1 scores. We see that given the system data in Table 4.1 and longitudinal run time data (see Section 4.2.2), the ML pipeline described in Section 4.3.1 is able to accurately predict run time variation.

The high F_1 scores show that the models can predict true labels or instances of variation well. While all of them perform well in this regard, the AdaBoost classifier outperforms the others. The results in the rest of the paper use Adaboost as the classifier.

The models are also insensitive to data exclusivity. We have two choices when aggregating data from the system. We can either aggregate over all the nodes on the system or only over

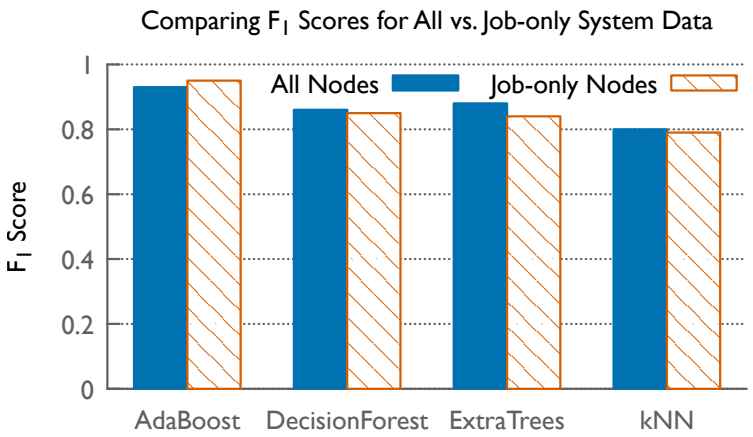


Figure 4.3: F_1 scores for different ML models. We see that the AdaBoost model has the highest F_1 scores. Additionally, we see that the models have comparable performance even without access to full system data.

those nodes that are allocated to the jobs in the dataset. When system data from only the job's nodes are used, we see comparable performance to training over all the nodes in the system. This is an important performance component as it allows the scheduler to only collect subsets of system data at a time (from the nodes a job is going to be scheduled on) when making scheduling decisions. This is a significant reduction in data processing that allows the scheduler to aggregate counters more frequently and efficiently.

4.5.2 Reduction in Application Performance Variability

The model accuracy results above confirm that we can use the trained model to advise the job scheduler regarding whether an incoming job will experience variation or not. Next, we discuss results from the experiments described in Section 4.4.1 and how `RUSH` helps in reducing performance variation.

Figures 4.5 and 4.4 show the number of runs that experienced variation in the first three experiments in Table 4.2. Averaged across the five repetitions of the ADAA experiment, FCFS+EASY

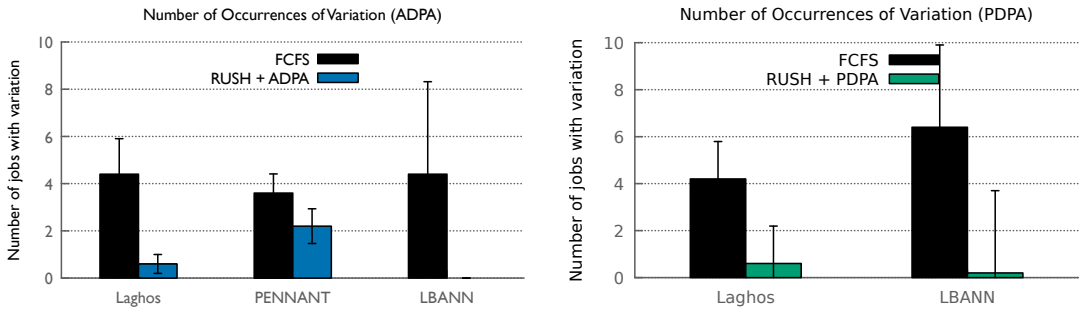


Figure 4.4: There is only a slight increase in the number of applications experiencing variation when using the ML model trained on data from all of the applications (left plot, ADPA) and separate applications (right plot, PDPA.)

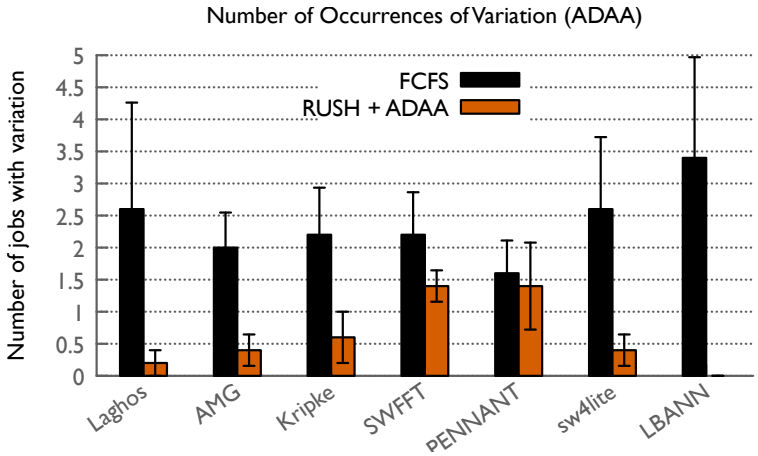


Figure 4.5: The number of runs that experience variation significantly reduces under the proposed scheduler (RUSH) for the ADAA experiment when compared to FCFS+EASY.

has between 1.5 and 3.5 runs on average per application with significant variation (see Figure 4.5). Using RUSH, this is reduced to between 0 and 1.5. The most variation prone applications, Laghos and LBANN, have almost no occurrences of significant variation when the RUSH scheduler is used. This shows the ability of the scheduler to reduce variation when its ML model has apriori knowledge of all the applications being run.

Experiments ADPA and PDPA show that the variation improvement also holds when the ML model has been trained on a subset of the running applications. This is shown in Figure 4.4 where we see similar improvement when the ML model is trained over the full dataset (left)

versus a partial dataset (right). Compared to ADAA, ADPA and PDPA show slightly higher amounts of variation in both FCFS+EASY and RUSH. This is due to the fact that Laghos and LBANN are the applications with the most variation and now more instances of them are running together than before. Pennant ends up having more runs with variation on average in PDPA than ADPA in the RUSH experiment. However, the increase is small in comparison to the decreases in LBANN and Laghos.

Since fewer runs suffer from variation using RUSH, we also expect the run times of each application to be more predictable. Figures 4.6, 4.7, and 4.8 present the run time distributions in each experiment. The run time distribution includes all of the runs of each experiment in Section 4.4.1 split by application. Figure 4.6 compares the run times of the proxy applications between FCFS+EASY and RUSH scheduling policies for the ADAA experiment. We observe that the maximum and mean run times reduce for the most sensitive applications, Laghos, LBANN, and sw4lite. The scheduling policy is able to successfully reduce variation in most instances. This is shown by the smaller ranges in run times and number of runs closer to the mean.

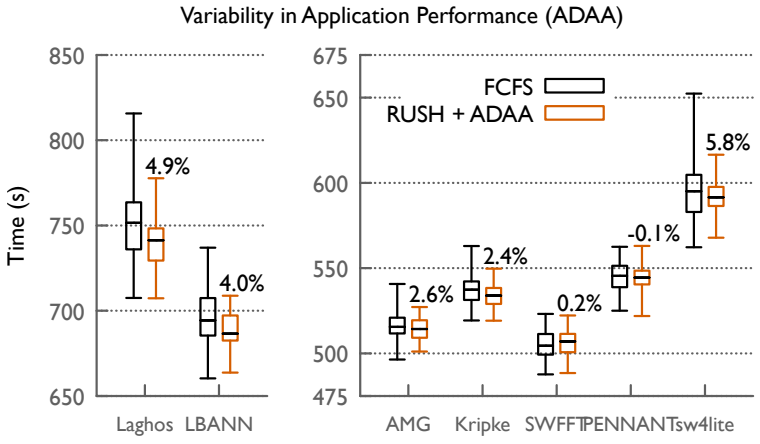


Figure 4.6: Distribution of execution times for each application in the ADAA experiment. RUSH reduces the maximum run time and the range of run times.

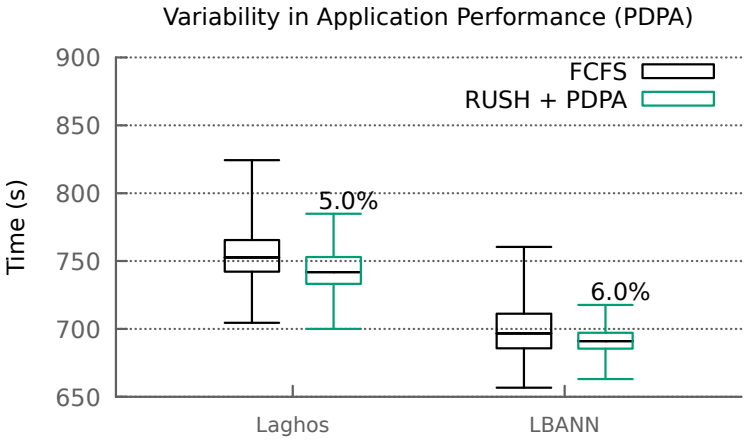


Figure 4.7: Distribution of execution times for each application in the PDPA experiment. The scheduler still performs well for applications where its ML model has never seen their data.

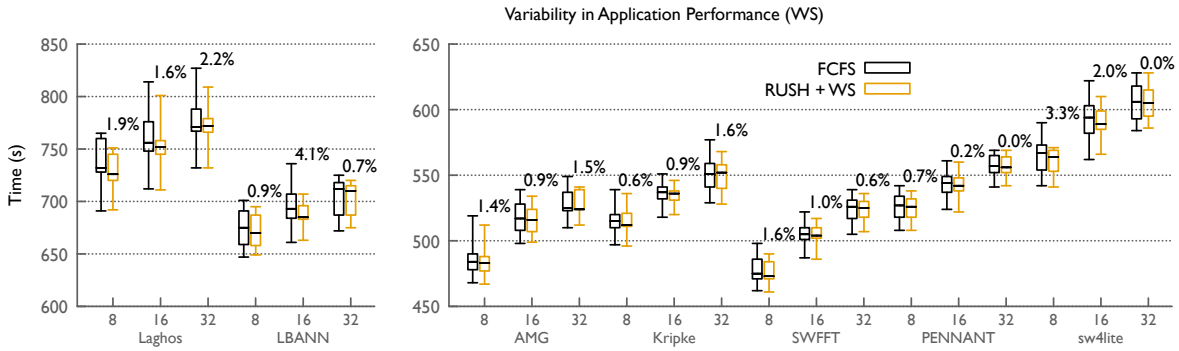


Figure 4.8: Distribution of execution times for each application in the Weak Scaling (WS) experiment.

From these results we also see an improvement in the maximum run time. This is likely the most important improvement from the perspective of an end-user as now they have a tighter upper limit on their application's running time. As before, Laghos, LBANN, and sw4lite, all experience the largest improvement in terms of maximum run time.

In Figure 4.7 we see that `RUSH` performs just as well when it has partial data versus full data. PDPA has similar improvements in maximum run time when compared with ADAA. In our experiments we find that ADPA, the control for PDPA, shows similar results to ADAA for LBANN, PENNANT, and Laghos. We can conclude that having access to historical runs for an

application prior to scheduling is not necessary to reduce its maximum run time. The generalizability of RUSH is important, since this data is typically not readily available.

Figures 4.8 and 4.9 present the experiments where the applications are run under weak and strong scaling respectively. In the WS experiment, RUSH reduces the spread of run times and the maximum run time more in the 8 and 16 node count runs. This is likely due to more communication in the 32 node runs and bias in the ML model from only training on 16 node runs.

Figure 4.9 shows the percent improvement in maximum run time when the applications are strong scaled. We see that the scheduler still provides improvement even as the amount of work per node decreases. For each application the maximum run time is reduced and sw4lite and LBANN show the greatest improvements. In experiments WS and SS, there were no applications with increase in the maximum run time. The run time distributions either stayed the same or, more often, reduced in range. This displays the ability of RUSH to extend to other node counts even under different types of scaling.

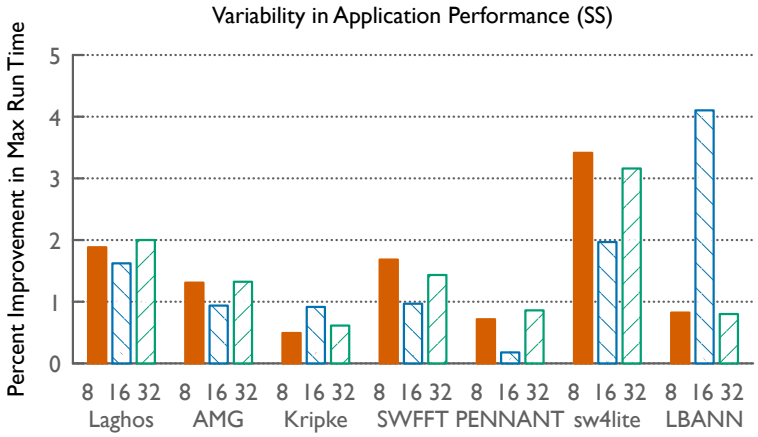


Figure 4.9: Percentage improvement in the maximum run time for each application in the Strong Scaling (SS) experiment when comparing RUSH with the baseline.

4.5.3 Scheduler Evaluation

In addition to mitigating variation for individual users, we also want to ensure that the scheduler does not impact system throughput negatively. We start with comparing the makespan for the two scheduling policies in Figure 4.10. For each experiment, the makespan is improved by between 18 and 66 seconds. The variation in each application has been reduced without burdening the makespan significantly and in some cases improving it. By reducing the expected run time of some of the applications, RUSH reduces the duration of some of its jobs. In cases where a significant amount of variation is prevented, the scheduler will have a lower makespan.

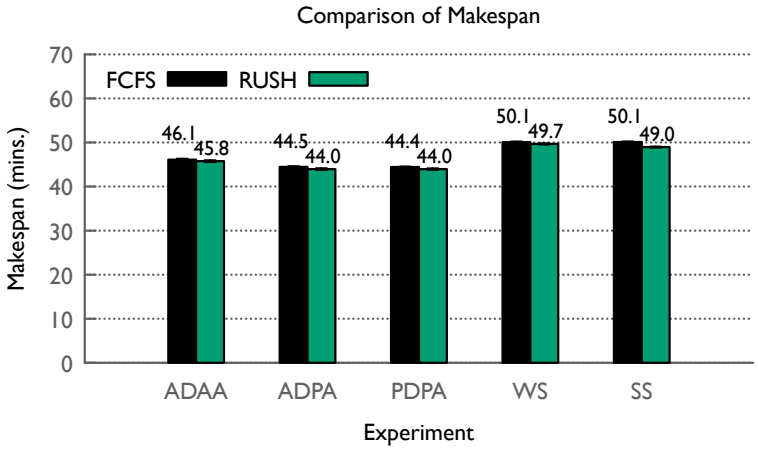


Figure 4.10: Scheduler makespans. For each experiment this figure displays FCFS+EASY and RUSH's makespans averaged over their five trials. RUSH outperforms FCFS+EASY in each experiment.

Figure 4.11 shows the differences in wait times for each application for the ADAA experiment. This plot only includes wait times for the 80% of applications that were not placed in the queue at the start of the experiment. In the case of RUSH, the wait times are spread out and show both favorable and worse performance compared to the FCFS+EASY scheduler. The average wait time went up for variation intensive applications such as Laghos, sw4lite, and LBANN. This

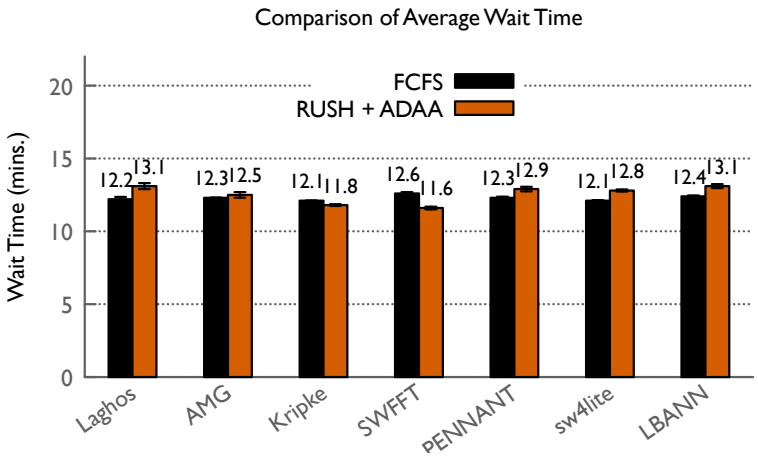


Figure 4.11: Average wait time per application in experiment ADAA. RUSH has a larger range of wait times and is often higher.

is due to them being pushed back in the queue more often than others. Both Kripke and AMG got through the queue faster on average in the RUSH scheduler. Though the wait times vary, they are always within a minute. This less than a percent increase in wait time is insignificant, especially compared to the reduction in variation that can be obtained at its cost.

Chapter 5: Predicting Cross-Architecture Performance of Parallel Programs

This chapter describes work in cross-platform performance modeling that develops an approach for using machine learning to predict the performance of an application on several different architectures using performance counters. The performance prediction model is then used in a novel batch scheduling algorithm to schedule work across a heterogeneous cluster efficiently. It is demonstrated that this approach can reduce the overall throughput of a multi-cluster setup by placing jobs on the most efficient resource. The contributions in this chapter are published in [104].

5.1 Motivation

An increasing number of scientific workloads are being expressed as workflows with sets of computational tasks and dependencies between them [67, 8]. These workflows typically involve ensembles of tasks (jobs) in a pipeline that run different codes such as simulations, uncertainty quantification analysis, and machine learning training. As applications become more portable due to the emergence of portable programming models [64], package managers [47], and containerization techniques, different tasks or jobs might be better suited for different hardware architectures. Given these portable workflows and the increasingly heterogeneous set of computing resources available to end users today, it is important to develop capabilities to efficiently place

these tasks on the most efficient resources available.

Different tasks or applications in a workflow can be assigned to different architectures if users have access to a variety of compute nodes via a multi-resource job scheduler, which is becoming increasingly common, both in data centers and HPC facilities. As a result, the demand for such multi-resource schedulers [7] is emerging. In an ideal setting, scheduler can automatically decide the most suitable architecture for different jobs in terms of performance. This can remove the user from the decision making process and let a system scheduler decide what hardware to run an application on. However, in practice, this requires being able to predict the performance of incoming jobs across diverse architectures. This is a complex problem that would involve developing models for understanding the performance of scientific applications across diverse architectures.

Cross-architecture performance modeling is a challenging problem because application execution times are dependent on several factors with non-trivial relationships to performance. The performance depends on how well the application's behavior aligns with the properties of the hardware it is running on. These hardware properties, such as peak flop/s, memory bandwidth, and cache sizes are easy to obtain, however, the behavior of the application is non-trivial to model. Application performance can depend on a number of characteristics such as arithmetic intensity, memory loads/stores, branching behavior, I/O, and many more. Characterizing these and using them to model performance on a diverse set of architectures is challenging due to the number of contributing factors and complexity of the relationship.

In this chapter, we propose a solution to the cross-architecture performance modeling task by training a machine learning model to predict the relative performance of an application across a set of architectures given performance counters of the application from one architecture. In order

to accomplish this, we collect a data set of application runs from four different HPC systems with different architectures and measure a hand selected set of performance counters. These counters, along with the recorded execution times, are used to train a regression model to predict relative performance vectors. Additionally, we demonstrate the generalizability of our model by evaluating it on a set of applications it has not seen before. To our knowledge, this is the first model to be able to predict performance across multiple architectures at the same time that works on entire applications. Finally, we demonstrate the makespan improvement from using this model in a multi-resource scheduling simulation.

5.2 Overview of Methodology

We first provide an overview of our methodology to predict the relative performance of an application across a set of architectures given performance counters of the application from one architecture. This includes two things – the data collection phase and the model training phase (Figure 5.1). In the first phase, we collect performance profiles for a variety of applications running on N different HPC systems with different architectures and record a hand-selected set of performance counters. These counters, along with the recorded execution times, are used to train a regression model to predict relative performance in the second phase.

Since our goal is to predict performance on other architectures relative to a baseline on one architecture, we introduce the term *Relative Performance Vector* (RPV) that encodes the relative performance of an application across several architectures. To define RPV, let us consider a set of applications A, corresponding input problems I_A, and systems S. For a particular application and input problem pair $(a, i) \in A \times I_A$ executed on N systems in S we can define the *Relative*

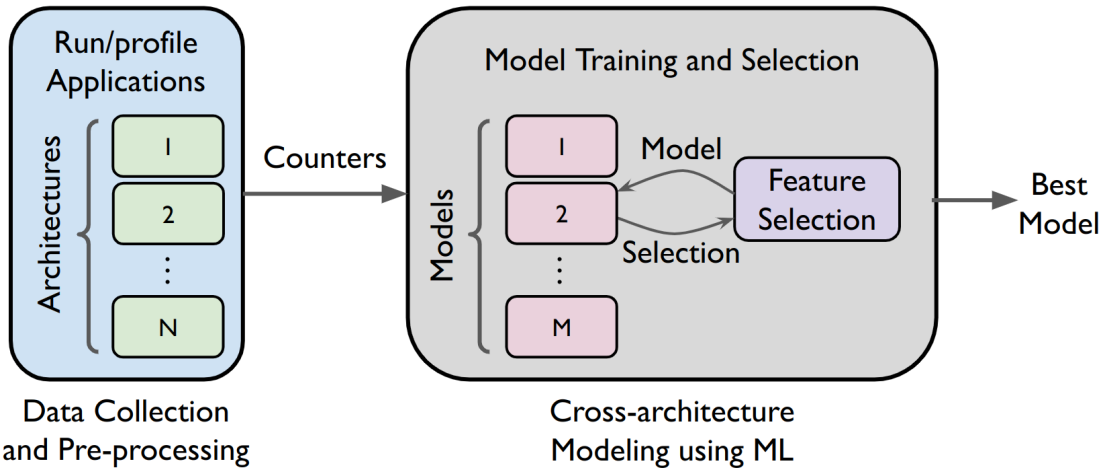
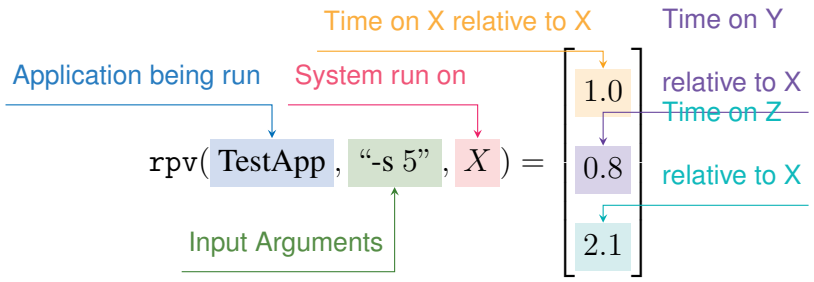


Figure 5.1: Overview of data and machine learning pipeline. Applications are profiled on several architectures and performance counters are collected for training the model. Model and feature selection are done iteratively until the best set is selected.

Performance Vector as $\texttt{rpv} : (A, I_A) \times S \mapsto \mathbb{R}^N$ such that $\texttt{rpv}(a, i, s)$ is the vector of the performance of (a, i) across all platforms relative to that on system s. Here we assume that (a, i) can run on all the systems in S. For example, consider running an application-input pair (TestApp, "-s 5") on systems X, Y, and Z. If the application runs in ten minutes on system X, eight minutes on system Y, and 21 minutes on system Z, then the performance vector relative to X would be:



We also define $\texttt{rpv}(\cdot, \cdot, \min)$ and $\texttt{rpv}(\cdot, \cdot, \max)$ as the performance vectors relative to the systems where lowest and highest performance is obtained, respectively. The rpv provides a concise, mathematical representation for relative performance across systems that can be used in

our further downstream modeling tasks.

In order to model the mapping $\texttt{rpv} : (A, I_A) \times S \mapsto \mathbb{R}^N$, we need a large number of input and output data to train on. This requires a large number of samples in the $(A, I_A) \times S$ space. To collect these, we profile a variety of aplications at several of their inputs on several architectures. These runs provide hardware counters that may provide insight into an application's behavior for many application, input, and architecture tuples.

We use the counters collected during profiling as the data set for the machine learning (ML) component (second phase). The ML component uses the profiled counters from a particular architecture to predict the relative performance vector across a set of systems. We try different ML models and feature sets to identify the best performing model. This model is exported and used in downstream relative performance prediction tasks such as cross-architecture scheduling.

5.3 Data Collection and Pre-processing

In this section, we provide details of how we generated the dataset used for our modeling problem. We describe the process of running and profiling the applications, and collecting the performance metrics.

5.3.1 Scientific Applications

In order to model the relative performance of applications run on an HPC machine, we need to collect performance data from applications that are typically run on these machines. We accomplish this by running a set of applications, benchmarks, and proxy applications from the ECP Proxy Applications Suite [44] and E4S Test Suite [133]. These are chosen because they are

Table 5.1: Overview of the four architectures we collect performance data on. There are two CPU only systems and two CPU+GPU systems. The CPUs span three vendors: Intel, IBM, and AMD, while the GPUs originate from two: NVIDIA and AMD.

System	**CPU Type**	**CPU** *cores/node*	**CPU** *Clock Rate (GHz)*	**GPU Type**	**GPUs/node**
Quartz	Intel Xeon E5-2695 v4	36	2.1	—	—
Ruby	Intel Xeon CLX-8276	56	2.2	—	—
Lassen	IBM Power9	44	3.5	NVIDIA V100	4
Corona	AMD Rome	48	2.8	AMD MI50	8

designed to be representative of actual workloads on HPC systems, but are simpler to build and run than full scientific applications.

Table 5.2 lists the applications used in our data set. There are 20 applications in total, and eleven of them have GPU support. The GPU support comes from a variety of libraries such as OpenMP, Kokkos [135], RAJA [64], and native CUDA or HIP. Each application is paired with different input configurations when run, in order to test different problems and problem sizes. We build and install all of the applications with their default build settings in their respective Spack [47] packages.

5.3.2 Architecture Descriptions

We run each application-input pair on four different machines with different architectures. These are listed in Table 5.1. There are two Intel Xeon based, CPU-only machines and two GPU-based machines. The first GPU machine uses IBM Power9 CPUs and NVIDIA V100 GPUs, while the second uses AMD Rome CPUs and AMD MI50 GPUs.

On each of these systems the applications are run in three configurations – on one core, on

Table 5.2: The applications used in our study listed alongside a brief description of what each application does and whether it supports running on a GPU.

Application	Description	GPU
AMG	Algebraic multigrid solver	✓
CANDLE	Deep learning models for cancer studies	✓
CoMD	Molecular dynamics and materials science algorithms	
CosmoFlow	3D convolutional neural network for astrological studies	✓
CRADL	Multiphysics and ALE hydrodynamics	✓
Ember	Communication patterns	
ExaMiniMD	Molecular dynamics simulations	✓
Laghos	FEM for compressible gas dynamics	✓
miniFE	Unstructured implicit FEM codes	✓
miniGAN	Generative Adversarial Neural Network training	✓
miniQMC	Real space quantum Monte Carlo algorithms	✓
miniTri	Triangle based data analytics algorithms	
miniVite	Graph community detection	
DeepCam	Climate segmentation benchmark	✓
Nekbone	High-order, incompressible Navier-Stokes solver	
PICSARlite	Particle-in-Cell simulation	
SW4lite	Seismic wave simulation	✓
SWFFT	Distributed-memory parallel 3D FFT	
Thornado-mini	Radiative transfer solver in multi-group, two-moment estimations	
XSBench	Monte Carlo neutronics simulations	

one node using all the cores, and on two nodes. The one-core runs use one GPU if applicable. MPI is used for the one and two node runs to make use of all the cores and GPUs on the node. Some applications only support run configurations with square or power of two MPI processes and are, thus, run on the nearest number of ranks possible to one or two nodes. If an application

does not support running on a GPU, we run it on the CPU only and use comparable CPU counters. If an application does support running on a GPU, then only GPU counters are collected. During these runs, HPCToolkit [3] (with CUPTI [39] on NVIDIA GPUs or rocProfiler [120] on AMD) is used to record the application counters, and after the application run is complete, Hatchet [21] is used to parse these counters from the HPCToolkit output. For multi-process and multi-GPU runs, we record the mean value of the counters across all processes. The final results from all runs are then collected into a Pandas dataframe for use in the later tasks.

5.3.3 Details of Recorded Hardware Counters

To understand the varied computational characteristics of different applications in Table 5.2, we record several hardware counters during the application runs. Table 5.3 lists the counters recorded on each architecture in our data set. Counter names are not consistent across different architectures and they may also represent slightly different data. However, we have tried to identify similar counters that model the same underlying performance characteristics that affect final performance. Most of these counters fit into one of three categories: control flow, data intensity, or I/O. These categories capture the main performance characteristics of applications across different architectures. Broadly speaking, applications with more complex control flow will fair better on CPUs, which are geared towards latency. On the other hand, applications with more data intensity generally benefit on throughput-geared GPUs.

Table 5.3: Final features in the collected data set and the counters/values they are derived from. We combine derived values from the recorded counters and meta-data about the run configuration.

Feature	**Description**	**Source Counters & Values**			
		Quartz	Ruby	Lassen	Corona
Branch Intensity	Ratio of branch instructions to total instructions	PAPI_BR_INS	PAPI_BR_INS	cf_executed	–
Store Intensity	Ratio of store instructions to total instructions	PAPI_SR_INS	PAPI_SR_INS	inst_executed_local_stores, inst_executed_global_stores	LDSInsts, GDSInsts
Load Intensity	Ratio of load instructions to total instructions	PAPI_LD_INS	PAPI_LD_INS	inst_executed_local_loads, inst_executed_global_loads	LDSInsts, GDSInsts
Single FP Intensity	Ratio of single precision FP instructions to total instructions	PAPI_SP_OPS	PAPI_SP_OPS	flop_count_dp	VALUInsts, SALUInsts
Double FP Intensity	Ratio of double precision FP instructions to total instructions	PAPI_DP_OPS	PAPI_DP_OPS	flop_count_sp	VALUInsts, SALUInsts
Arithmetic Intensity	Ratio of integer arithmetic instructions to total instructions	bdw_ep::ARITH	clx::ARITH	inst_integer	–
L1 Load Misses	L1 cache load misses	PAPI_L1_LDM	PAPI_L1_LDM	local_load_requests, local_hit_rate	–
L1 Store Misses	L1 cache store misses	PAPI_L1_STM	PAPI_L1_STM	local_store_requests, local_hit_rate	–
L2 Load Misses	L2 cache load misses	PAPI_L2_LDM	PAPI_L2_LDM	gld_efficiency	TCC_MISS_sum, TCC_EA_RDREQ
L2 Store Misses	L2 cache store misses	PAPI_L2_STM	PAPI_L2_STM	gst_efficiency	TCC_MISS_sum, TCC_EA_WRREQ
IO Bytes Written	Bytes written to IO	IO	IO	IO	IO
IO Bytes Read	Bytes read from IO	IO	IO	IO	IO
Extended Page Table	Extended page table size	EPT	EPT	EPT	EPT
Memory Stalls	Memory stalls	PAPI_MEM_SCY	PAPI_MEM_SCY	GINST:STL_ANY	MemUnitStalled
Nodes	Nodes	Run Configuration	Run Configuration	Run Configuration	Run Configuration
Cores	Cores	Run Configuration	Run Configuration	Run Configuration	Run Configuration
Uses GPU	1 if counters from GPU; 0 otherwise	0	0	1 if app uses GPU	1 if app uses GPU
Architecture	one-hot-encoded vector for what architecture these counters were recorded on	(1 0 0 0)	(0 1 0 0)	(0 0 1 0)	(0 0 0 1)

5.3.4 Preparing the Final Dataset

Using the counters listed on the right of Table 5.3 we compute a set of derived values as the final features in the data set. These features are detailed on the left of Table 5.3. The instruction related counters branch, store, load, single FP, double FP, and integer arithmetic are all computed to be ratios of the total number of instructions. This normalizes the values across runs, which may have drastically different numbers of total instructions. The remaining eight features are normalized by subtracting that feature's mean to center its values and dividing them by its standard deviation. We additionally include whether the run was from a GPU or not, how

many nodes, and how many cores the run used. The architecture feature is a one-hot-encoded vector encoding what architecture the counters were collected on. In the context of this paper, that is four separate features that are used to denote whether the run is from Quartz, Ruby, Corona, or Lassen.

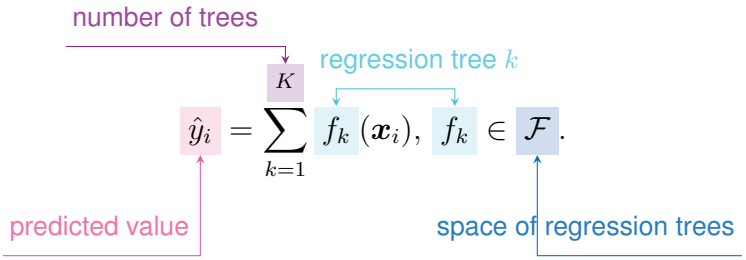
The final data set has 21 columns and 11,312 rows. Each row represents a run of an application-input pair for a specific number of MPI processes on a single architecture. The columns are derived from the counters collected during the run and meta-data about the run (see Table 5.3).

5.4 Modeling with Machine Learning

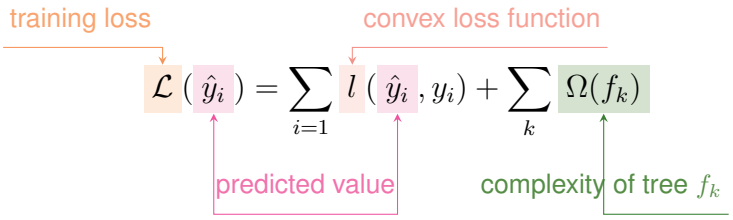
In this section we present our methodology for training the machine learning models, finding the best features/models, and evaluating their performance.

5.4.1 Training

Now that we have a data set of counters from applications and the corresponding relative performance vectors across a set of architectures, we want to use machine learning to predict the relative performance vectors given counters from one of the architectures. In order to learn how to predict relative performance vectors we use the XGBoost (eXtreme Gradient Boosting) regression model [33]. This model is an ensemble of decision trees that are additively combined to make final predictions. If $\hat{y}_i \in \mathbb{R}$ is the predicted regression value of the model, then it can be computed as

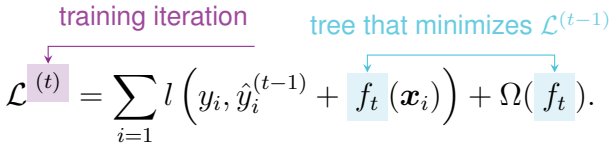


To minimize over-fitting we can add a regularized objective function that models the complexity of the trees.



(5.1)

Since this is parameterized by functions ($f_k \in \mathcal{F}$) it cannot be optimized using typical optimization methods. Thus, *gradient tree boosting* greedily adds in the best functions throughout training iterations by selecting the f_t that minimizes Equation 5.1 the most. These f_t can be additively combined into a new loss function as



This can be optimized using 2nd-order approximizations and standard convex minimization methods. XGBoost implements this gradient tree boosting method alongside a number of state-of-the-art techniques for tree splitting and pruning. Additionally, it provides efficient implementations that can scale to large numbers of data samples and run on GPUs. It is a state-of-the-art machine learning algorithm for learning on tabular data.

In order to train an XGBoost regressor we use its publicly available Python library at version 1.7.1. We train the model on a CPU on the Ruby system. Training the XGBoost model takes on the order of tens of seconds on average. The model takes the features from Section 5.3.4 and predicts the relative performance across all four architectures as a vector. Mean absolute error (MAE) is used as the minimization objective during training. During this training 10% of the data is set aside as a testing data set, while the other 90% is shown to the model as a training data set. While training on the training data set, the data is further split into 5 folds as part of k-fold cross-validation. The model is trained on 4 out of the 5 folds at a time, while the other is used as validation. This is done for all 5 combinations and the average MAE is reported.

We additionally train several other common machine learning regressors to compare the quality of the XGBoost model to other state-of-the-art methods. For this we include linear regression and decision forests. These are implemented from the scikit-learn Python library. As with XGBoost these are trained with a 90-10 train-test split and 5-fold cross-validation. We also test against *mean* prediction as a baseline for the ML models. This regressor guesses the mean rpv in the training set for all samples in the test set.

5.4.2 Model and Feature Selection

To select the best model and feature set we first train all the models on all the features. After training we select the best set of features using those reported by XGBoost and the decision forest, since these models expose feature importances. These features are then used to re-train all the models again.

In order to measure the feature importances of the trained model we use XGBoost to easily

recover importance values. XGBoost, in its Python framework, computes feature importances during training and exposes them in its model interface. It calculates them based on the average gain across all decision splits in the trees. During training a tree will add splits on a feature to improve its predictive performance. The improvement in performance from this split is called the gain. When there are multiple regression targets the gain is averaged over each output.

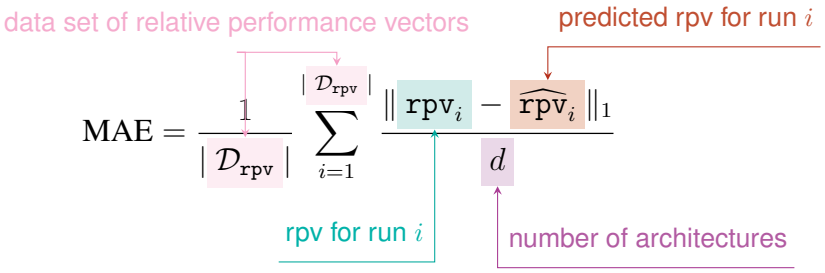
For any given feature if we average the gain from all the splits on that particular feature in a tree, then we can compute the importance of the feature for that tree. This includes all the splits in XGBoost's sets of trees. Finally, we can compute this value for all of the features in the data set to retrieve a feature importance vector.

With this method of calculating feature importances we can expose the relative contribution of each feature to the model's predictions. A higher importance indicates that that feature contributes more to the models performance than other lower scored features. Decision tree feature importances can also be calculated based on the frequency and coverage of splits for a feature, however, these can be biased towards features with a large number of unique values and numeric features. Both of these are present in our data set, so we elect to use the average gain.

Since the data set has a relatively small number of features, the feature selection will likely have negligible impact on model training time. However, discovering the most impactful features gives insight into what is most necessary in predicting cross-architecture performance. Additionally, it allows us to collect less features in future implementations of this methodology. This is a considerable optimization as data collection is the most time and resource intensive portion of our machine learning pipeline.

5.4.3 Evaluation Metrics

We evaluate the model's performance by two different metrics: *Mean Absolute Error* (MAE) and *Same Order Score* (SOS). The MAE encodes the average magnitude of error in the relative performance predictions. This measure provides a value that is easy to reason about regarding predictive performance. An MAE of 0.1 means that the model predicts the relative performance of applications within ± 0.1 on average across each vector.



The SOS score denotes the number of samples where the model predicts the relative performance vector in the correct order. We define two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ as being in the same order if the i-th elements $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$ are both the n-th largest in their respective vector, for all i. The SOS is then defined as the fraction of predicted relative performance vectors that are in the same order as their respective true relative performance vector. This metric shows how well the model understands the ordering of performance on different architectures, but ignores the magnitude of its predictions. Thus, the SOS combined with MAE gives reasonable insight into how well the model is predicting relative performance vectors. Both of these metrics are computed over the testing set for data samples that the model has not seen before.

5.5 Scheduling Experiment

Once a model is trained to predict relative performance vectors it can be used to make informed cross platform scheduling decisions. We test this capability in our trained model by simulating a multi-resource scheduling environment. We create a workload of 50,000 jobs randomly sampled from our existing data set with replacement. These are scheduled using the First-Come-First-Serve with EASY backfilling scheduling algorithm (FCFS+EASY) [81] presented in Algorithm 3. This algorithm uses the `Machine` function to map a job to a machine: Quartz, Ruby, Lassen, or Corona. If the machine cannot satisfy the resource requirement of a job (the number of nodes it needs), then the job is reserved at the earliest possible time or backfilled. Otherwise, it is run immediately, and the function $\texttt{Start}(j, m)$ represents running job j on machine m. We use the observed run times on each machine from the data set to determine how long the job would run for simulation purposes.

We run this scheduling simulation with four different machine placement functions: Round-Robin, Random, User+RR, and Model-based. These functions expose the common interface for scheduling, $\texttt{Machine}(j, i, M)$, where j is the job to schedule, i is the index of j in the queue, and M is the set of machines considered for multi-resource scheduling. Depending on the algorithm some of these arguments are not used. The Round-Robin placement places jobs on machines in a round robin fashion rotating between machines for each consecutive job. The Random placement uniformly selects a random machine of the four to run on. The User+RR placement mimics traditional user behavior by running on GPU systems for GPU enabled apps and CPU only systems otherwise. Round robin is used to decide which GPU system to use for GPU enabled apps and likewise for CPU-only apps. Finally, the Model-based placement, Algorithm 4, uses an ML-

Algorithm 3 Multi-Resource Scheduling Algorithm using FCFS+EASY. This standard algorithm queues jobs using policy $\mathcal{R}_1$ and uses EASY to backfill smaller jobs. The function `Machine` is used to map jobs to resources. The symbol $\setminus$ represents the set minus operation.

Input $Q \leftarrow$ queue of jobs
$\mathcal{R}_1 \leftarrow$ Queue ordering policy
$\mathcal{R}_2 \leftarrow$ Backfill ordering policy
$M \leftarrow$ Set of machines used for multi-resource scheduling
$\texttt{Machine}(j, i, M) \leftarrow$ Function that maps jobs to machines

$i \leftarrow 0$
sort Q according to $\mathcal{R}_1$
for job $j \in Q$ **do**
  if j can start now **then**
    pop j from Q
    $Start(j, \texttt{Machine}(j, i, M))$
    $i \leftarrow i + 1$
  else
    Reserve j at earliest possible time
    $L \leftarrow Q \setminus \{j\}$
    sort L according to $\mathcal{R}_2$
    for job $j' \in L$ **do**
      if j' can start now without delaying j **then**
        pop j' from L and Q
        $Start(j', \texttt{Machine}(j', i, M))$
        $i \leftarrow i + 1$
      end if
    end for
  end if
end for

based model to pick the fastest machine for each job and run it there. If the machine cannot satisfy the resource requirement of the job, then it picks the next fastest and so on.

We implement this scheduling simulation in Python using our data set to get run time information for jobs. The nodes available on each machine reflect the number available on the actual machines. This is not meant to substitute rigorous scheduling simulation studies but only to demonstrate a potential use case.

Algorithm 4 Performance-aware machine placement for scheduled jobs using the machine learning model to predict relative performance.

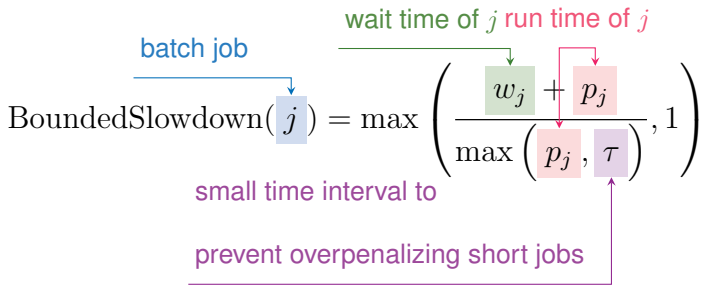
Input $j \leftarrow$ Job to schedule
$i \leftarrow$ Index of j in queue
$M \leftarrow$ Set of machines used for multi-resource scheduling

function $\texttt{Machine}_{\texttt{Greedy}} j, i, M$
$rpv \leftarrow \texttt{Model}(j)$
$m \leftarrow \text{argmax}_{i \in M}\, rpv$
if all $i \in M$ are full **then**
  return m
else
  $M' \leftarrow M$
  while m is full **do**
    $M' \leftarrow M' \setminus \{m\}$
    $m \leftarrow \text{argmax}_{i \in M'}\, rpv$
  end while
end if
return m
end function

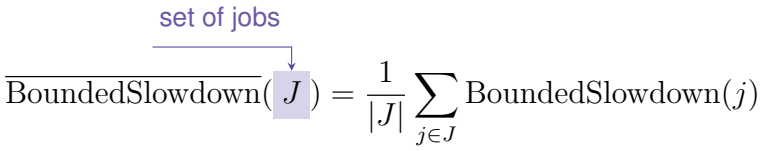
5.5.1 Evaluation Metrics

When evaluating the efficiency of our scheduling algorithm we are concerned with performance from the perspective of individual jobs as well as the scheduler as a whole. Users will hope to see a faster turnaround time from job submission to completion for their jobs, while system administrators may look at the job throughput of a given scheduler to measure its performance. To quantify both of these we use *average bounded slowdown* and *makespan*.

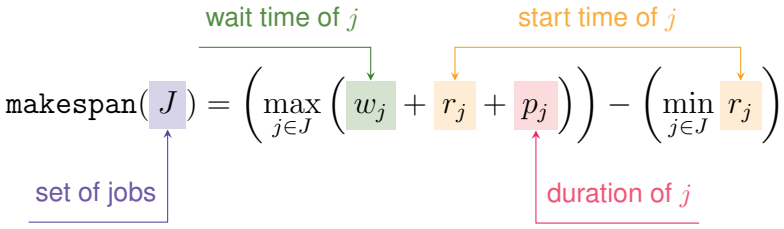
The average bounded slowdown represents the average slowdown of a set of jobs with a fixed bound to prevent overpenalizing very short jobs. Slowdown is the ratio of submission-to-completion time with a wait time versus without a wait time. This provides a per-job evaluation metric to see how much each job is affected by the scheduling algorithm. The bounded slowdown can be calculated as shown below.



$\tau = 10$ in our evaluation. Using the function below we can compute the average bounded slowdown over a set of jobs J.



Using the same set of jobs J we can also define the makespan as the time from the first job submission to the time when the last job finishes. This measures the amount of time it takes for a scheduler to complete a set of work and is commonly used to compared different scheduling algorithms over fixed workloads.



Both of these metrics are computed for each scheduling algorithm across our workload. We compare them between all the scheduling algorithms to observe the benefit from cross architecture performance modeling. For each metric a lower value indicates better performance.

5.6 Results

In this section we present the results from our training and scheduling experiments.

5.6.1 Evaluation of ML Models

Figures 5.2 and 5.3 show the mean absolute error and same-order-score of each model on the testing data set. We see that XGBoost performs the best for both of these metrics.

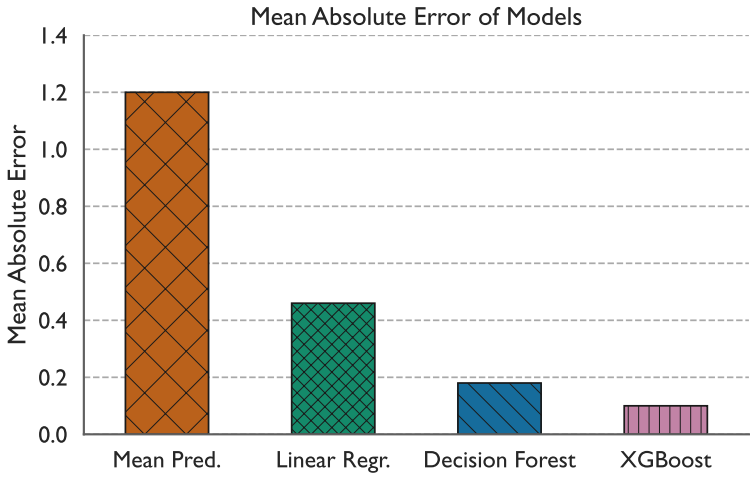


Figure 5.2: The MAE of each machine learning model over the testing data set after training. XGBoost outperforms the other models with an MAE of 0.11. Lower MAE is better.

The XGBoost model scores a MAE of 0.11 (see Figure 5.2). This means that the model can take counters recorded on one architecture and predict its relative performance to the others within 0.11 on average. This is a 81.6% improvement over guessing the mean relative performance vector from the data. From this we can infer that the model is not simply guessing according to the distribution of the runtime data, but is rather correlating counter data with its performance prediction.

The linear and decision forest models perform better than guessing the mean, but do not exceed the MAE of XGBoost. The decision forest scores the closest to XGBoost likely since they are both ensembles of decision trees. However, XGBoost implements boosting alongside a number of other pruning techniques that strengthen its prediction.

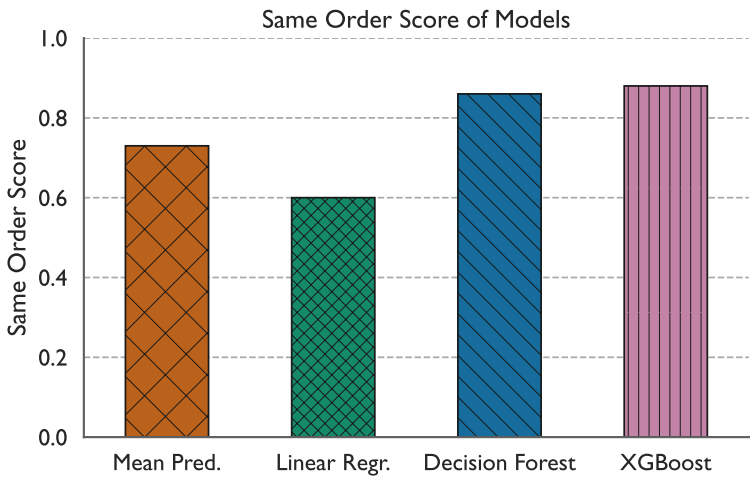


Figure 5.3: The SOS of each machine learning model over the testing data set after training. Higher SOS is better.

We see similar performance from XGBoost on the SOS metric where it is the best model (see Figure 5.3). It is able to predict the relative performance vector in the correct architecture order in 76% of samples in the testing set. This means it is frequently able to predict the fastest and slowest architectures for a particular application and input, which is a valuable result to a user who is likely trying to avoid the slowest architecture and run on the fastest. Additionally, if the system with the fastest architecture is busy, then the user can select the next fastest and so on.

As with the MAE metric the decision forest has similar, but lower performance to XGBoost. The linear model is next as with MAE. It scores slightly higher than the SOS from guessing only the mean relative performance vector.

5.6.2 Ablation Study

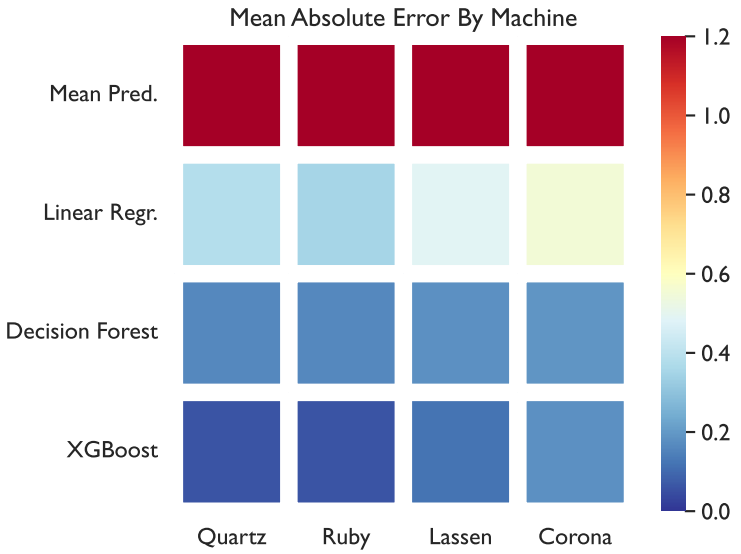


Figure 5.4: The MAE of each model when predicting using profiles from one particular machine. For instance, the bottom right of the plot represents the MAE when predicting relative performance vectors with XGBoost and profiles from Ruby.

Here we study the effects on modeling performance when removing certain features and/or data from the training set. Figures 5.4 and 5.5 further detail how well the models perform when given counters from each individual architecture. In both of these figures the ”mean” prediction row is constant, since the mean relative performance vector is independent of the input features. The first, Figure 5.4, shows the MAE scores for each ML model. We observe the same trends as Figure 5.2 where XGBoost has the best MAE. However, we notice that profiles from Ruby lead to a lower MAE and, thus, better predicted relative performance vectors. In fact, profiles from the two CPU systems, Ruby and Quartz, generally leader to better MAE. This same trend continues for the SOS metric.

The fact that counters recorded on CPU machines lead to better predictions on average is

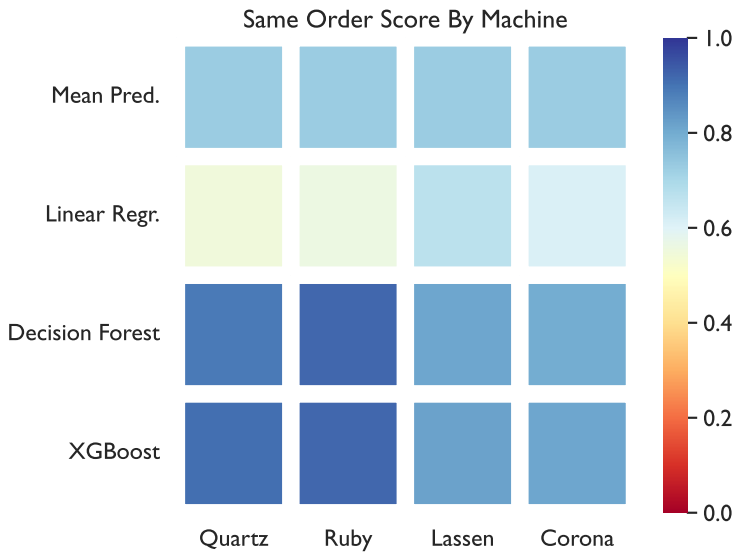


Figure 5.5: The SOS of each model when predicting using profiles from one particular machine.

an important observation for using this model in practice. CPU machines are generally cheaper and more readily available. Users can run their code on them and get predictions from the model for less available resources, such as GPUs. Additionally, users can obtain an estimate of the speedup from running on a given architecture without actually being capable of running on that architecture. For instance, if a particular application does not support AMD GPUs a user could estimate the performance increase/decrease if they were to implement AMD GPU support.

We hypothesize that the CPU performance metrics give better predictions due to the maturity of CPU performance counters and the profiling tools used to record them. CPU performance counters have been used extensively and the difficulties in recording them accurately have been well studied. On the other hand, GPU profiling, particularly for AMD, is a relatively new feature in HPCToolkit and the counters may not be as reliable as those recorded on a CPU.

Figure 5.6 shows the performance of XGBoost when trained on data from two of the three resources amounts and evaluated on the third. We observe that predicting the one node relative

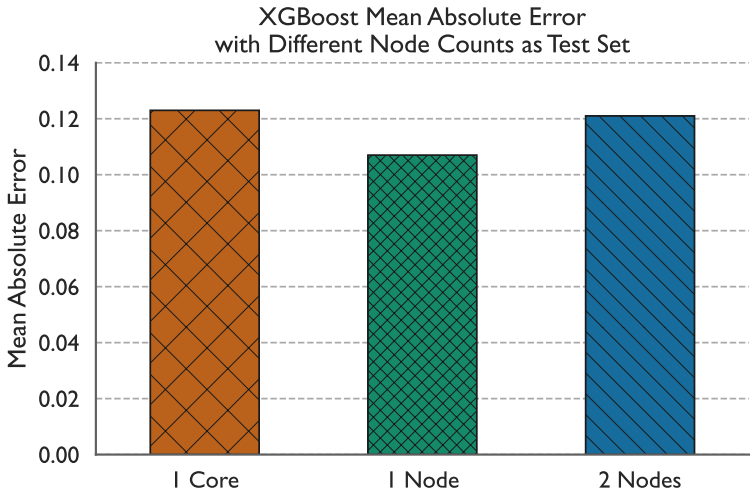


Figure 5.6: Evaluation MAE of XGBoost when each resource count is removed from the training set and used for evaluation. The model performs best at predicting 1 node performance when trained on 1 core and 2 node date. Note that all scores are lower and still very strong.

performance vectors gives the best MAE. It is unclear whether this is because modeling the one node performance is easier or that the one core and two node data is more representative. Regardless, all three node counts score very close to 0.11 MAE, which is still a strong result.

Additionally, we can see the performance of XGBoost when trained on all but one application and evaluated on the remaining applications in Figure 5.7. Again, we see that the model performs well across all applications. However, it does notably perform worse for the ML and Python-based applications. This is possibly due to more noise and/or complicated software stacks involved in running each of these applications. These applications also tend to depend on more libraries and have more dependencies than the other applications.

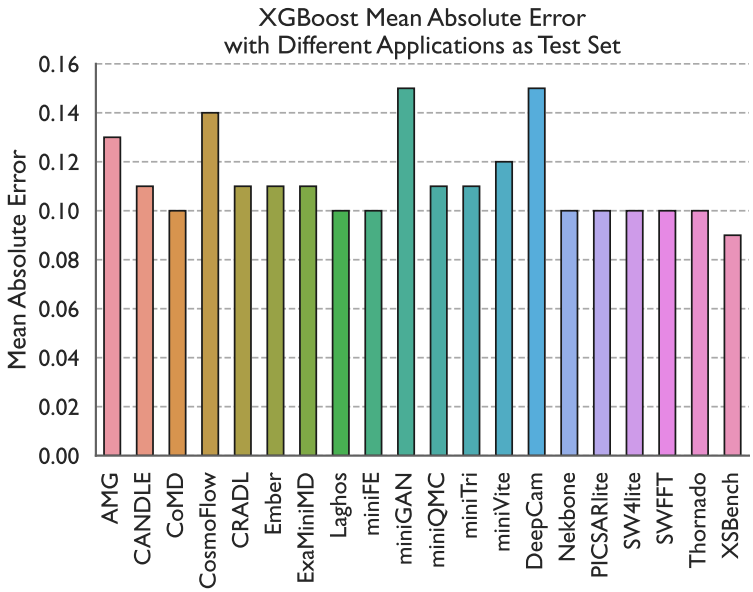


Figure 5.7: Evaluation MAE of XGBoost when each application is removed from the training set and used for evaluation. Results are generally strong across all applications.

5.6.3 Feature Importances

Figure 5.8 shows the feature importances for the XGBoost model. The most important feature is the ratio of branch to total instructions. This feature captures the control flow complexity of a program as those with more branch instructions have a more complex control flow. Since programs with more control flow generally perform worse on GPUs, the model likely uses this feature to make CPU-GPU predictions.

Next we see that the ratio of integer and single precision FP arithmetic to total instructions are the next most important features in prediction. These provide insight into the data throughput of the model. In this case, applications with higher data intensity are more likely to perform better on the GPU as they are designed for high throughput data-parallel computation. These two features combined with the branching intensity make sense as the three most important features as they help the model predict relative performance between CPUs and GPUs, which is where

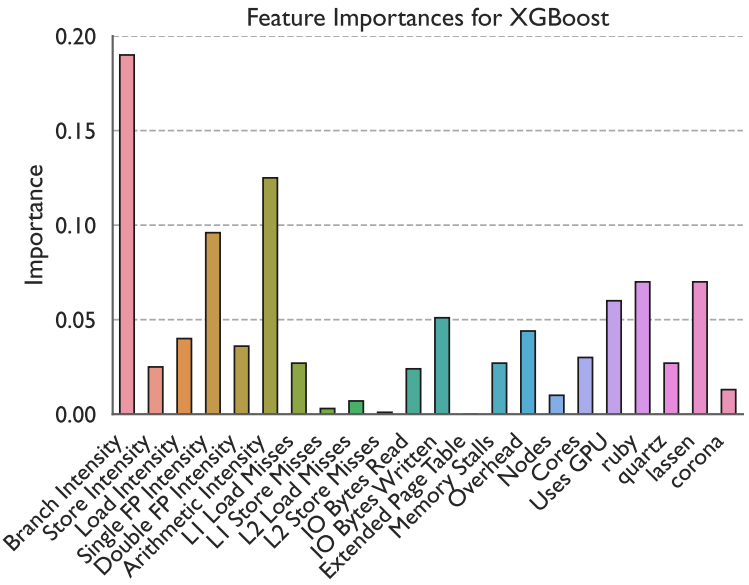


Figure 5.8: Importances of each feature in the XGBoost model. A higher feature importance value means it is more influential in the decision making of the model. The branch instructions intensity is the most important feature followed by the integer and floating point arithmetic intensity.

we see the largest performance differences in the data.

The next three most important features are Ruby, Lassen, and Uses GPU, which detail where the counters were collected. This is necessary for the model to predict the relative performance vector and is likely why these are the next three most important features. We also see that the L2 store misses and extended page table features are not used in the prediction, so we can remove these during feature selection.

5.6.4 Evaluation of Scheduling Simulations

Figures 5.9 and 5.10 show the results from the scheduling simulation. The first, Figure 5.9, lists the makespan for the scheduler with each machine placement algorithm. The Model-based machine placement method gives the lowest makespan at 0.87 hours on the set of jobs meaning it is able to finish the job set in a shorter amount of time than the others. Placing jobs on the

most efficient resource helps improve the makespan by allowing jobs to finish sooner. The next best method is the User+RR placement algorithm. This method represents how users submit jobs to the scheduler with only the limited knowledge of the performance of their applications across machines. This is followed by the Round-Robin and Random placement methods that perform the worst.

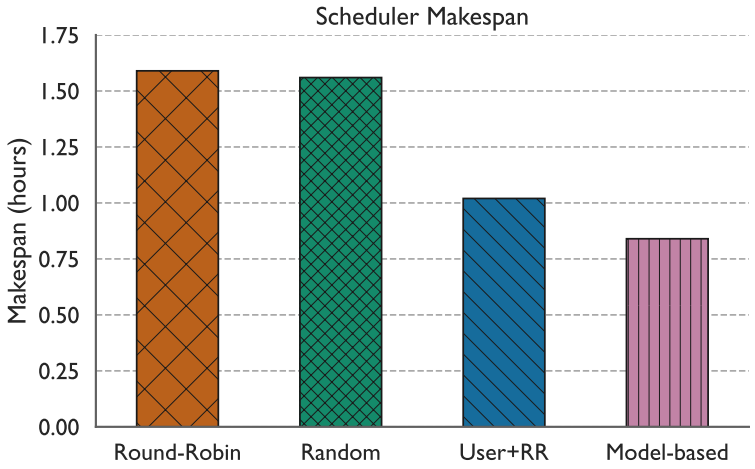


Figure 5.9: The makespan of each machine selection algorithm in the scheduling simulation. Lower is better.

Figure 5.10 shows the average bounded-slowdown for each machine placement method. The slowdown measures the ratio of wait time and run time to just run time. As with makespan the Model-based placement performances the best compared to the other algorithms.

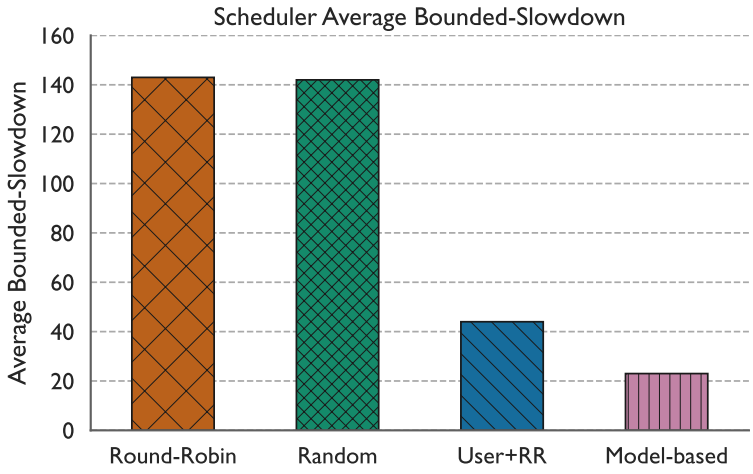


Figure 5.10: The average bounded-slowdown of each machine selection algorithm in the scheduling simulation. Lower is better.

Chapter 6: PAREVAL: Creating a Benchmark for Understanding Parallel Code Modeling Capabilities

Before developing techniques to model parallel code we first need to understand how capable current LLMs are at modeling parallel code. In this chapter, we introduce the PAREVAL benchmark for evaluating the parallel code modeling capabilities of state-of-the-art LLMs. The insights from this chapter will aid in incorporating code as an input modality to performance models. The contents of this chapter are presented in [102].

6.1 Motivation

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 [56, 6, 59, 5, 119, 73, 50]. Popular models such as StarCoder [83], 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" [71]. 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 [28]). Only more recent benchmarks such as DS-1000 [79], 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 chapter, 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:

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

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

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

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

6.2 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 6.1. 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.

Table 6.1: 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.

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 [111], MPI [128], MPI+OpenMP, Kokkos [135], CUDA [106], and HIP [62]. 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.

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

```
#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) {
```

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 [16]. 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.

Table 6.2: 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 [121]	6.7B	✓	llama2	29.98	41.4
CodeLlama-13B [121]	13.0B	✓	llama2	35.07	47.0
StarCoderBase [83]	15.5B	✓	BigCode OpenRAIL-M	30.35	49.0
CodeLlama-34B [121]	32.5B	✓	llama2	45.11	55.0
Phind-CodeLlama-V2 [115]	32.5B	✓	llama2	71.95	—
GPT-3.5 [27]	—	✗	—	61.50	52.2
GPT-4 [107]	—	✗	—	84.10	—

†HumanEval results are from the BigCode Models Leaderboard [23], except for GPT-3.5 and GPT-4 which are from [148].
‡MBPP results are from [121].

6.3 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.

6.3.1 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 6.5. 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.

6.3.2 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: $\text{serial} \rightarrow \text{OpenMP}$, $\text{serial} \rightarrow \text{MPI}$, and $\text{CUDA} \rightarrow \text{Kokkos}$. 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.

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.

```
// 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) {
```

6.4 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 6.2 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 [121] as variants of the Llama 2 model [134], 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 [83] is a 15.5B parameter model trained on 1 trillion tokens from The Stack [77]. 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 [11], 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 [115] 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 [23] 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 [27, 107]. 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 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.

6.5 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.

6.5.1 Metric for Correctness

We adopt the pass@k metric from [32] 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 (6.1).

$$\text{pass@}k = \frac{1}{|P|} \sum_{p \in P} \left[1 - \binom{N - c_p}{k} \Big/ \binom{N}{k} \right] \tag{6.1}$$

Number of samples generated per prompt

Set of prompts

Number of correct samples for prompt p

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 $100 \times \text{pass@}k$ as is common in related literature and online leaderboards [23, 31]. 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.

6.5.2 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: $\text{speedup}_n@k$ and $\text{efficiency}_n@k$.

$\textbf{speedup}_\textbf{n}@\textbf{k}$

The first metric, $\text{speedup}_n@k$, measures the expected best performance speedup of the generated code relative to the performance of a sequential baseline (see Section 6.6.2) 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 (6.2).

runtime of sequential baseline for prompt p

$$\mathbb{E}\left[\max\left\{\frac{T_p^*}{T_{p,s_1,n}},\ldots,\frac{T_p^*}{T_{p,s_k,n}}\right\}\right]=\sum_{j=1}^{N}\frac{\binom{j-1}{k-1}}{\binom{N}{k}}\frac{T_p^*}{T_{p,j,n}} \tag{6.2}$$

runtime of sample j of prompt p on n resources

To demonstrate that Equation (6.2) 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 (6.2) over all prompts we can define the $\text{speedup}_n@k$ metric as shown in Equation (6.3).

$$\text{speedup}_n@k = \frac{1}{|P|} \sum_{p \in P} \sum_{j=1}^{N} \frac{\binom{j-1}{k-1}}{\binom{N}{k}} \frac{T_p^*}{T_{p,j,n}} \tag{6.3}$$

For a single LLM, the $\text{speedup}_n@k$ 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 $\text{speedup}_n@k$ 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.

$$\text{efficiency}_{\text{max}}@k = \frac{1}{|P|} \sum_{p \in P} \sum_{\substack{j=1 \\ n \in \text{procs}}}^{N \cdot |\text{procs}|} \frac{\binom{j-1}{k-1}}{\binom{N \cdot |\text{procs}|}{k}} \frac{T_p^*}{n \cdot T_{p,j,n}} \tag{6.4}$$

The $\text{speedup}_n@k$ 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 $\text{speedup}_n@k$ 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.

$\mathbf{speedup_{max}@k}$

We also define a variant of the $\text{speedup}_n@k$ metric, $\text{speedup}_{\text{max}}@k$, as shown in Equation (6.5), which estimates the maximum speedup over all n and not a fixed resource count.

$$\text{speedup}_{\text{max}}@k = \frac{1}{|P|} \sum_{p \in P} \sum_{\substack{j=1 \\ n \in \text{procs}}}^{N \cdot |\text{procs}|} \frac{\binom{j-1}{k-1}}{\binom{N \cdot |\text{procs}|}{k}} \frac{T_p^*}{T_{p,j,n}} \tag{6.5}$$

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

$\mathbf{efficiency_n@k}$

To further understand the parallel performance of the generated code, we define the $\text{efficiency}_n@k$ 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 (6.3) to divide by n as shown in Equation (6.6). 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 $\text{efficiency}_n@k$, we also define $\text{efficiency}_{\text{max}}@k$ in the same fashion as Equation (6.5).

$$\text{efficiency}_n@k = \frac{1}{|P|} \sum_{p \in P} \sum_{j=1}^{N} \frac{\binom{j-1}{k-1}}{\binom{N}{k}} \frac{T_p^*}{n \cdot T_{p,j,n}} \tag{6.6}$$

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 $\text{speedup}_1@k$ 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.

6.6 Experimental Setup

This section describes how we generate outputs using each of the LLMs (Section 6.4) and the prompts in PAREVAL, and how we evaluated the generated code using the PAREVAL test harness.

6.6.1 LLM Inference: Generating Code Output

To generate outputs with the open-source models, we use the HuggingFace library [143] with PyTorch [113] 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 [108] via OpenAI's Python

client [109].

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 [83, 121]. 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.

6.6.2 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 6.5.2). 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 $1, 2, 4, \ldots, 32$ threads for OpenMP and Kokkos. For MPI, we run with $1, 2, 4, \ldots, 512$ 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 $1, 2, 4, \ldots, 64$ 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*).

6.7 Evaluation Results

We now present detailed results from evaluating the LLMs described in Section 6.4 using the ParEval prompts and test harness.

6.7.1 Experiment 1: Parallel Code Generation

To evaluate the correctness of the code generated by the LLMs we first look at the pass@1 scores over ParEval. Figure 6.1 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 (6.1), 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 6.2). 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

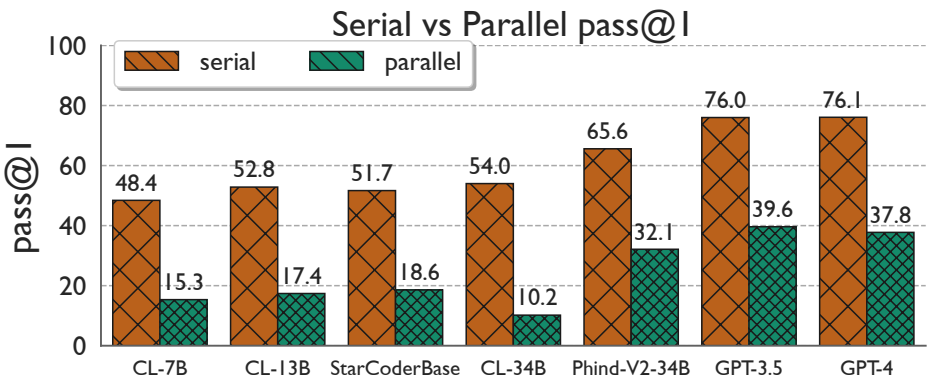


Figure 6.1: Each LLM's pass@1 score over PAREVAL. All of the LLMs score significantly worse in generating parallel code than serial code.

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.

Another interesting trend we observe in Figure 6.1 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 6.2 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 6.1 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.

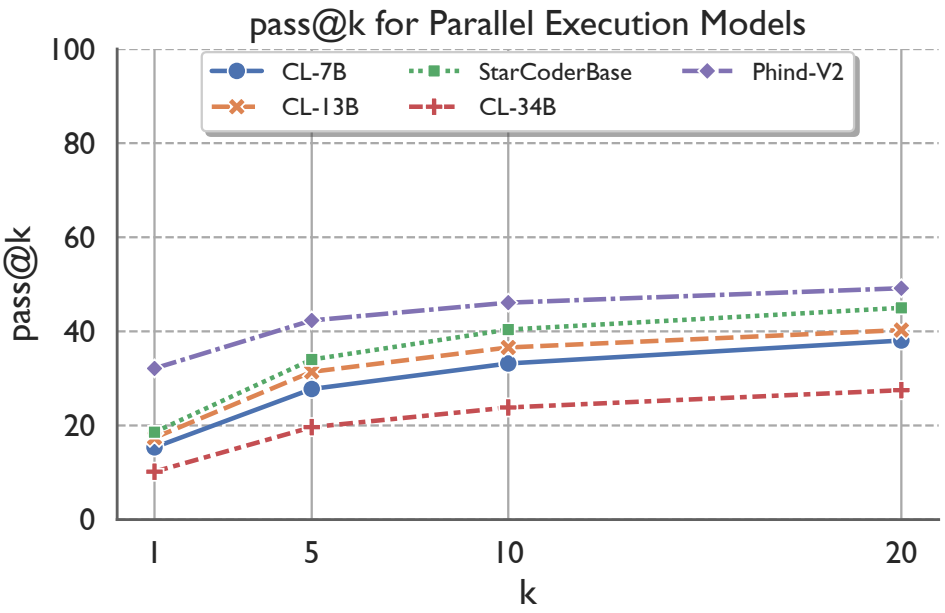


Figure 6.2: The pass@k for various values of k. The relative order of the LLMs is the same for all values of k with Phind-V2 leading the group.

6.7.1.1 Breakdowns by Execution Models

We further break down the pass@1 results by each execution model in Figure 6.3. 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.

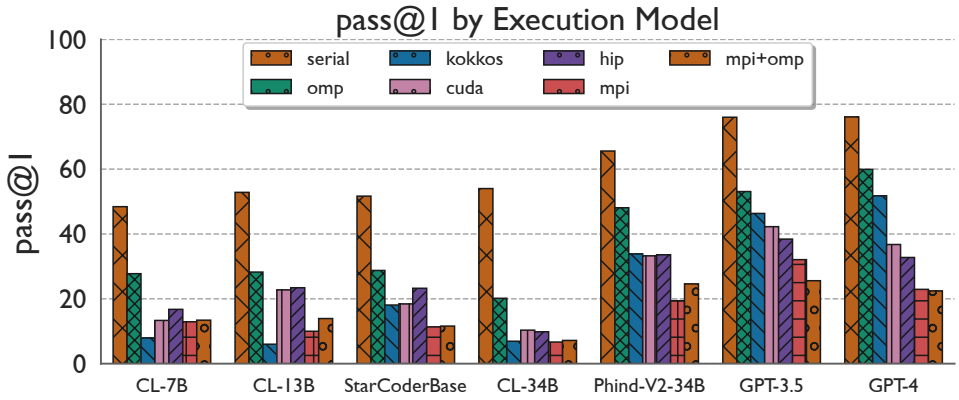


Figure 6.3: pass@1 for each execution model. The LLMs generally follow the same distribution of scores across the execution models: serial (best), OpenMP, CUDA/HIP, and MPI/MPI+OpenMP (worst) with Kokkos varying between LLMs.

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.

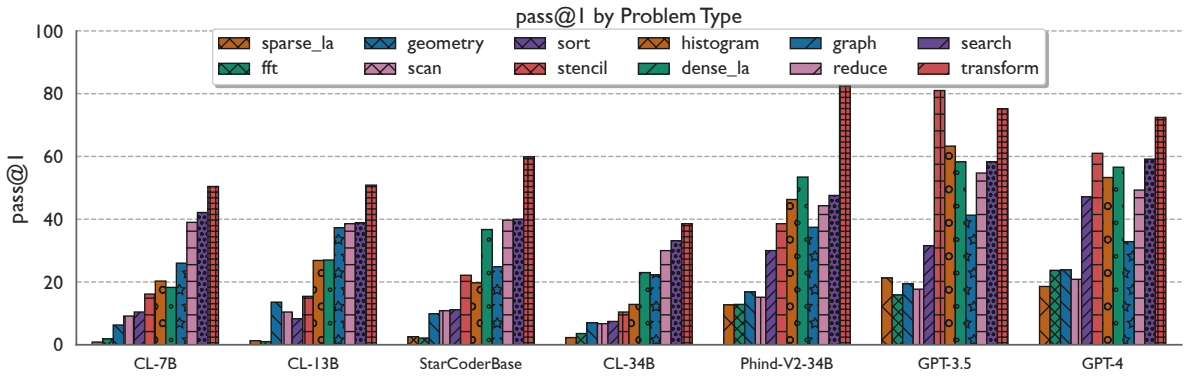


Figure 6.4: pass@1 for each problem type. The LLMs are best at transform problems, while they are worst at sparse linear algebra problems.

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.

6.7.1.2 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 6.4 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 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 6.5 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 6.3 and 6.4 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.

6.7.1.3 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.

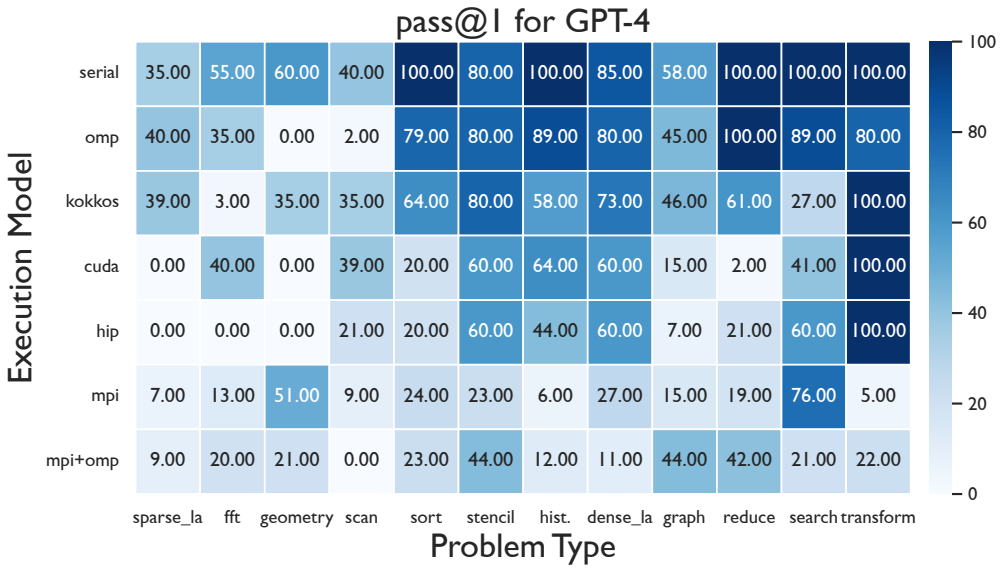


Figure 6.5: pass@1 for GPT-4 across all execution models and problem types. GPT-4 excels with the Kokkos and OpenMP execution models, while getting more problems correct for transform, search, and reduce problems.

Figure 6.6 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: $n = 32\,\text{threads}$ for OpenMP and Kokkos, $n = 512\,\text{processes}$ 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]

In Figure 6.6, we see a trend similar to the pass@1 scores in Figure 6.1, 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 6.6. From this we see that none of the LLMs use

[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.

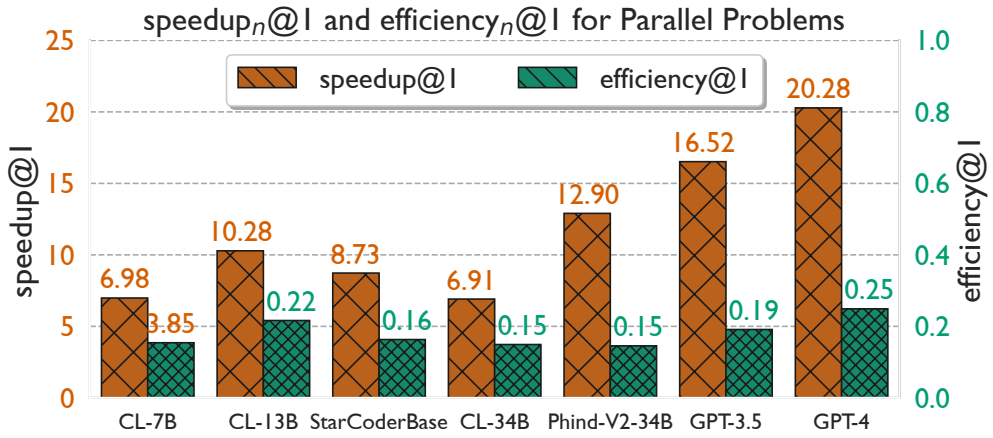


Figure 6.6: speedup$_n$@1 and efficiency$_n$@1 for parallel prompts. Results are shown for $n = 32$ threads for OpenMP and Kokkos, $n = 512$ ranks for MPI, and $n = (4 \text{ ranks}) \times (64 \text{ threads})$ for MPI+OpenMP. For CUDA/HIP n is set to the number of kernel threads, which varies across prompts. [1]

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 6.6 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 6.7 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 6.8 further shows the expected maximum speedup and efficiency across all resource counts n. We see the same trends as in Figure 6.6 with the speedups at similar values and the

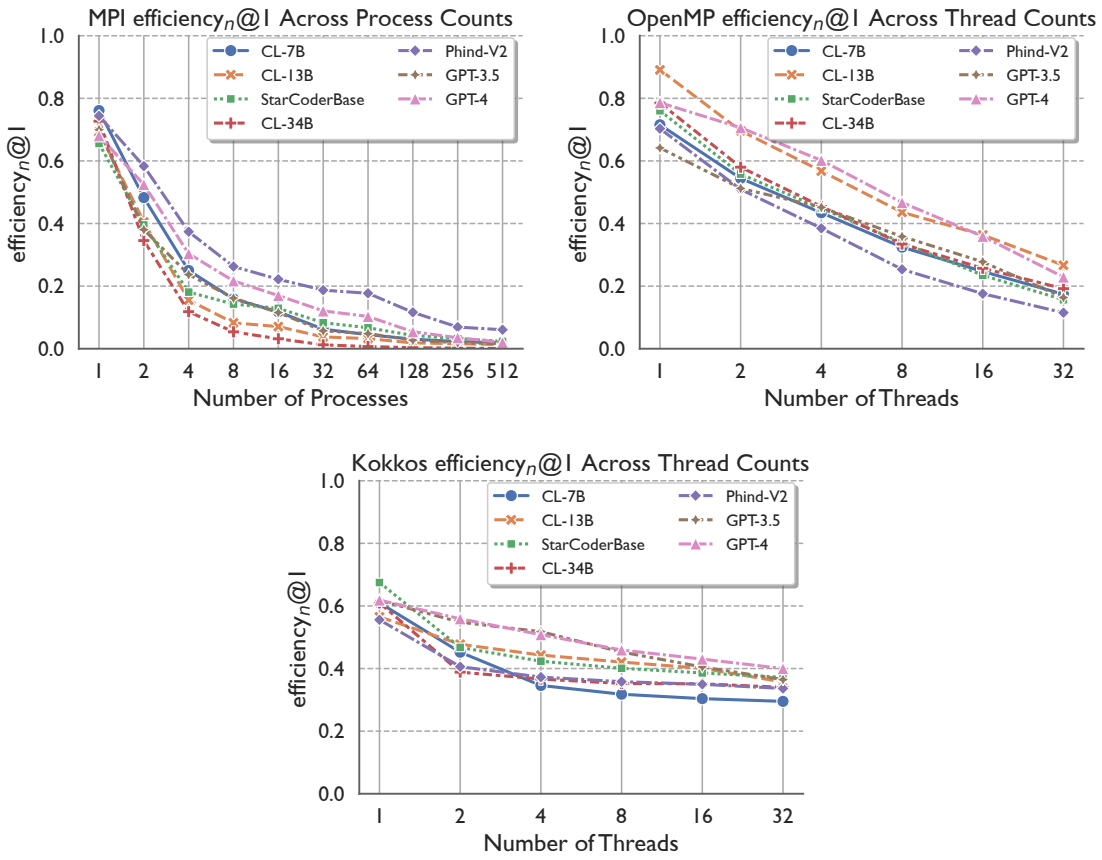


Figure 6.7: efficiency@1 for MPI (left), OpenMP (middle), and Kokkos (right) prompts across rank and thread counts. Phind-V2 is most efficient for MPI prompts, but is one of the least efficient for OpenMP and Kokkos. GPT-4 is the most efficient for OpenMP and Kokkos prompts. [1]

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.

6.7.2 Experiment 2: Parallel Code Translation

In addition to generating parallel code from scratch, we also evaluate the LLMs ability to translate between execution models (see Section 6.3.2). Figure 6.9 shows the pass@1 scores for each LLM for translating serial to OpenMP, serial to MPI, and CUDA to Kokkos. We also include the generation pass@1 scores from Figure 6.3 for each LLM for OpenMP, MPI, and Kokkos.

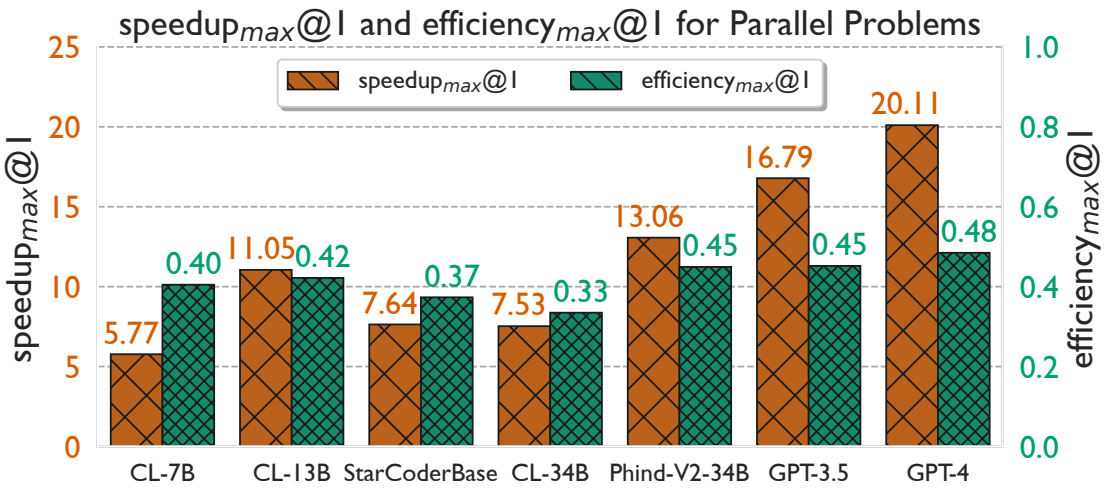


Figure 6.8: The expected max speedup and efficiency across all resource counts n.

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.

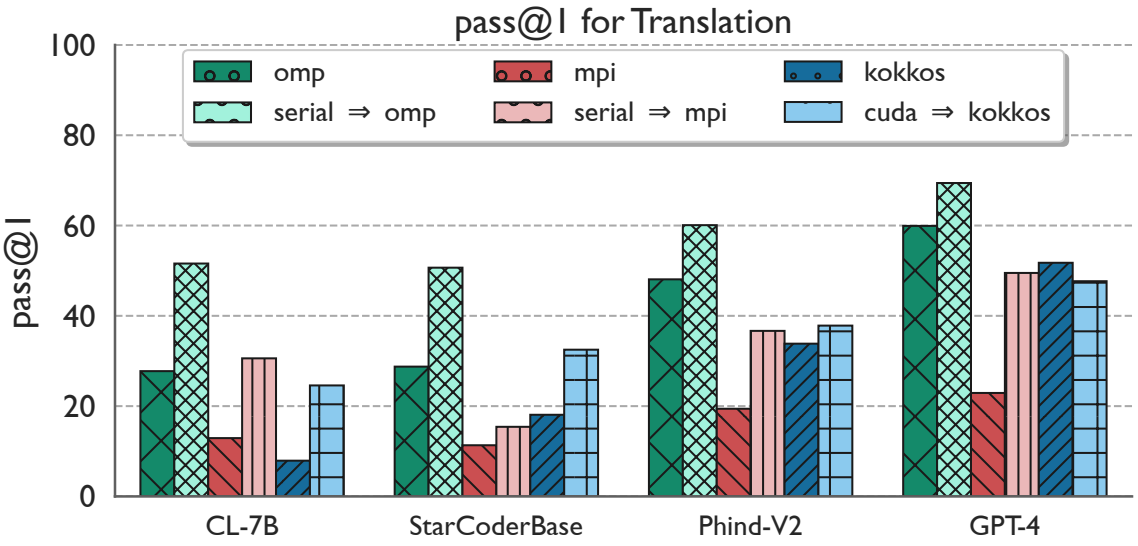


Figure 6.9: pass@1 for each LLM when translating serial to OpenMP, serial to MPI, and CUDA to Kokkos compared to the pass@1 score for generating code in the destination execution model. The smaller LLMs see a significant improvement when shown an example correct implementation.

6.7.2.1 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 6.10. 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.

We observe similar trends with OpenMP and Kokkos for speedup$_n$@1 as shown in Figure 6.11. 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 6.9 to 6.11 we conclude that providing LLMs with correct implementations in one execution model helps

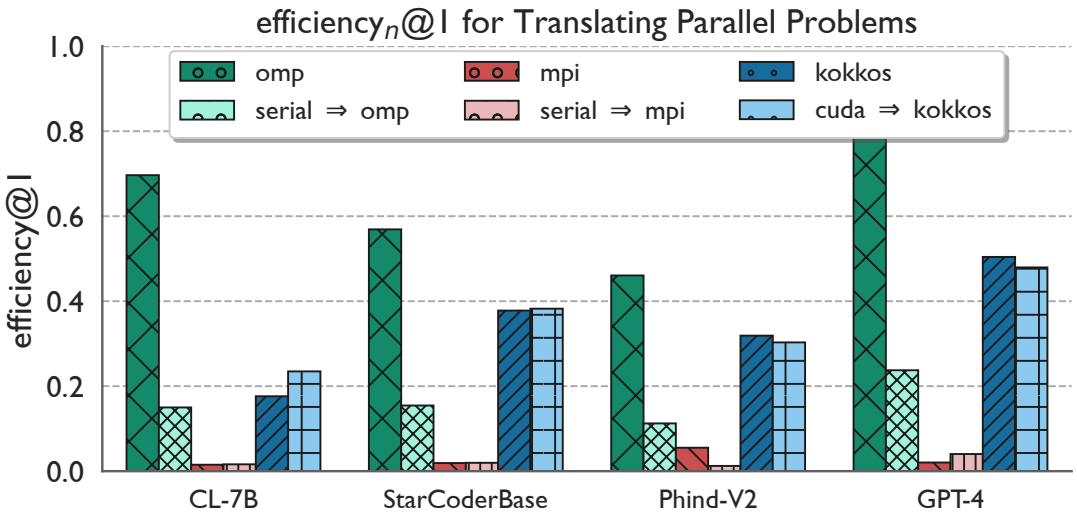


Figure 6.10: efficiency@1 translation scores compared to generation scores. The LLMs generally score similarly for translation and generation.[1]

them generate correct code in another execution model, but does not necessarily improve the performance of the generated code.

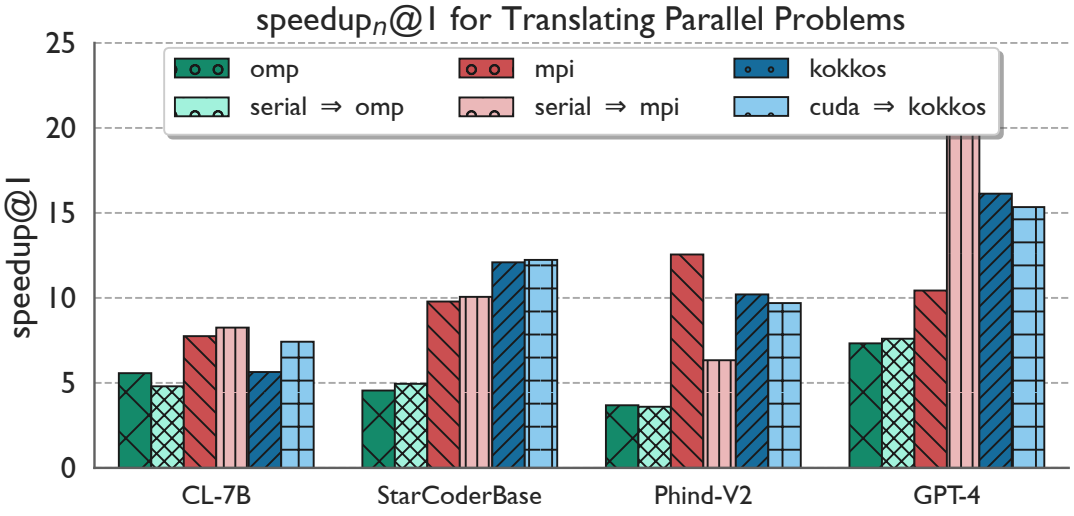


Figure 6.11: speedup@1 translation scores compared to generation scores. The LLMs generally perform similarly for translation and generation with the exception of MPI.[1]

Chapter 7: Modeling Parallel Programs with Large Language Models

This chapter presents work towards fine-tuning large language models to model parallel programs. These models are evaluated against several downstream tasks such as parallel code generation, sequential to parallel code translation, and performance modeling. It is found that fine-tuned language models are capable of modeling relative performance differences across commit history in real programs. The work presented in this chapter is published in [103].

7.1 Motivation

In recent years, large language models (LLMs) have become the state of the art for many language modeling related tasks [149]. Their ability to model token probabilities within a sequential context make them desirable for language tasks such as text generation and sequence classification. In addition to being used for natural language, such models have recently been applied to many programming language related tasks [31, 83, 121]. The predictive capabilities of these models translate well to coding tasks, and the wealth of open-source code available online provides significant data for training large models.

LLMs trained on source code data have been utilized to automate numerous software development tasks such as code completion, malware detection, code refactoring, etc [121, 83, 124, 96, 56, 6, 59, 5, 119, 73]. Additionally, they have been able to automate tasks previously consid-

ered impossible to automate such as code summarization and generation using natural language. Training LLMs for these tasks requires significant amounts of source code data that is fortunately available online from open-source code repositories on GitHub, gitlab etc. However, this data requirement for training LLMs is prohibitive for tasks where such data may not exist. One such task is that of modeling performance (execution time) based on source code. Another difficult task is modeling parallel and HPC code where there is less data available and it is often more complex code.

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

When a lack of data becomes a hurdle in machine learning tasks, it is typically solved through data augmentation and/or transfer learning. Data augmentation involves extending and/or duplicating data in a manner that still preserves meaning and representational capacity. Transfer

learning is done by first training a model on a related or simpler task and then *transferring* that knowledge to a new problem requiring fewer samples to learn. For our task we employ transfer learning by using LLMs that have learned to model source code and then transferring that knowledge to then learn how to model performance of source code using fewer samples. In particular, we explore modeling parallel and HPC codes.

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

7.2 Overview

Figure 7.1 provides an overview of the data gathering, training, and downstream application in this paper. In order to train a large HPC-specific language model we need a large dataset of HPC code. To obtain this, we gather a dataset of HPC source code and use it to fine-tune a pre-trained language model. This data gathering is described in Section 7.3 and presents what HPC sources are used and how they are pre-processed. Following this, the model fine-tuning and selection are detailed in Section 7.4 where we explain the training setup and methodology.

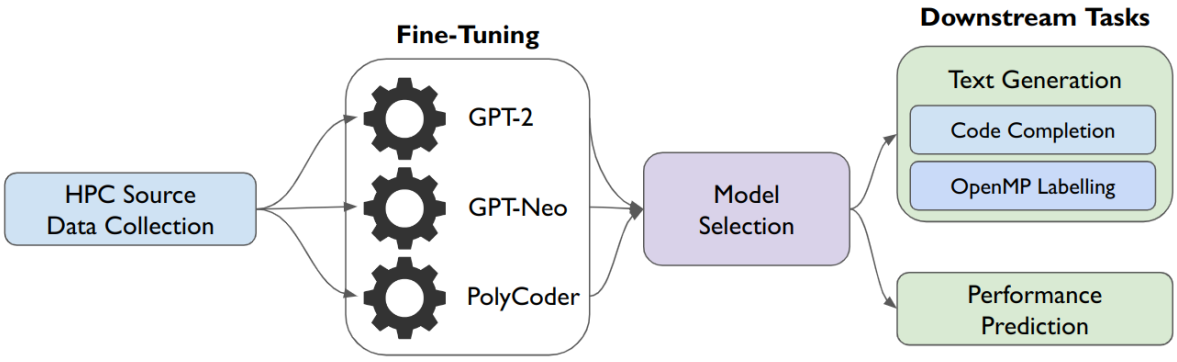


Figure 7.1: Overview of the steps described in this paper to train an HPC specific model and run it on several downstream tasks. After collecting a large dataset of HPC code we fine-tune several pre-trained language models and select the best one. The selected model is then used to generate code, label OpenMP pragmas, and predict relative performance as part of several downstream tasks.

We need several realistic tests to study the performance of the language model on relevant metrics. We present three main downstream tasks for evaluation in Section 7.5. The first two, code generation and OpenMP pragma labeling, test the model on its ability to generate correct and meaningful code. The last test, relative performance prediction, shows how this trained model can be used for useful tasks that require language comprehension. Results from each of these tests are presented and discussed in Section 7.6.

7.3 Data Gathering and Pre-processing

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

7.3.1 HPC Source Code Data

We first collect a sufficiently large dataset of source code to train the model on HPC and scientific code. The HPC source dataset is collected from GitHub repositories. The source files are pulled from repositories with C/C++ marked as the primary language and with ≥ 3 stars. The repositories are additionally filtered by HPC related GitHub *topics*. Once cloned, we collect all the C/C++ source files based on their file extension.

This dataset is collected and structured in the same manner as the C/C++ source dataset from Xu et al. [146]. Their dataset is scraped from GitHub in a similar manner with the exception of only including repositories with ≥ 5 stars. Figure 7.2 shows the distribution of lines of code (LOC) by file types in the HPC source dataset. There are roughly the same number of LOC in both C and C++ files. The distribution of actual file counts follows the same trend.

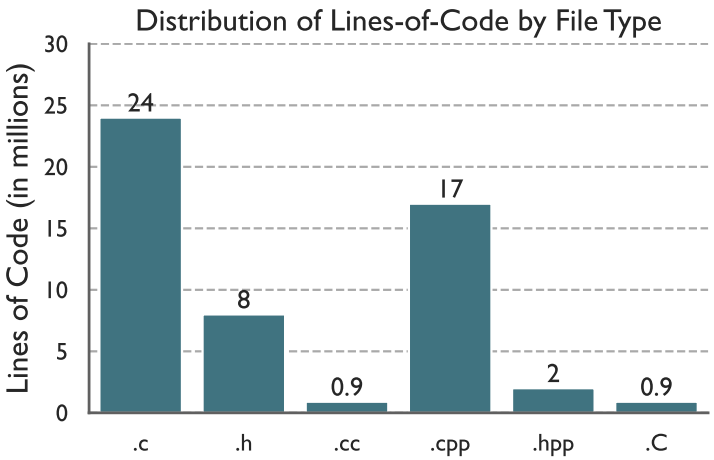


Figure 7.2: Distribution of no. of lines of code in each file type. *.cxx*, *.hh*, *.H*, and *.hxx* files are included in the dataset, but omitted here due to small counts.

7.3.2 Data Pre-processing

Allamanis [12] shows how duplicate source data, which is prevalent across GitHub repositories, can adversely bias LLMs during training. To prevent this we filter our datasets by removing duplicate files based on the hash of their contents. We use sha256 to hash the contents of the file.

In addition to deduplicating we also filter out small and large files. Source files larger than 1 MB are designated as large files and removed. These are generally entire libraries in a single source file or contain raw data within the code. Additionally, files containing less than 15 tokens, as defined by the language vocab, are not included. The reduced dataset sizes after deduplication and filtering are listed in Table 7.1. Approximately 18% of the files are removed during this processing. Table 7.1 shows the properties of the dataset after each step of deduplication and filtering.

Table 7.1: Properties of the HPC source code dataset.

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

After filtering source files, we tokenize the dataset to obtain integer values for the text that can be used as input into the model. We use the pre-trained tokenizers for each of our selected models (see Section 7.4). These are all GPT-2 [116] based Byte-Pair Encoding (BPE) tokenizers.

7.3.3 Performance Datasets

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

We create two datasets – one with pairs of code that are functionally different and one where they are the same. The first dataset is created by using version control history to capture performance regressions. We run each commit for the Kripke [78] and Laghos [41] applications. These are small HPC apps meant to mimic the computational behavior of larger scientific applications. We automate building and running each commit to the best of our ability and collect performance results for 830 commits in total.

The second dataset is a set of programming competition solutions from the *code_contests* dataset [85]. These are aggregated from several online programming competitions: Aizu, AtCoder, CodeChef, CodeForces, and HackerEarth. This dataset allows us to create pairs of code that solve the same problem (the contest problem), but may be different in implementation. We run every correct solution for each problem in the dataset, with the corresponding problem's test cases as inputs, and record the run time. Using all the C++ solutions in the dataset we create ~1.7 million samples of code. Using the run times, we group the solutions into pairs and label them as *slower* and *faster* pairs.

7.4 Fine-Tuning Methodology

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

7.4.1 Models Selected For Fine-tuning

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

Table 7.2: Description of the models used for fine-tuning.

Model	# Params.	# Layers	Hidden Size	Window Size	Pre-Training Set
GPT-2 [116]	1.5B	48	1600	1024	WebText [53]
GPT-Neo [24]	2.7B	32	2560	256	Pile [48]
PolyCoder [146]	2.7B	32	2560	2048	Source Code

Keeping the above mentioned requirements in mind, we select several models for fine-tuning and/or testing. These are listed in Table 7.2. All of these are based on GPT-2 [116] and/or GPT-3 [27] architectures with slight variations in size, configuration, and pre-training data. GPT-2, the smallest in our experiments, is pre-trained on the WebText [53] dataset, which is a collection of language data scraped from the internet. We use the 1.5 billion parameter GPT-2 model variant in this paper. PolyCoder [146] is pre-trained on a collection of solely source code data from GitHub that contains a mixture of 12 popular programming languages [146]. Between these two is GPT-Neo [24] that is pre-trained on the Pile dataset [48]. This dataset contains a collection of approximately 800GB of text data from the internet, academic articles, source code,

etc. Notably this dataset has a mixture of natural language and code. It has been demonstrated that pre-training over *both* natural language and code can improve the performance of the model.

We exclude models such as GPT-4 [107], the state-of-the-art model that powers GitHub CoPilot, from our experiments due to the model and its dataset being closed source. It is currently only accessible for inference via a non-free API. GPT-4's dataset being closed source is significant as we cannot remove data it has trained on from the dataset we use to evaluate its performance, so its results would be overly optimistic. This prevents a realistic evaluation and comparison.

7.4.2 Fine-tuning Setup and Hyperparameters

We rely on the functionality provided in the HuggingFace [143] Python library for fine-tuning the models. This library automates many of the tasks related to loading and pre-processing datasets, and running language models on the datasets. In particular, we use the `Trainer` interface with DeepSpeed [95] as the backend to optimize fine-tuning. DeepSpeed is a framework that provides distributed training functionality and several memory optimizations to enable large models to fit in GPU memory.

Starting with the pre-trained models, we fine-tune them on a single node with an AMD EPYC 7763 CPU, 512 GB memory, and four 40 GB NVIDIA A100 GPUs. With DeepSpeed's ZeRO memory optimizations [118], all of the models fit entirely within a single A100 GPU and are, thus, fine-tuned using pure data parallelism. We refer the reader to [20, 101] for a comprehensive overview of training deep neural networks in parallel.

We use the AdamW [90] optimizer for all the models to update model weights and mini-

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

7.5 Downstream Inference Tasks and Evaluation Metrics

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

7.5.1 Code Completion

A standard benchmark for code generation tasks is the HumanEval benchmark [32]. This is comprised of 164 sample Python problems, where the input to the model is a natural language description of a function and function header. The model generates code for the function implementation, and is scored on functional correctness rather than textual similarity or equivalence.

We introduce our own adaptation of this benchmark for HPC C/C++ programs. Our benchmark consists of 25 custom HPC code generation problems including simple numerics, OpenMP parallel code, and MPI routines. Table 7.3 lists the tests used in our evaluation. Figure 7.3 shows a sample prompt (top) and output (bottom) for a shared-memory parallel implementation of `saxpy`. The prompt is provided as input to the model and it is expected to generate text

functionally equivalent to the text on the bottom.

Table 7.3: Code generation tests. OpenMP and MPI columns denote if the test includes a version with that parallel backend.

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

(a) Prompt

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

(b) Output

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

Figure 7.3: An example prompt asking the model to generate a parallel version of saxpy. The comment and function header make up the prompt. The function body on the bottom shows a potential model output.

Evaluation Metric: We first record the ratio of generated samples that build correctly to those

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

$$\text{pass@}k = 1 - \binom{N_p - c_p}{k} \Big/ \binom{N_p}{k} \tag{7.1}$$

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

$$\text{average\_pass@}k = \frac{1}{P} \sum_{i=1}^{P} \left[1 - \frac{\binom{N_i - c_i}{k}}{\binom{N_i}{k}} \right] \tag{7.2}$$

This metric provides insight into the probability of a model generating functionally correct code. In our experiments, we calculate the pass@k score for several temperatures, namely 0.1, 0.2, 0.4, 0.6, and 0.8, and select the best one. This is in line with experiments in related literature [146]. For each temperature and prompt, we generate $N_p = 100$ samples. The code is generated with nucleus sampling using 0.93 as the cutoff value in the CDF.

To compile the generated code samples, we use `g++` with the "`-O2 -std=c++17 -fopenmp`" flags. For tests that need MPI we use the OpenMPI `mpicxx` compiler. If the build is successful, then a corresponding driver binary is called that will call and test the generated function for correctness. These are run on a AMD EPYC 7763 CPUs with 64 physical cores at 2.45 GHz

each. For tests that require OpenMP or MPI we only denote them as correct if they used the corresponding parallel framework to compute their result.

7.5.2 Predicting OpenMP Pragmas

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

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

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

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

Evaluation Metric: To measure the success of this test, we use the accuracy of generating correct

pragmas. This is calculated as shown in Equation 7.3.

$$\text{accuracy} = \frac{\text{\# correct pragmas}}{\text{total pragmas tested}} \tag{7.3}$$

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

7.5.3 Relative Performance Prediction

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

Further Fine-tuning: In order to use the models for relative performance classification we need to first fine-tune them on new data for this output task. Using the Git commit data from Section 7.3.3 we give the model text for a region of code before and after a Git commit. The codes are concatenated with a unique token separating them, namely `<COMMIT>`. We repeat a similar process for the code contest dataset, but instead separate pairs by the token `<PAIR>`. With this data the model is fine-tuned to predict whether the second code will be slower (*positive*) or the

same/faster (*negative*).

For each dataset we fine-tune the model on 90% of the data with the other 10% set aside for evaluation. The model takes the concatenated sequences of the two versions of the code implementation and is fine-tuned for the binary classification problem of predicting relative performance. The training objective is classification accuracy, which we also use to measure success for this task.

Evaluation Metric: To evaluate the performance on this task we measure the model's classification accuracy. This is calculated as shown in Equation 7.4.

$$\text{accuracy} = \frac{\text{\# correct performance predictions}}{\text{total performance predictions}} \tag{7.4}$$

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

7.6 Results

We now present the fine-tuning and evaluation results using the downstream tasks discussed in Section 7.5.

7.6.1 Fine-tuning on HPC Source Code Data

We first show the results of fine-tuning the three models selected in Table 7.2. Table 7.4 shows the validation perplexity at the end of fine-tuning. Here perplexity is calculated as the exponential of the loss. Each model converges to a low perplexity score over the separate testing set (between 2 and 4). GPT-Neo and PolyCoder achieve comparable perplexity scores (within

0.01) while GPT2 achieves a higher perplexity. All three have different pre-training datasets and the former two are of a larger size than GPT2 (see Table 7.2). From this we can conclude that for this problem the pre-training dataset had less of an impact on validation perplexity than the model size. The lower perplexity of the larger models means that they model the language better.

Table 7.4: Final validation perplexities for each model after fine-tuning on the HPC source code dataset.

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

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

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

To explore this phenomenon we ran the code generation tasks every 1000 samples during fine-tuning of the PolyCoder model. Figure 7.4 presents the results from our evaluation tests during fine-tuning on the PolyCoder model. After seeing about 45,000 samples during fine-tuning the model starts to decrease in evaluation performance. This is in contrast to the perplexity which keeps improving past 45,000 samples. Based on this result we stop fine-tuning at 45,000 samples and use these weights for the rest of the evaluations. Additionally, due to the computation time

needed to run this test we use the 45,000 samples stopping point for fine-tuning all the models.

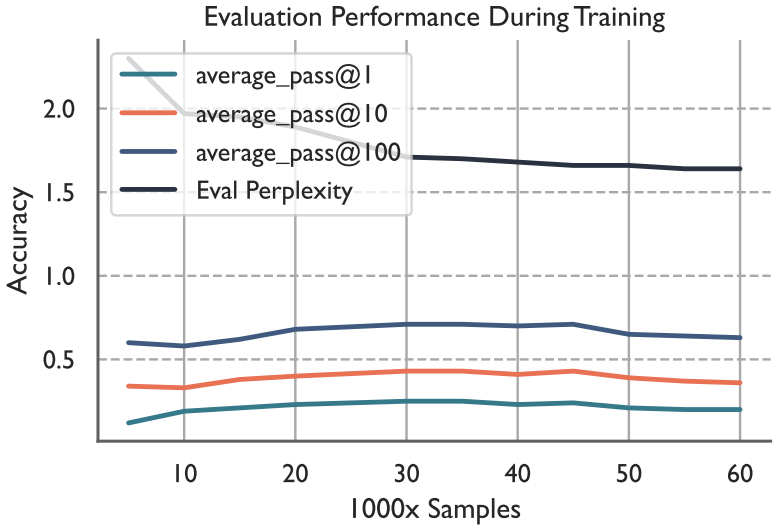


Figure 7.4: Downstream evaluation performance across training iterations for PolyCoder+HPC. The model starts to perform worse around 45,000 samples even though the perplexity keeps improving.

7.6.2 Code Completion

Having fine-tuned the three models, we now start using them for the different downstream tasks described in Section 7.5. The first downstream task is code generation, described in Section 7.5.1. Figure 7.5 shows the average_pass@k rates for the code generation tests. The average_pass@k values are computed according to Equation 7.2. We use PolyCoder as a baseline for comparison since it is a state-of-the-art LLM for code generation. PolyCoder+HPC scores the best for average pass@1, pass@10, and pass@100. For each value of k the models score in the order of PolyCoder+HPC, PolyCoder, GPT-Neo+HPC, and GPT2+HPC. PolyCoder+HPC gains the slight edge over the original PolyCoder by successfully generating code for the HPC-specific tasks (see Figure 7.6).

In Figure 7.5 we see that GPT2+HPC scores significantly lower than the other models. This

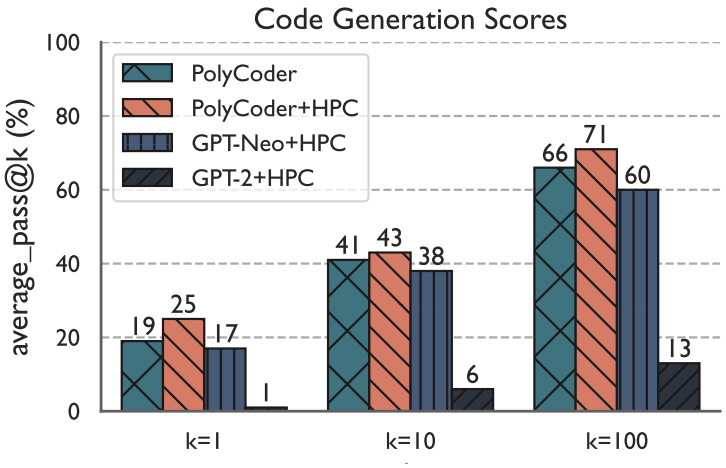


Figure 7.5: Comparison of models on code generation. The clusters represent the average pass@k scores for $k = 1, 10$ and 100. Higher percentage is better.

is likely due to the smaller model size and the fact that there is no source code in its pre-training dataset. In this instance fine-tuning is not enough to enable GPT-2 to generate correct C++ HPC code.

Altogether, the scores are indicative that PolyCoder+HPC and GPT-Neo+HPC has learned how to generate valid C++ code. For instance, if the best model, PolyCoder+HPC, is permitted to generate 100 samples, then 71% of them are correct on average across all the tests. Similarly for 1 sample generated this is 25%. These numbers roughly align with results from [146] on the HumanEval Python tests. However, the results are not directly comparable since they are a different set of tests in a different programming language.

To demonstrate the generative capabilities of the specialized models we reduce the code generation tasks to those that are specific to HPC. This includes code that uses OpenMP and/or MPI parallelism. Figure 7.6 shows the performance when restricted to these tests. We see that PolyCoder is unable to generate OpenMP and MPI code as it scores significantly lower than the rest. GPT2+HPC still performs fairly low, however, its score has actually improved slightly over

Figure 7.5. This is due to the fact that it has only seen HPC-specific code during training and that is what is being tested here.

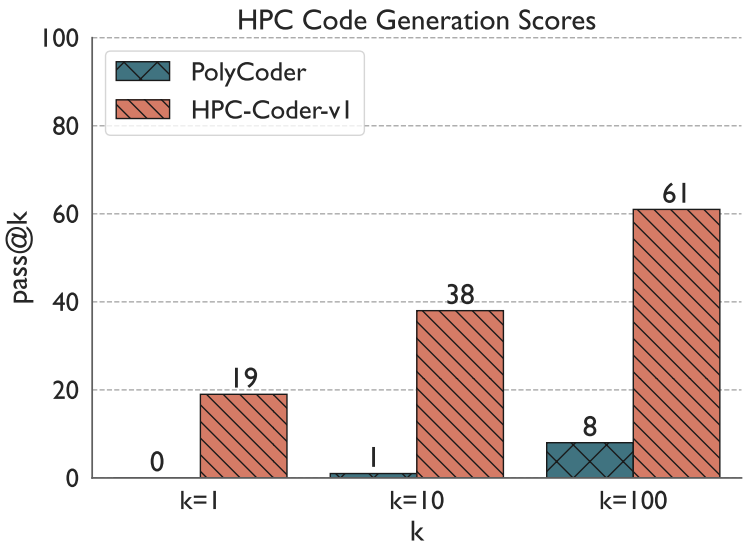


Figure 7.6: Comparison of models on code generation for HPC-specific functions. The clusters represent the average pass@k scores for $k = 1, 10$ and 100. Higher percentage is better.

Another point of interest besides functional correctness is syntactic correctness. This can be measured by the total number of generated samples that compile successfully. This is how often the model generates valid code, whether it is functionally correct or not. This data is presented in Figure 7.7. PolyCoder and PolyCoder+HPC both perform the best compared to the other models with 84% and 86% of samples compiling correctly, respectively. GPT-Neo+HPC performs slightly worse at 74% and GPT2-HPC has only 30% of samples compile. The worse performance of the latter two can likely be attribute to their pre-training datasets having less code. We also observe that for all models there is a visual correlation between build and correctness rates, which is expected as a model needs to compile in order to be functionally correct.

The code in Figure 7.8 shows example output from PolyCoder and PolyCoder+HPC on generating OpenMP code to compute a sum in parallel. We see that PolyCoder is able to produce

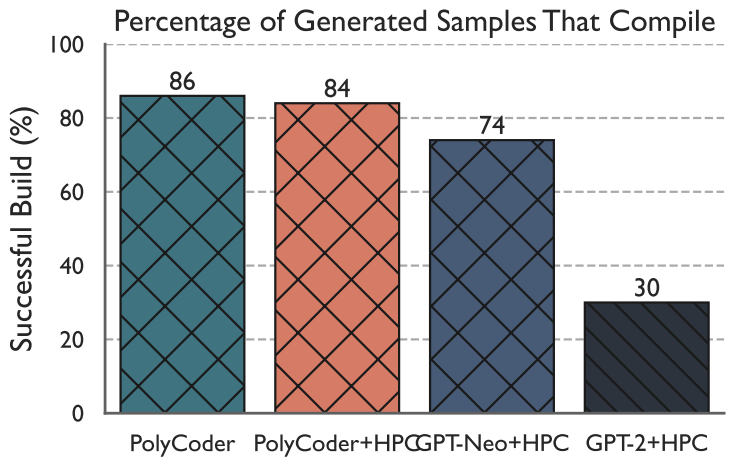


Figure 7.7: Comparison of the models' build rate. Both PolyCoder and PolyCoder+HPC have the best percentage of total samples that successfully compile. Higher percentage is better.

correct sequential code, however, it fails to add any OpenMP pragmas. The PolyCoder+HPC model is able to correctly tag the for loop with an OpenMP pragma.

A similar example with distributed memory is shown in Figure 7.9. The PolyCoder+HPC is able to generate correct MPI code to compute an average in parallel across ranks. PolyCoder often generated long and incorrect code that demonstrated little understanding of how to write MPI routines. The example presented is indicative of the rest of the MPI samples generated by PolyCoder. Notably, some instances did contain MPI calls, so PolyCoder must have some MPI code in its pre-training dataset. However, it lacked the ability to properly use MPI.

Figure 7.10 shows the speedups for the code generated by PolyCoder+HPC over sequential baselines. These are hand-written efficient, sequential implementations for each test. We see that PolyCoder+HPC is able to generate code that is faster than the sequential baseline. This demonstrates that it is not generating very poor performing parallel code and is likely using the parallelism correctly.

Since PolyCoder+HPC scores the highest in training and these code generation tests we

(a) Prompt

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

(b) PolyCoder output

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

(c) PolyCoder+HPC output

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

Figure 7.8: Example OpenMP output from (b) PolyCoder and (c) PolyCoder+HPC. The comment and function description (top) make up the prompt that is given to the model, while the bottom two blocks are the generated text. We see that PolyCoder is unable to generate OpenMP pragmas for the reduction in this example.

select it for further comparisons in the rest of the paper. PolyCoder+HPC is the fine-tuned model we present as HPC-Coder. We continue to use PolyCoder as a baseline.

7.6.3 Predicting OpenMP Pragmas

Next, we examine the result from the OpenMP prediction tests described in Section 7.5.2. Figure 7.11 shows the results from the OpenMP experiments detailed in Section 7.5.2. We see that both models are able to generate functionally correct OpenMP pragmas with high accuracy (right plot). PolyCoder+HPC is able to do this with 97% accuracy and PolyCoder 94%. The

LLMs are exemplary at understanding the dependencies of the `for` loop and what clauses are required to correctly parallelize them. We see that the model that has seen large amounts of OpenMP code performs better.

We can also look at how well the models reproduce the pragmas exactly. This means all the clauses and variables within those clauses are in the same order in the dataset and in the output from the model. These results are shown in the left plot in Figure 7.11. While less meaningful than functional correctness, it is interesting that the model is able to exactly reproduce pragmas it has not seen before with relatively high accuracy (67% and 61%). This is likely due to certain trends in the construction and ordering of OpenMP clauses that the LLMs are learning as they train.

7.6.4 Relative Performance Prediction

Finally, we look at the results from the relative performance prediction tests described in Section 7.5.3. Figure 7.12 shows the results from the relative performance prediction tests (see Section 7.5.3). Both models achieve high classification accuracy with PolyCoder+HPC being slightly better for the two proxy applications at 88% and PolyCoder at 86%. This means that for 88% of the code changes in the two repositories version control history PolyCoder+HPC is able to correctly identify if there will be a performance slowdown. Likewise for the programming competition dataset we see that PolyCoder+HPC outperforms the PolyCoder baseline with an accuracy of 92% vs 86%. This is a higher accuracy improvement than the proxy applications by 4 percentage points. This is likely due to the fact that the programming competition dataset is larger and PolyCoder+HPC has been trained on more C/C++ code.

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

(a) Prompt

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

(b) PolyCoder output

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

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

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

(c) PolyCoder+HPC output

```
    int size;
    double sum;

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

    return sum / size;
}
```

Figure 7.9: Example MPI output from (b) PolyCoder and (c) PolyCoder+HPC. The highlighted region is code generated by the model (reformatted to fit the column). PolyCoder results varied significantly, however, the above example demonstrates the general lack of understanding it had for MPI.

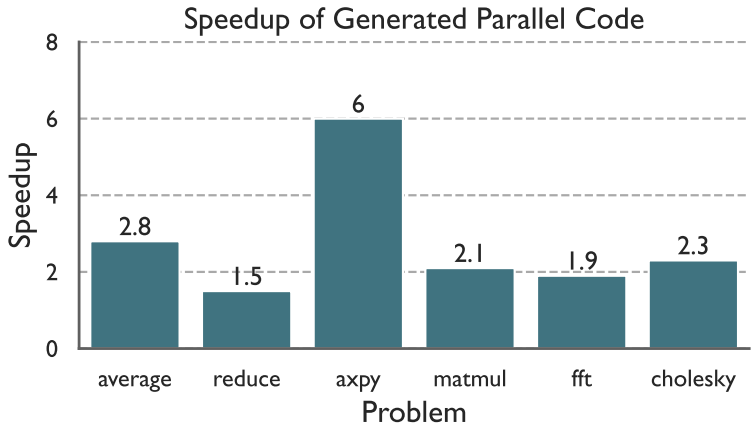


Figure 7.10: Comparison of the speedups for the code generation tests over sequential baselines. They are all above 1 demonstrating that the model is not generating very poor performing parallel code.

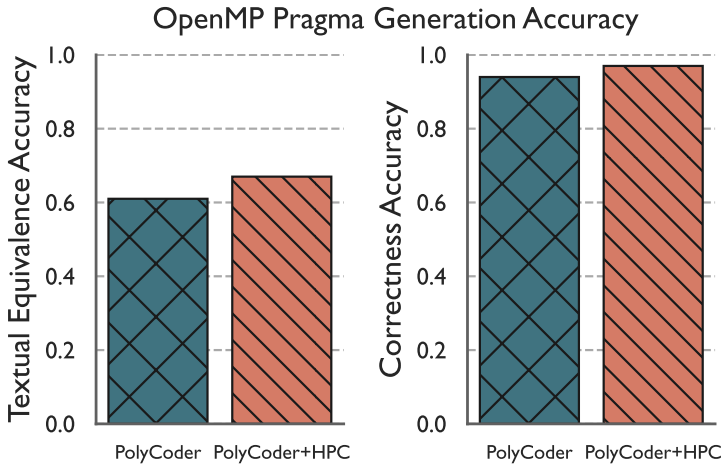


Figure 7.11: Comparison of models on predicting OpenMP pragmas. The left plot presents accuracy in predicting OpenMP pragmas exactly as they appear in the dataset. The right plot shows the accuracy in predicting functionally correct OpenMP pragmas. Higher accuracy is better.

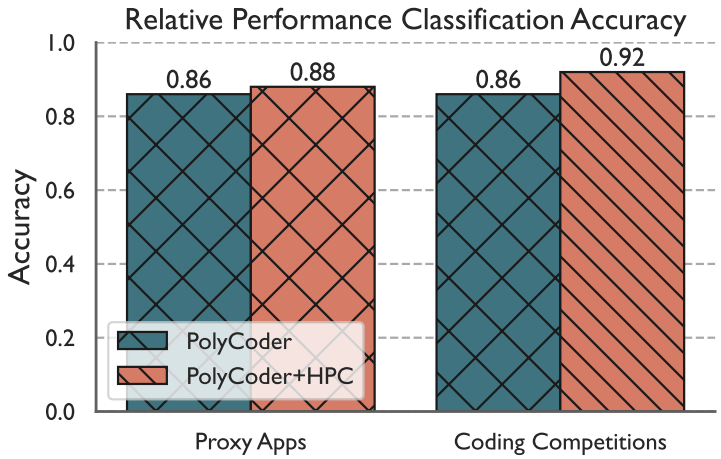


Figure 7.12: Comparison of models on predicting relative performance of code changes. Both models achieve similarly high accuracy. The PolyCoder+HPC model performs slightly better on both datasets. Higher accuracy is better.

Chapter 8: Understanding LLM Capabilities to Model Parallel Code: A Detailed Ablation Study

8.1 Motivation

Large language models (LLMs) have been a transformational technology in aiding software development. Their ability to automate coding tasks and connect natural language descriptions to code has improved developer productivity and enabled developers to more rapidly move from concept to implementation. As of 2023 over 92% of surveyed developers use AI in some form to aid their development process [125]. Beyond general development assistance these tools have the potential to enhance developer capabilities on more complex programming tasks such as writing parallel code. Writing correct, parallel code is an important problem facing modern developers and is already difficult for humans. Using LLMs to improve the quality and quantity of parallel code is an important step in improving the performance of modern software.

While code LLMs have shown promise in their code generation capabilities, they still struggle with more complex programming tasks such as parallel code. Previous work [102] has extensively studied LLMs across various parallel execution models and algorithms and found that LLMs are significantly worse at generating parallel code compared to sequential code. Two main reasons are identified for this discrepancy: the lack of parallel code data in the pre-training data of

modern LLMs and the intrinsic difficulty of parallel code generation. Solving the latter issue is a long-term effort that will require the development of more sophisticated AI models that can plan and reason through complex problems. However, the former issue of obtaining high-quality parallel code data at scale and effectively learning from that data is a much more tractable problem to tackle with current language modeling capabilities.

Creating HPC and parallel capable LLMs offers a great number of benefits to the HPC community. They will drastically improve the productivity of scientific developers and, in turn, the speed at which scientific discoveries are made. The process of designing these HPC capable LLMs will involve the creation of large HPC datasets and studies into modeling that data. Building out a large corpus of HPC data and understanding how to best learn from and model that data will be critical to developing future HPC AI developer tools. As the field of AI and code LLMs continues to progress it is important that the HPC community understands and addresses the unique challenges associated with HPC code generation.

Gathering parallel code data at scale and effectively learning from it is difficult. The data samples are already underrepresented in large code datasets and simply collecting more is often not enough; high-quality parallel code data is needed to train models effectively. This is evinced by the results of the StarCoder2 project which trained code LLMs on The Stack v2 dataset that contains nearly all permissively licensed code and code related data online [91]. Despite the impressive data collection efforts, the StarCoder2 models perform similar or worse than comparable models trained on less data. This suggests that we cannot keep improving model performance by collecting more data, but rather we need to collect better data. Furthermore, it is not well understood what makes data "better" for training code LLMs.

In this paper we address the lack of high-quality parallel code data by creating a large

synthetic code dataset, PARALLEL-INSTRUCT, using our proposed methodology to map existing parallel code samples to high-quality instruct-answer pairs. We then fine-tune code LLMs on this dataset and evaluate them against other code LLMs on ParEval [102], a state-of-the-art parallel code generation benchmark. We find that our fine-tuned model, Parallel-Coder, is the best performing open-source code LLM for parallel code generation and performs near GPT-4 level. We conduct an in-depth study to better understand how data representation and training parameters impact the models ability to learn how to model parallel code. These insights will be critical for future efforts developing the next generation of HPC AI developer tools.

In this paper we make the following important contributions.

- We collect a large synthetic dataset of high quality parallel code instruction data, PARALLEL-INSTRUCT.
- We fine-tune a code LLM, Parallel-Coder, that is the most capable open-source code LLM for parallel code generation and completion.
- We conduct an in-depth study along the data and fine-tuning parameters to understand how to best fine-tune code LLMs for parallel code generation.

Furthermore, we answer the following research questions:

RQ1 *How does the choice of fine-tuning base model and the use of instruction masking impact the performance of a code LLM on parallel code generation?*

RQ2 *How does the amount of fine-tuning data for a particular parallel execution model affect the performance of a code LLM on that model?*

RQ3 *How does the quality of parallel code fine-tuning data impact the performance of a code LLM on parallel code generation?*

RQ4 *How does model size impact the ability of a code LLM to learn from distilled synthetic data?*

8.2 Approach to Studying Data and Model Design Impacts on Parallel Code Modeling

Our approach to improving Code LLMs for parallel languages involves creating a large synthetic code dataset, PARALLEL-INSTRUCT, and then fine-tuning existing pre-trained Code LLMs on this dataset. We first present an overview of our proposed approach (Figure 8.1) and then present details of the various components.

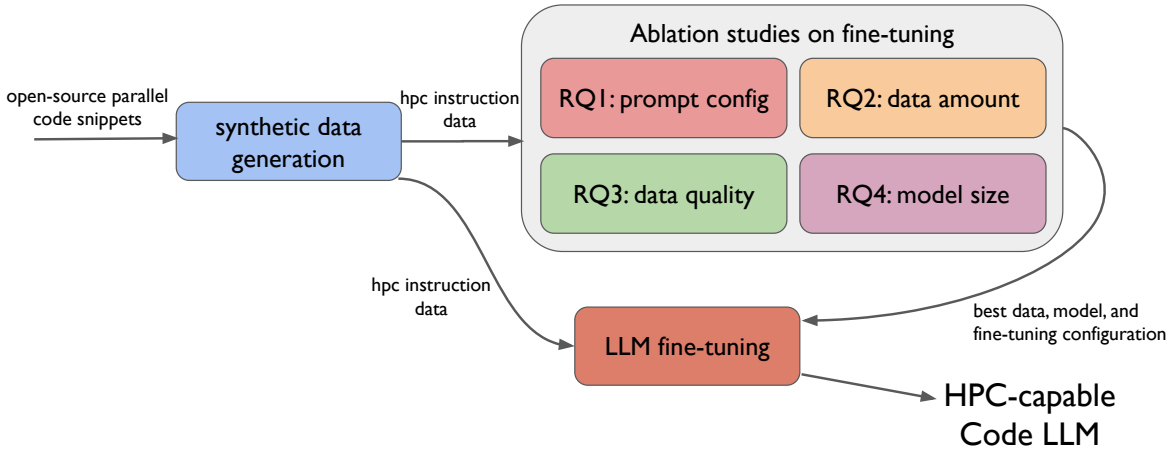


Figure 8.1: Overview of the methodology proposed in this paper. First, we use open-source parallel code snippets to generate a large synthetic instruction dataset of parallel code samples. We then conduct ablation studies to understand how data, model, and fine-tuning parameters impact the capability of a code LLM to write parallel code. Finally, we utilize the dataset and insights from the ablation studies to fine-tune a code LLM for parallel code generation and evaluate it against other code LLMs on the parallel code generation benchmark ParEval.

We begin by generating a large scale synthetic dataset of code samples using open-source

parallel code snippets and state-of-the-art LLMs. This dataset is comprised of roughly 120k parallel code instruction-response pairs where the instruction is a natural language problem description and the response is the code that solves the problem. The construction of this dataset is inspired by previous work [141] that demonstrated the success of fine-tuning smaller code LLMs on synthetic data generated from larger foundation models.

Using the HPC instruction dataset, we then conduct an in-depth study along the axes of code model fine-tuning to better understand how data representation and quality, model size, and prompt construction impact the ability of a code LLM to learn how to generate parallel code. Each of these ablation studies explores a research question raised in Section 8.1. During these studies we evaluate each of the fine-tuned models against the ParEval [102] benchmark to understand their performance on real parallel code generation tasks. These studies yield critical insights into best practices for fine-tuning HPC code LLMs.

Finally, with the full HPC instruction dataset and insights from the ablation studies, we fine-tune three state-of-the-art HPC capable code LLMs. These are evaluated against the ParEval benchmark and compared to other state-of-the-art LLMs for their ability to generate parallel code.

8.3 Generating Synthetic Data for Studying Axes of Parallel Code Modeling

Before we can fine-tune HPC LLMs, we need to collect a large dataset of HPC relevant code and dialog. While large datasets of open-source code exist [91], previous work has shown that generating structured synthetic data with state-of-the-art LLMs can yield data much more effective for fine-tuning specialized code LLMs [141]. This section details our approach to collecting large-scale synthetic data for HPC based on this insight.

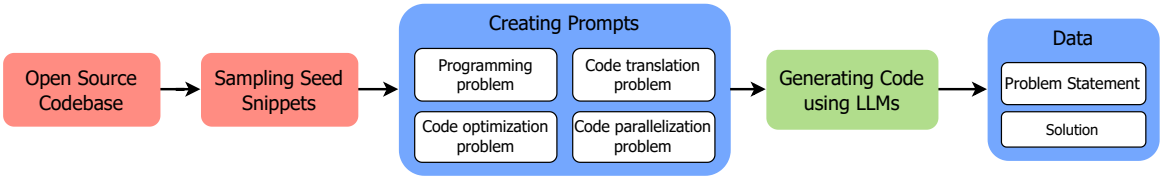


Figure 8.2: Synthetic data generation process. We collect seed snippets from open source codebases and combine them with multiple prompt templates to create data generation prompts for an LLM. These prompts are then used to generate problem-solution pairs with an LLM.

While state-of-the-art commercial LLMs like GPT-4o can generate high-quality instruction samples, they tend to generate very repetitive samples. To address this, we adapt the use of seed code snippets from [141] to get diverse outputs from the LLM. We gather a diverse set of seed snippets from open-source codebases in The Stack V2 [91], focusing on code in HPC languages (C, Fortran, etc.) and using HPC libraries (MPI, OpenMP, etc.). In total we collect 125k seed snippets including 25,000 samples in Python, C, FORTRAN, and C++, 15,000 samples in CUDA, and 5,000 samples in Chapel and OpenCL. When asked to generate a data sample, the LLM is asked to be inspired by the seed snippet, yielding more diverse and creative outputs. This process is visualized in Figure 8.2. An example programming template response can be seen in Figure 8.3, illustrating the workflow from seed snippet selection to the final dataset sample.

We obtain further variety in the generated data by generating multiple sample types:

Programming Prompts: In this template, the LLM is tasked with generating a parallel programming problem and a corresponding solution.

Translation Prompts: The translation template directs the LLM to create a problem focused on converting code from one parallel programming language to another. For example, the model might be prompted to translate a CUDA-based implementation into OpenMP or OpenMP to MPI.

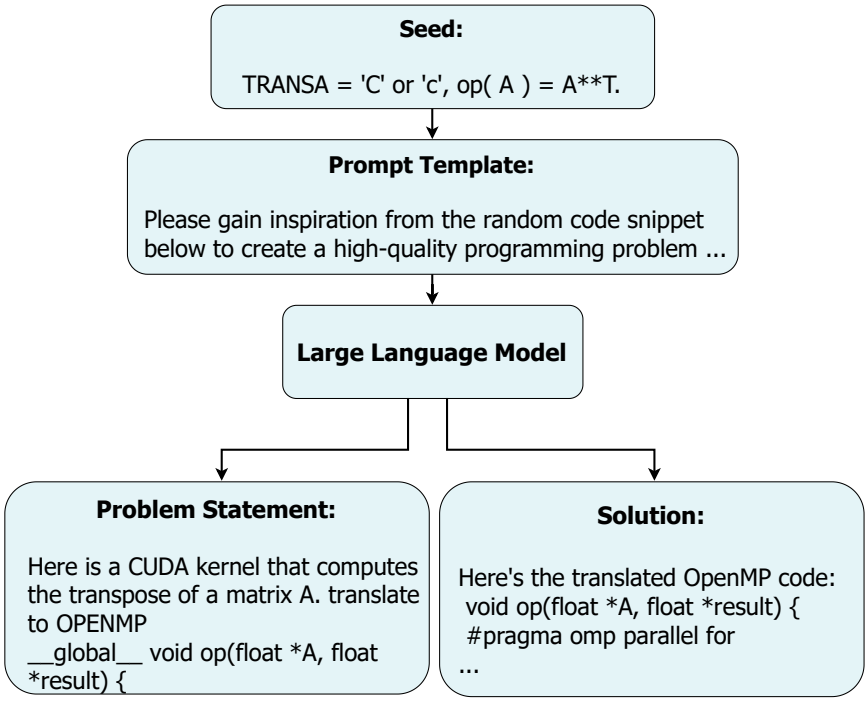


Figure 8.3: Example synthetic data generation output. Here, a random seed snippet is used alongside the translation prompt template and fed into the LLM. The resulting synthetic sample from the LLM is a problem of translating some code to OpenMP and the corresponding solution.

Optimization Prompts: For these prompts, we ask the LLM to generate an optimization problem and a corresponding optimized solution.

Parallelization Prompts: The parallelization template asks the LLM to parallelize a given code snippet, transforming it from a sequential implementation to an efficient parallel version.

Using the 125k formatted prompts we generate synthetic data samples with four state-of-the-art LLMs: Gemini-Pro, DBRX, Llama-3-70B, and Mixtral-8x7B. The resulting dataset, named PARALLEL-INSTRUCT, comprises over 122k synthetic data samples (some outputs were not parsable and discarded). We use several LLMs to gather a variety of samples, further ensuring data diversity. It also enables us to study the impact of data quality along the axis of source

generation model.

8.4 Ablation Studies Exploring the Impact of Data, Model, and Fine-tuning Parameters

We now have a large dataset of synthetic instruction HPC data, PARALLEL-INSTRUCT, and our goal is to fine-tuning existing models with this data. However, there are many unknowns regarding the configuration of fine-tuning such as how to format prompts, how much and what quality of data to use, what size of model to use, etc. In this section, we design a series of ablation studies along different axes of model fine-tuning to better understand how each contributes to the ability of a fine-tuned code LLM to generate parallel code.

8.4.1 Choice of Base Model and Instruction Masking

In this experiment, we explore the choice of base versus instruct models, and whether to use instruction masking with the goal of answering RQ1. We choose to study the impact of base versus instruct models as it is unclear from related literature which model type is better for fine-tuning on specific tasks. Generally, most users interact with instruct models as they are able to follow instructions and better engage in dialog-like interactions. For this reason, most open-source models have instruct models available that have been fine-tuned from a base model. When fine-tuning a new instruct model, on one hand, it may be better to reap the benefits of the existing fine-tuned model and start from there. On the other hand, it may be better to start from scratch with a base model, since they will be more general and easier to fine-tune. Instruction masking is usually employed to prevent the model from learning bad patterns that may be present in the user

instruction. We only want to learn from the responses. While intuitive, we are actually learning from less information when we mask the instruction and it is unclear if this trade-off between learning from less information and learning from less noise is worth it.

We fine-tune the Deepseek-Coder 1.3B and 6.7B base and instruct models with and without instruction masking on the PARALLEL-INSTRUCT, Magicoder-OSS-Instruct-75K, and Evol-Instruct-Code-80k-v1 datasets. In total, we fine-tune eight models: $\{1.3B, 6.7B\} \times \{base, instruct\} \times \{masked, unmasked\}$. We omit the 16B model from this experiment due to its high computational cost for fine-tuning. The goal of this experiment is to better understand the impact of the choice of base model and instruction masking.

8.4.2 Studying the Impact of the Amount and Quality of Parallel Code Data

Even with an ideal base model and prompting strategy it is still difficult to fine-tune a good model without the right amount and quality of data. To answer RQ2 and RQ3, we design two experiments: one to study the impact of the amount of data from individual parallel models and another to study the impact of the quality of data.

For the first experiment, we create several versions of the PARALLEL-INSTRUCT each with varying amounts of MPI code samples: 0k, 2k, 4k, 6k, 8k, 10k, and 12k. We leave the other data in the dataset unchanged and just vary the amount of MPI data. MPI samples are identified by the presence of certain substrings like "mpi.h" or "MPI_Init" in the code. These datasets are used to fine-tune the 1.3B and 6.7B models resulting in 14 total models. We omit the 16B model from this experiment due to its high computational cost for fine-tuning. The purpose of this study is to shed light on how the amount of data from a specific parallel model affects the final performance

of the LLM on that parallel model. Does performance keep increasing with more data or does it plateau at some point? This is important as it informs how we collect future data for fine-tuning. We select MPI for this study as LLMs consistently perform worse at generating MPI code than any other parallel programming model [102] and, therefore, it is desirable to improve their ability to generate MPI code.

Tangentially, we also study the impact of the quality of data on the fine-tuned models. As LLMs are increasingly getting more dependent on synthetic data for training, it is also getting extremely important to validate the quality of the synthetic data being produced to see its effect on model performance. We hypothesize that there is a trade-off between the amount of data and the quality of data, where eventually more data stops improving performance and quality becomes more important. Understanding this trade-off is particularly vital for synthetic data where we are expending compute to create the data; we need to know whether compute time is better spent on more data or better data.

Directly studying data quality is difficult as it is hard to quantify and the scale of data is too large for qualitative analysis. In order to overcome this we instead use the base model used for generating the synthetic data as a proxy for differences in data quality. We presume that the different models generate data of different quality. This will not allow us to infer what makes the data better or worse, but it will allow us to see if quality impacts the ability of the fine-tuned model to generate parallel code. To conduct this experiment we fine-tune the 1.3B and 6.7B models on the PARALLEL-INSTRUCT dataset generated from four different LLMs: Gemini-Pro, DBRX, Llama-3-70B, and Mixtral-8x7B. We also fine-tune both models on all of the data together. Again, we omit the 16B model from this experiment due to its high computational cost for fine-tuning. This results in ten total models that we can compare to see if the quality of the

data impacts the final performance of the fine-tuned model.

8.4.3 Studying the Impact of Model Size

Finally, we aim to study how model size impacts the final performance of a fine-tuned model (RQ4). While larger models tend to be better at most tasks, there is a trade-off where the time and resources necessary to run a larger model may not be worth the marginal increase in performance. For example, if a 7B parameter model is able to generate code for a particular niche task *nearly* as well as a 70B parameter model, then it is likely much more practical for a user to simply use the 7B model. It will run quickly on a consumer laptop whereas the 70B model will require specialized hosting or multiple GPUs. To study the impact of model size, we fine-tune the 1.3B, 6.7B, and 16B models on the PARALLEL-INSTRUCT dataset. This will allow us to compare the performance of the models across different sizes and see if the larger models are worth the extra resources.

8.5 LLM Fine-tuning Setup

We use what we learn from conducting the experiments described in the previous section to fine-tune the final versions of the fine-tuned Deepseek-Coder of different sizes.

8.5.1 Selecting a Pre-trained Model

We have to select a pre-trained model to fine-tune before starting to fine-tune. When fine-tuning smaller open-source models, choosing a model already trained for code tasks tends to yield better results [23]. Based on this and the successful results of previous code LLM fine-

tuning studies [141], we select the DeepSeek-Coder [57, 40] family of models for fine-tuning. In particular, we fine-tuned the 1.3b, 6.7b [57], and 16b [40] parameter models. These models are state-of-the-art in code modeling and outperform other LLMs on many coding benchmarks [23, 57, 40]. They are trained on a dataset of 87% code and 13% natural language with a 16k context window. The 1.3b and 6.7b are based on the llama [134] model architecture, while the 16b is a custom mixture-of-experts (MOE) [14] architecture. The MOE architecture enables the 16b model to scale to larger sizes while maintaining faster runtime performance.

8.5.2 Fine-Tuning on Synthetic HPC Code Data

We fine-tune each of the models on the PARALLEL-INSTRUCT, Magicoder-OSS-Instruct-75K [141], and Evol-Instruct-Code-80k-v1 [92] datasets. The latter two datasets are state-of-the-art synthetic and semi-synthetic code instruction datasets. We include these since, although they are not HPC specific, they can still improve the model's generalization capabilities. In total the fine-tuning dataset has 277k samples.

For generating the best 1.3b, 6.7b, and 16b fine-tuned models, we use the findings of the ablation studies presented in Section 8.7. The ablation studies are not exhaustive, hence analyze them to decide the best configuration setup.

8.6 Experimental Setup and Evaluation

In this section, we detail the fine-tuning step, other models used for comparison, and the benchmarks and metrics used to compare models for parallel code generation.

8.6.1 Fine-tuning Setup

We use the AxoNN [126] framework to fine-tune the models. This is a parallel deep learning framework wrapped around PyTorch [113]. It handles automatically parallelizing the model across GPUs and allows us to fine-tune the models that do not fit in memory on a single node. The 6.7b and 16b models are fine-tuned on four nodes each with four 80GB A100 GPUs, while the 1.3b model is fine-tuned on two A100 GPUs. The total fine-tuning times range between 3 and 20 hours.

We fine-tune the 1.3b and 6.7b models in bfloat16 precision with a batch size of 128 and a sequence length of 8192 for two epochs. The 16b model is fine-tuned with a batch size of 1024 for one epoch. Furthermore, we employ the AdamW optimizer [90] to update the model weights based on the fine-tuning loss. This training setup and hyperparameters are selected based on those used in related literature to fine-tune code LLMs. Cursory experiments showed that these hyperparameters work well for our fine-tuning task, however, it is possible that an exhaustive search could yield better results. Performance hyperparameters, like batch size, are selected based on the model size, available GPU memory, and desired performance. The context window length is lowered from 16k to 8k from the base models, since none of the data samples in the dataset exceed 8k tokens and this saves memory and performance during fine-tuning.

8.6.2 Other Models Used for Evaluation

We compare our final models with several other state-of-the-art code LLMs to better understand their performance and how our study's insights can lead to improvements in the field. We compare our models with the following models:

- **StarCoder2 (1.3B, 7B, 15B)**: LLMs pre-trained on a large corpus of mostly code data from The Stack V2 [91].
- **Magicoder (6.7B)**: A fine-tuning of the DeepseekCoder-6.7B model fine-tuned on synthetic data generated based on open-source code [141].
- **Phind-V2 (34B)**: A fine-tuning of the CodeLlama-34B [121] model on a proprietary dataset [115]. At the time of its release it was the best model on the BigCode leaderboard [23].
- **Gemini-1.5-flash**: A commercial model avaiable via API from Google [132].
- **GPT-3.5, GPT-4**: State-of-the-art commercial LLMs from OpenAI only accessible via API [27, 107].

8.6.3 Benchmark Used

When evaluating LLMs for code generation it is imperative to evaluate them on code correctness. To do this for parallel code generation we use the state-of-the-art benchmark ParEval [102]. ParEval has 420 coding problems that it uses to test an LLM's parallel code generation capabilities. These problems range across 12 different problem types: sort, scan, dense linear algebra, sparse linear algebra, search, reduce, histogram, stencil, graph, geometry, fourier transform and transform help us show the diversity on which the model has been tested on. For each of the problem types there are problems across seven different execution models: mpi, mpi+omp, cuda, kokkos, serial, hip, omp. ParEval provides drivers to run and unit test the generated code for correctness. Furthermore, the results can be analyzed along the many different axes of the problem types and execution models.

We also compared our model's memory requirements and throughput with other models to better understand the trade-offs between model size, performance and accuracy. These numbers are recorded on the ParEval benchmark when generating outputs using an H100 and a batch size of one. These results are important to users who may be constrained by hardware with limited memory or speed.

8.6.4 Metrics for Comparison

Since LLMs are probabilistic and may output different results for the same problem it is generally best to evaluate them in a probabilistic manner. For code LLMs most papers have adopted the pass@k metric to do this [31]. This metric quantifies the probability that an LLM can generate at least one correct solution within k attempts. Since we cannot calculate this probability directly we need to estimate it. To do this for one prompt, N samples are generated where N is much greater than k, which are then evaluated on code correctness and used to estimate pass@k. Choosing N to be much greater than k ensures that we can compute a statistically significant estimate of pass@k. The pass@k compute is shown in Equation (8.1).

$$\text{pass@}k = \frac{1}{|P|} \sum_{p \in P} \left[1 - \binom{N - c_p}{k} \Big/ \binom{N}{k} \right] \tag{8.1}$$

Number of samples generated per prompt

Set of prompts

Number of correct samples for prompt p

To further demonstrate pass@k, say we want to generate a pass@1 score for a model, it will generate $N = 10$ samples for a given prompt and out of these $c_p = 3$ samples are correct. Using the formula, we will get a score of 0.3 so the model has a 30 percent chance of generating the

correct solution in it's first attempt. The pass@1 metric is an important benchmark that is used to evaluate models' usability which is why we use it to compare our model with other models to see where it stands. In recent years, papers have resorted to just reporting pass@k for $k = 1$ as LLMs have become more powerful and can generate correct code more often. It is also a more desirable metric for the user who wants code to be generated correctly the first time.

8.7 Ablation Study Results

With the different models trained across the various configurations and data partitions, we can now analyze each model's parallel code generation performance to better understand the impact of different training configurations. In this section we detail the results from each of these ablation studies and provide insights into how to best train an HPC specialized code LLM.

8.7.1 Choice of Base Model and Instruction Masking

Figure 8.4 details the parallel code generation results on ParEval for the masked/unmasked and instruct/non-instruct prompt formats. There are eight models shown in the figure; they were fine-tuned on the Deepseek-Coder base models and the Deepseek-Coder instruct models using either masked or unmasked gradients. **We observe little correlation between using masked and unmasked gradients on the instruction prompts**. Using masked gradients instead of unmasked provides a slight less than one percentage point improvement for the 1.3B models. However, using masked gradients hurt performance when fine-tuning the 6.7B model. This goes against traditional wisdom that using masked gradients is better for fine-tuning instruction models.

Unlike for masking, there is a notable difference between fine-tuning the base version of a

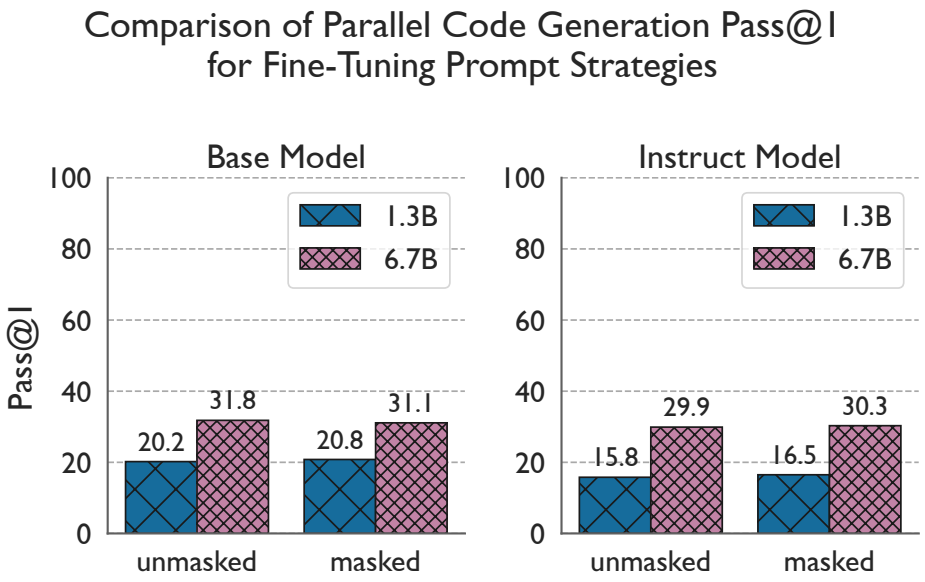


Figure 8.4: ParEval parallel code generation scores for various prompt formats. Results are shown for 8 total model configurations: {masked, unmasked} gradients × {instruct, non-instruct} base models × {1.3B, 6.7B} model sizes. There is no correlation in parallel code generation performance between masked and unmasked gradients, however, fine-tuning the base model rather than the instruct gives much better results for both 1.3B and 6.7B models.

model and an existing instruct variant. **We observe that fine-tuning base models, rather than instruct variants, leads to better performance at parallel code generation.** This is true across all configurations: 1.3B and 6.7B models, masked and unmasked gradients. The difference is most pronounced for the 1.3B models, where fine-tuning the base models gives a roughly 4 percentage point advantage over fine-tuning the instruct models. While it is difficult to pinpoint the exact cause of this difference, it is likely that the instruct models were fine-tuned to model a less general distribution when they were first fine-tuned from the base model. In other words, it is better to fine-tune base models and not further derivations (fine-tunings) of them, since the base models are more general and can be fine-tuned to a specific task more effectively.

8.7.2 Studying the Impact of the Amount and Quality of Parallel Code Data

Figure 8.5 presents the MPI code generation performance for various amounts of MPI fine-tuning data. MPI is selected for this study since LLMs consistently perform worse at generating MPI code than any other parallel execution model [102] and, therefore, it is desirable to improve their ability to generate MPI code. In total there are 14 models shown in the figure: the 1.3B and 6.7B Deepseek-Coder models each fine-tuned on datasets with 0k, 2k, 4k, 6k, 8k, 10k, and 12k MPI samples. After running ParEval's MPI benchmarks on these models, **we observe that increasing the amount of training data for a particular parallel execution model can improve the performance of smaller code LLMs on that execution model with diminishing returns, but has little to no effect on larger code models.**

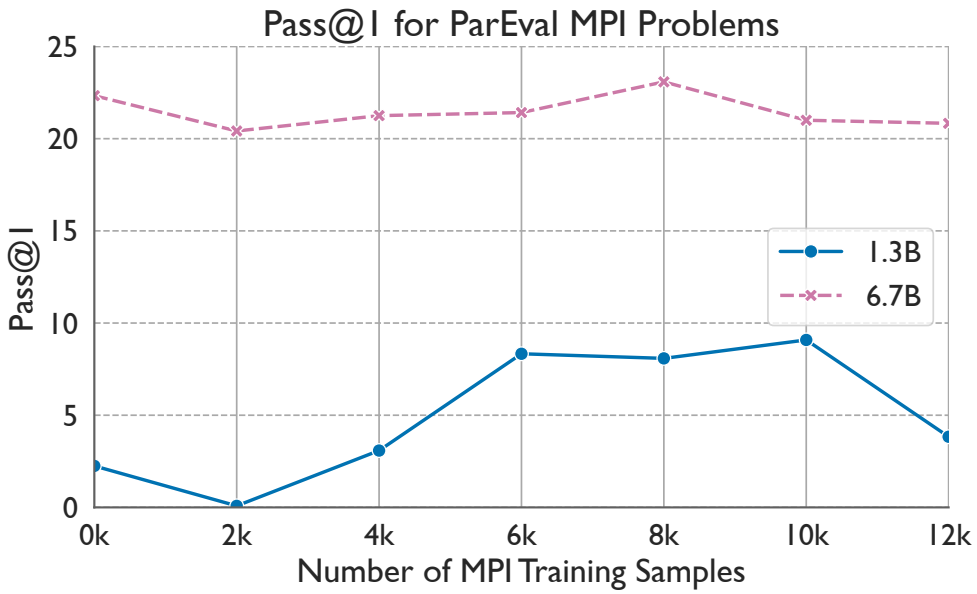


Figure 8.5: ParEval MPI code generation performance for increasing amounts of MPI fine-tuning date. As the amount of MPI fine-tuning date increases the smaller 1.3B model sees an increase in ability to generate MPI code with diminishing returns after 6k samples. The larger 6.7B model sees no improvement in MPI code generation performance with additional data.

The 1.3B models see a gradual increase in MPI code generation performance until 6k MPI

samples, after which the performance plateaus and eventually decreases at 12k MPI samples. The plateau can be explained by smaller models being more susceptible to overfitting. The 6.7B models, on the other hand, have fairly consistent MPI code generation performance across all amounts of MPI fine-tuning data. The model has already learned all it can from the data and adding more has no effect on performance.

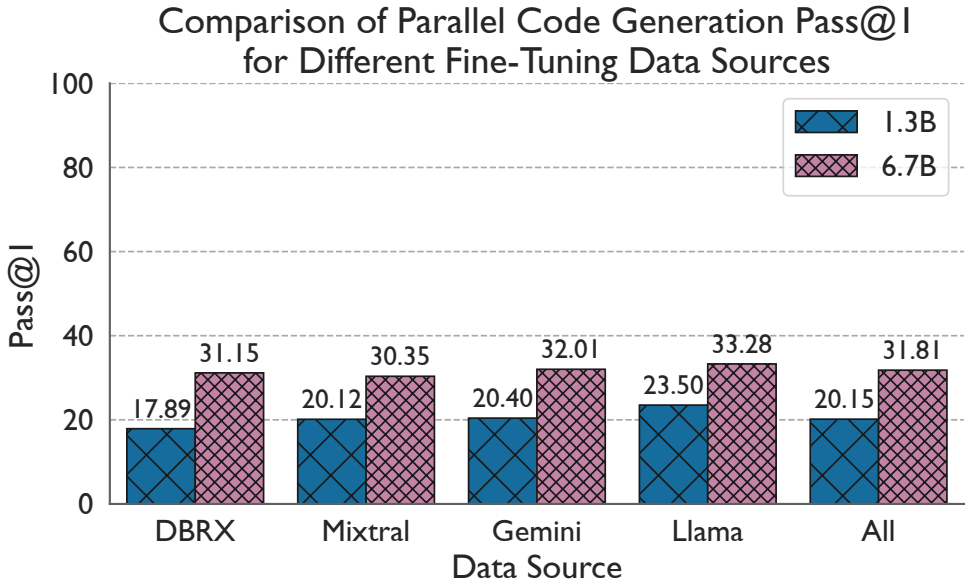


Figure 8.6: ParEval parallel code generation performance across different synthetic data sources. There is a clear difference in performance across data sources with Llama generated synthetic data leading to the best performing LLMs and DBRX leading to the worst.

In addition to the amount of data, the quality of the data can also impact the ability of an LLM to learn from it. To study this, we examine the performance of the models when fine-tuned on PARALLEL-INSTRUCT synthetic data with different LLMs used to generate the data. Figure 8.6 shows the ParEval performance of each of these models. **We observe that the quality of the parallel code fine-tuning data can have a significant impact on the performance of a code LLM on parallel code generation.** Models trained on Llama3-70B generated data have up to six percentage points higher parallel code generation performance than those trained on

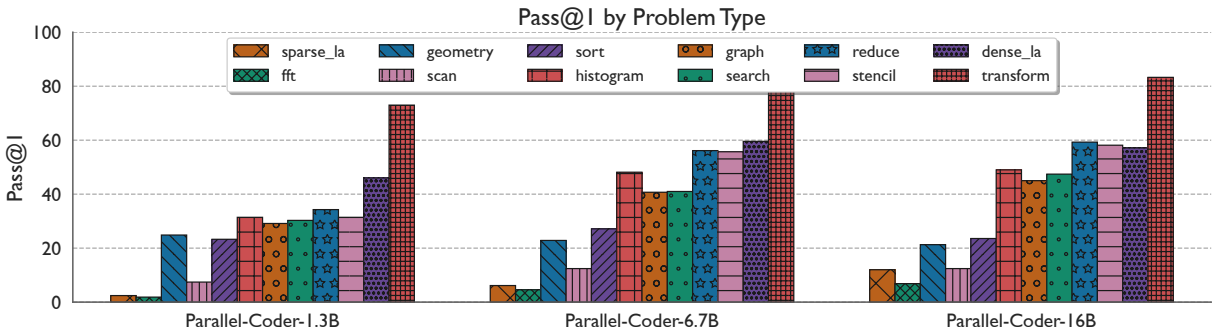


Figure 8.7: ParEval code generation performance by problem type. These results follow similar trends to those shown in [102] except with higher performance across all problem types.

DBRX data. While it is difficult quantify the quality of these data samples, it is clear that the quality of the data does lead to a measurable difference in generation quality. This motivates further investigation into what makes a training data sample high quality.

8.7.3 Studying the Impact of Model Size

Finally, we investigate the impact of base model size when fine-tuning a code LLM. This is a crucial question as larger models are considerably more expensive to fine-tune, store, and deploy for inference. Understanding the trade-offs between size and generative capabilities is essential for designing practical code LLMs. Figure 8.8 shows the ParEval performance of the 1.3B, 6.7B, and 16B models fine-tuned on the same PARALLEL-INSTRUCT data. **We observe a significant increase in performance from 1.3B to 6.7B, but a much smaller increase from 6.7B to 16B.**

The diminishing return as model size increases is expected as we are using knowledge distillation to train the models; the performance of the LLMs is unlikely to surpass the performance of the teacher model. Based on the ParEval results in [102], the 16B model is approaching the parallel code generation performance of foundation models like GPT-3.5 and GPT-4.

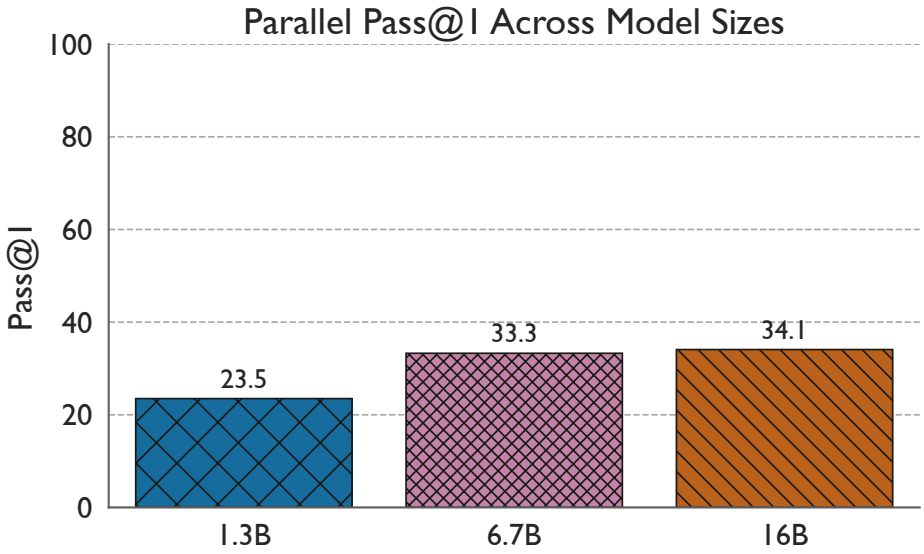


Figure 8.8: ParEval serial and parallel code generation performance along various base model sizes. There is a significant increase in performance from 1.3B to 6.7B, but a much smaller increase from 6.7B to 16B.

8.8 An Improved Parallel Code LLM Based on Ablation Study Results

Using the insights from the ablation studies we train a series of models with the best configuration to create state-of-the-art parallel code generation LLMs. In this section we evaluate these models, Parallel-Coder-1.3B, Parallel-Coder-6.7B, and Parallel-Coder-16B, on the ParEval benchmark suite and compare their performance with other state-of-the-art code LLMs.

8.8.1 Parallel-Coder Across Problem Types and Execution Models

Figure 8.7 shows the code generation performance of Parallel-Coder across the twelve problem types in the ParEval benchmark suite. We observe similar trends to those shown in [102] except with higher performance across all problem types. The LLMs tend to struggle with sparse unstructured problems, such as *sparse linear algebra* and *geometric* problems. The models per-

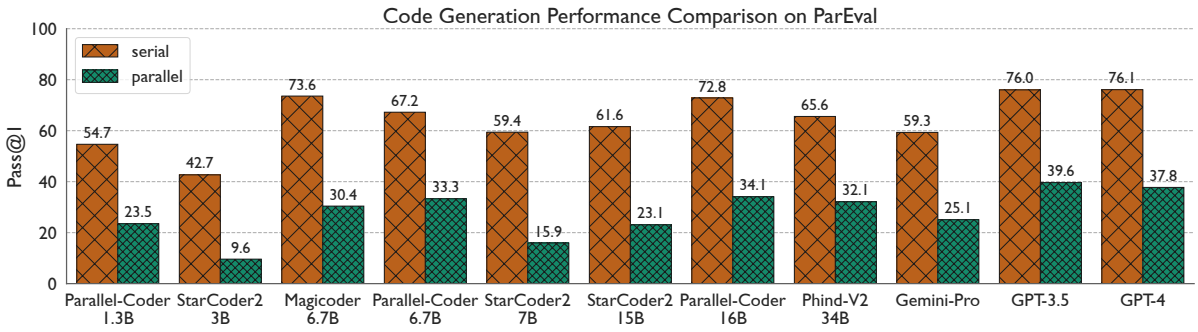


Figure 8.9: Comparison of ParEval parallel and serial code generation performance across all models. The Parallel-Coder models perform as well or better than other models of similar size.

form much better on dense, structured problems such as *dense linear algebra*, *stencil*, and simple *data transformation* problems. With the exception of *geometric* problems, the models perform better as their size increases with the 16B model performing the best across all problem types. Interestingly, the models perform worse on *geometric* problems as the model size increases.

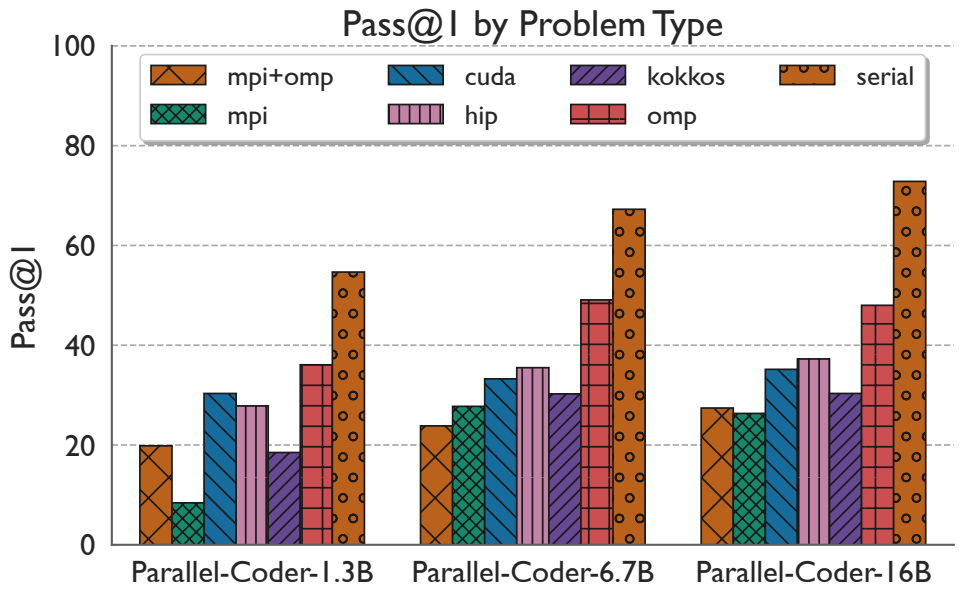


Figure 8.10: ParEval code generation performance by execution model. The LLMs perform best on serial code followed by OpenMP. The models struggle most with MPI code generation.

Another axis of comparison besides problem type is the parallel execution model. Figure 8.10 shows the code generation performance of the three LLMs across the seven execution

models in ParEval. As with the problem types we see similar trends as in [102]. The LLMs always perform best on *serial* code followed by *OpenMP*. This is expected as OpenMP code is most similar to its serial counterpart. The next best performing execution models are the GPU models, *CUDA* and *HIP*. These are followed by *Kokkos* and the MPI models, *MPI* and *MPI+OpenMP*, reinforcing the trend that LLMs struggle with MPI code generation.

8.8.2 Comparison with Other Models

Finally, we compare the performance of the Parallel-Coder models with other state-of-the-art code LLMs. Figure 8.9 shows ParEval parallel and serial code generation performance across all models. We see that, while the commercial models still dominate, the Parallel-Coder models are competitive. At each relative model size class we see that the Parallel-Coder models perform better than comparative models for parallel code generation. The Parallel-Coder-1.3B is significantly better than StarCoder2-3B despite being much smaller. Furthermore, the Parallel-Coder-6.7B model performs better than the 34B Phind-V2 model. Despite their success at parallel code generation, the Parallel-Coder models are still beaten by Magicoder-6.7B for serial code. This highlights, however, the success of our data and fine-tuning strategies at training models to generate parallel code.

Although parallel code correctness is the most important metric for an HPC code LLM, the system requirements of the model and the speed at which it can generate code are also very important to developers. A model that can generate correct code nearly as often as a larger model, but can run quickly on a consumer laptop, is arguably much more useful for developers than the larger model. To study this trade-off in the Parallel-Coder models, we present the throughput,

required memory, and ParEval parallel pass@1 results for each model in Figure 8.11. The size of the dots are scaled based on the memory requirement of the model with larger dots indicating larger models. The ideal location for a model is the top right where the model generates correct code quickly.

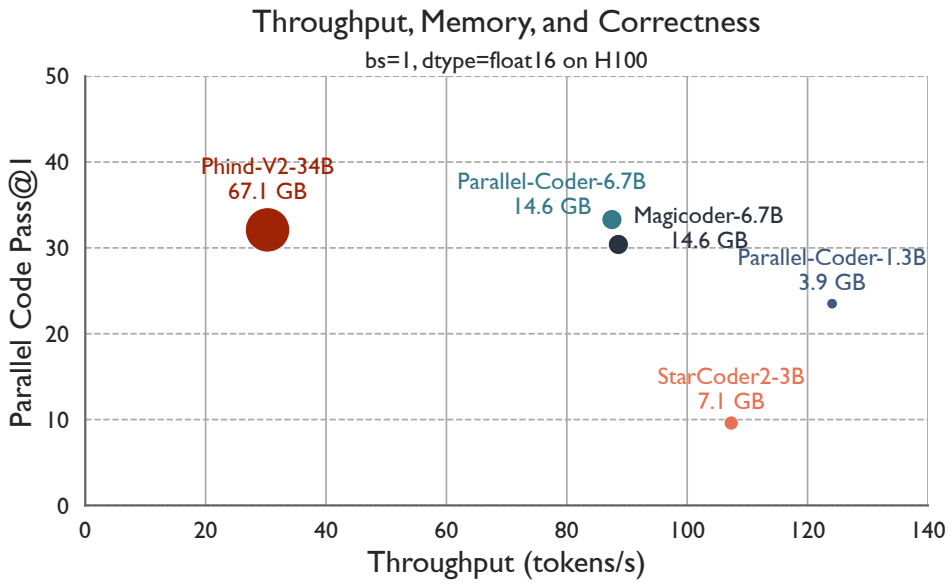


Figure 8.11: Comparison of parallel code generation pass rate (pass@1), model memory requirements (GB), and generation throughput (tokens per second). The top right of the graph is the ideal location where models generation correct code quickly. The smaller the dot the lower the model memory requirements. We see that the 6.7B model gets similar performance to the much larger 34B model while generating tokens significantly faster.

We see that the Parallel-Coder models generate parallel code just as well or better than the other models while being faster and more memory efficient. Parallel-Coder-6.7B is significantly faster than Phind-V2-34B while requiring much less memory and having slightly better performance on ParEval. Magicoder-6.7B has similar throughput and memory requirements as Parallel-Coder-6.7B, but performs worse at generating parallel code. The Parallel-Coder-1.3B model is the fastest and requires the least amount of memory, yet it outperforms other models in its size class (StarCoder2-3B). These results demonstrate that with high quality fine-tuning data

we do not need to sacrifice memory and throughput to generate high quality parallel code.

Chapter 9: Improving the Performance of LLM Generated Code using Reinforcement Learning

9.1 Motivation

Developing fast and scalable code is a difficult, but often necessary task for scientific software developers. It can require expert knowledge of the application domain, algorithm design, programming languages, and hardware. This is a challenging task for even serial code, and even more complex for parallel code. Further, programmers and performance engineers are often tasked with optimizing existing code, often not written by them, which requires understanding an existing codebase and the performance implications of changes. Large language models (LLMs) have emerged as a powerful tool for assisting in the software development process for a variety of tasks such as code completion [31], bug detection [119, 73], and code summarization [5, 59, 56, 6]. Recently, they have also been used with limited success to generate parallel code [103]. Yet they struggle to understand performance aspects of code because they were not designed for this task. Code LLMs are trained on just code as text, and as a result, are not well-suited to reason about complex performance issues. Additionally, the code they generate does not consider performance and could be slow, despite being correct. This has been demonstrated in existing works that show LLMs often generate inefficient parallel code [102, 136].

Creating artificial intelligence (AI) models that can generate faster code has the potential to significantly improve the productivity of software developers. By using performance-aware code LLMs, developers can focus on design and correctness without worrying about the performance implications of using LLMs to generate code. Additionally, as LLM-based tools become more integrated with software development workflows, developers will become more and more reliant on the quality of their outputs. Improving the performance of LLM generated code while maintaining its correctness will improve the quality of the target software being developed. Further, code LLMs that can write fast code can remove the need for every scientific and parallel programmer to be a performance expert in addition to their existing domain expertise.

It is non-trivial to create code LLMs that can generate faster code. Since creating performance-aware code LLMs will require fine-tuning of LLMs using performance data, one challenge is creating such datasets. LLMs typically require very large, general datasets for training tasks, and it is challenging to create such large datasets for performance data. Arbitrary code can have a wide range of performance characteristics, and depend on many factors such as input data, hardware, and software environment. Due to the complexity in collecting performance data for arbitrary code, performance datasets are often small and/or narrow in focus. Further, even with such a dataset in hand, an LLM needs to be carefully fine-tuned to *align* its generated outputs with more performant code. There are many potential pitfalls here, for instance, improving the performance of generated code at the cost of correctness. Additionally, fine-tuned LLMs can learn a distribution too disjoint from their initial code distribution they modeled and lose their ability to generalize.

In order to overcome the challenges associated with collecting large scale performance data, we propose a new approach that combines a structured, narrow performance dataset with

a more general synthetic code dataset for fine-tuning. We also propose two novel fine-tuning methodologies: (1) reinforcement learning with performance feedback (RLPF), which is based on reinforcement learning with human feedback (RLHF) [112], and direct performance alignment (DPA), which is based on direct performance optimization (DPO) [117]. We use these two approaches and the new dataset to align an existing code LLM to generate faster code. These proposed fine-tuning methodologies use fast and slow code pairs to fine-tune the LLMs to generate samples more similar to the fast code and less similar to the slow code. The aligned model is then evaluated on two code generation benchmarks and one code optimization benchmark. We find that the aligned model is able to generate code with higher expected speedups than that of the original model, while maintaining correctness.

This work makes the following important contributions:

- A code performance dataset that combines narrow, structured performance data with broad synthetic data to help models learn performance properties, but maintain their ability to generalize.
- Two novel fine-tuning methodologies, reinforcement learning with performance feedback (RLPF) and direct performance alignment (DPA), for aligning code LLMs to generate faster code.
- A fine-tuned, performance-aligned LLM that generates faster code than traditional code LLMs.
- A detailed study of the performance and correctness of the code generated by performance-aligned LLMs including serial, OpenMP, and MPI code. Additionally, an ablation study motivating the use of synthetic data to fine-tune code LLMs for performance.

9.2 Overview of Methodology

Figure 9.1 presents an overview of our methodology for aligning code large language models (LLMs) to generate faster code. We start by creating a dataset that can be used to fine-tune an LLM to generate code that is both correct and fast (Section 9.3). To accomplish this, we collect a large, structured code dataset with performance data and test cases to measure correctness. This structured dataset is, however, not representative of the entire distribution of code we want an LLM to optimize so we ameliorate its shortcomings by using LLMs to generate a *synthetic* code dataset that covers a wider range of code.

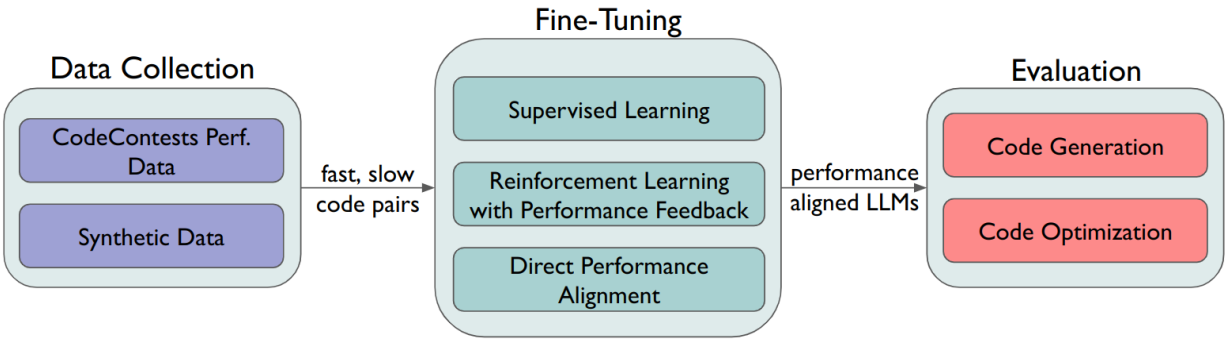


Figure 9.1: An overview of the proposed methodology. We first collect a large dataset of fast and slow code pairs using coding contest submissions and synthetically generated data. Then we fine-tune three different LLMs on this data to generate faster code. Finally, we evaluate the fine-tuned models on code generation and optimization tasks.

These datasets are then used to *align* the outputs of an LLM with performance considerations. We employ three different techniques – supervised learning, reinforcement learning, and direct alignment, to fine-tune code LLMs (Section 9.4). The models are aligned to answers that are not only correct, but also fast. Using the fine-tuned models we then generate code for a set of three different benchmark tasks for code generation and optimization (Section 9.5). These tasks measure the correctness and performance of the generated code for coding problems within and

outside the distribution of the training data.

9.3 Data Collection and Labeling

In order to align LLMs to generate more performant output, we need to fine-tune them on performance data. Further, to apply the proposed fine-tuning methods, we need a dataset of code where we have a slow and a fast implementation of a particular problem. This type of structured performance data paired with source code is difficult to collect. It requires being able to build, execute, validate, and profile arbitrary code snippets, which is difficult to accomplish at scale. In this section, we describe our process of collecting a large performance dataset ($\mathcal{D}_c$). Additionally, we discuss how we extend the dataset with synthetic data ($\mathcal{D}_s$) to cover a wider distribution of code patterns. The final dataset $\mathcal{D}$ contains over 4.5 million code samples, distributed over three source languages (C++, Java, and Python) as shown in Table 9.1.

Table 9.1: The number of samples in both datasets distributed by source language.

Dataset ($\mathcal{D}$)	Runtime Data	C++	Java	Python	No. of Samples
CodeContests+Perf ($\mathcal{D}_c$)	✓	1.8M	0.9M	1.8M	**4.5M**
Synthetic ($\mathcal{D}_s$)	✗	5k	0	5k	**10k**

9.3.1 Performance Dataset Collection

We build our performance dataset using the CodeContests dataset introduced by DeepMind in [86]. This dataset contains coding contest problems and solutions from the Aizu [9], AtCoder [15], CodeChef [35], Codeforces [36], and HackerEarth [58] online competition platforms. In total there are 13,610 coding problems in the dataset. These range in difficulty from

simple to very difficult, and cover a wide range of topics such as graph algorithms, dynamic programming, and search. Each problem in the dataset has a corresponding set of submissions from users, labeled as correct or incorrect on the respective coding contest website. The number of submissions per problem ranges between tens and thousands. There are solutions in three different programming languages: C++, Java, and Python. Additionally, the dataset includes meta-data for the problem such as the problem statement, test cases, time limits, and memory limits.

This dataset is extremely valuable for our study as it provides a large amount of code samples along with the necessary tests to measure correctness and performance. More so, it contains many code samples that solve the same problem, but in different ways and with different runtimes. While many of the code contest websites record runtimes for submissions, the CodeContests dataset as provided by DeepMind does not include this information. We collect this data ourselves into a new dataset, *CodeContests-Perf* ($\mathcal{D}_c$), by executing each of the correct submissions and recording their runtimes. Each submission is run on all the test cases for its problem. Generally, there are between 5 and 20 test cases per problem. We create submission-runtime pairs using the average runtime over all the test cases. Each run is executed on a single core of an AMD EPYC 7763 CPU with a 2.45 GHz base frequency.

The final *CodeContests-Perf* dataset contains 4.5 million samples. The distribution of samples by source language is shown in Table 9.1. There were a small fraction of submissions labeled as correct in the CodeContests dataset that errored or failed the test cases when we ran them. These are omitted from the final dataset. We also include code submissions that were marked as incorrect in the original dataset, however, we do not run them. These will eventually be useful to prevent the model from generating fast, but incorrect code.

9.3.2 Synthetic Data Generation

The amount of data and the availability of easy testing in the *CodeContests-Perf* dataset makes it a crucial component of our study. However, the distribution of code represented in the dataset is significantly different than that of the code that is typically found in production code. Coding contests generally award participants based on time-to-submission leading to users writing messy and/or disorganized code to solve problems as quickly as possible. Further, the types of problems typically found in coding contests such as depth-first search and dynamic programming, while an important subset of problems, do not cover the full range of relevant computational problems that are found in production code, and in particular, in scientific computing.

To address the shortcomings of the *CodeContests-Perf* data, we generate an additional synthetic dataset $\mathcal{D}_s$ of fast and slow code samples. This is inspired by several recent works demonstrating the effectiveness of fine-tuning LLMs on synthetic data to improve performance on real tasks [141, 151, 51, 60, 19]. Gilardi et al. [51] even find that LLMs can outperform humans for many text annotation tasks. In our case of annotating code performance, real runtimes are the best annotation, but in the absence of runtime data, synthetic data is a promising candidate to obtaining labeled code performance data.

We use the Gemini-Pro-1.0 LLM model [132] to generate synthetic code samples as we found it to give the best outputs among a number of models we tested. We adapt the methodology in [141], where samples are generated using *seed* code snippets to get diverse outputs from the model. First, we create a dataset of 10,000 seed samples that are 1-15 line random substrings of random files from The Stack dataset [77], which is a large, 3TB dataset of permissively licensed code. Then the LLM is asked to generate three pieces of text: a problem statement inspired by

the seed snippet, a fast solution to the problem, and a slow solution to the problem. This produces inherently noisy data, since the LLM does not always generate correct or optimal (fast vs. slow) outputs. However, prior work has shown that the gain in predictive performance from fine-tuning on synthetic data often outweighs the downsides from noisy data [141].

In total, we collect 10,000 synthetic samples, 5,000 in C++ and 5,000 in Python. While adding more synthetic samples would likely continue to improve the quality of the fine-tuned model, we found that limiting to 10,000 samples provided adequate model quality while operating within time/cost constraints for this study. Table 9.1 shows the distribution of samples by language in the synthetic dataset.

9.4 Aligning LLMs to Generate Faster Code: Proposed Fine-Tuning Approaches

Large language models have been shown to be capable of generating correct code with high frequency on several benchmarks [31, 16, 107], yet they do not always generate code that is efficient [102]. They require further fine-tuning to align them with performance considerations. In this section, we detail how we fine-tune large language models with supervised learning and reinforcement learning techniques to generate faster code. We utilize the dataset introduced in Section 9.3 to train three different models using supervised learning, reinforcement learning with performance feedback, and direct performance alignment.

9.4.1 Supervised Learning

In the first approach, we fine-tune a language model on the dataset of code snippets from $\mathcal{D}$ to predict the next token in a sequence given previous tokens. For our methodology, we begin

with a model that has already been trained on a large corpus of text and code, and then fine-tune it on a smaller dataset of coding problems and fast solutions.

We create two types of prompts using the samples in $\mathcal{D}$ to fine-tune the model. In the first type of prompt, we use a standard instruction prompt where the model is given a problem statement and a fast solution (shown in Listing 3). Using the coding contest data in $\mathcal{D}_c$, we use the problem description as the instruction and randomly sample one of the five fastest solutions as the response. In the second type of prompt, we use a variation of the standard instruction prompt where the task is to optimize a given code snippet and the output is an optimized version of the code. For this, we use the problem description and one of the slowest 33% of solutions as the instruction, and one of the five fastest solutions as the response. Forming prompts from the synthetic dataset $\mathcal{D}_s$ is similar except we only have one slow and one fast solution for each problem, so we do not sample from ranges of solutions.

Over these prompts, the model is fine-tuned to minimize the cross-entropy loss between its predicted next token and the actual next token. We refer the reader to [100] for more details on fine-tuning LLMs for text generation. After fine-tuning, the model should have more fast code snippets in its training data and its probability distribution should shift toward faster code. Several prior works, however, have observed that methods more sophisticated than supervised fine-tuning are required to align LLM outputs with certain properties, such as safety and human preferences [112, 155, 17].

Supervised Fine-Tuning Evaluation Metric: We evaluate the success of the supervised fine-tuning by measuring the perplexity of the tuned model over an evaluation dataset. Perplexity is inversely proportional to how confident a model is that a data sample is in the distribution it

Listing 3 The instruction prompt format used to fine-tune the models. During fine-tuning, a coding problem is given to the model as an instruction-response pair, and the model is trained to generate similar responses when used for inference.

````
### Instruction:
Given a list of strings, find the longest
common prefix shared by all strings in the
list. The prefix should be the longest
possible string that is a prefix of every
string in the list.

### Response:
```python
def longest_common_prefix(strings):
  if not strings:
    return ""

  prefix = strings[0]
  for string in strings[1:]:
    while string.startswith(prefix):
      prefix = prefix[:-1]
  return prefix
```
````

models. A lower perplexity is better and indicates the LLM is less "perplexed" by a particular sample. A model's perplexity over t tokens from a dataset $\mathcal{X}$ is given by Equation (9.1).

$$\text{Perplexity}(\mathcal{X}) = \exp\left\{-\frac{1}{t}\sum_{i}^{t}\log \underbrace{p_\theta\left(x_i \mid x_{<i}\right)}_{\substack{\text{Predicted probability of token } x_i \\ \text{given the previous tokens } x_{<i}}}\right\} \tag{9.1}$$

9.4.2 Reinforcement Learning with Performance Feedback

To further align an LLM's outputs with performance considerations, we propose a new method, which we call reinforcement learning with performance feedback. This method is inspired by the success of reinforcement learning with human feedback (RLHF) [112], which aligns

LLM outputs with human preferences. RLHF uses human-labeled preference data to train a reward model that assigns rewards to LLM outputs that are more preferred by humans. This reward model is used in conjunction with reinforcement learning to fine-tune a LLM to generate outputs that are more preferred by humans. We adapt this method into reinforcement learning with performance feedback (RLPF) that uses performance feedback instead of human feedback to fine-tune LLMs to generate faster code.

Reward Model: We first need to design a reward function that can be used to guide the reinforcement learning process. If we can automatically run, test, and measure the performance of a generated LLM output, then we can simply use a function of the recorded runtime as the reward. In our case, this is possible for the coding contests dataset $\mathcal{D}_c$, where we have unit tests available to run and test the generated code (see Section 9.3.1). This further highlights the utility of this dataset for our study.

As mentioned in Section 9.3.2, we want to be capable of generating fast code outside the context of coding contests i.e. we do not want to exclusively use the code contests data for RL fine-tuning. Since we may not be able to obtain runtime data for other arbitrary code samples, we need to train a reward model that rewards faster code more than slower code for samples where we cannot obtain runtime data. Fine-tuning LLMs for relative performance modeling was previously demonstrated by Nichols et al. [103] and, thus, a fine-tuned LLM is a viable candidate for the reward model.

To accomplish this we train a reward model (an LLM), r_θ, to predict a reward for a given code sample, where a higher reward indicates faster code. To train this model, we first use a subset of $\mathcal{D}$ to create a dataset of triplets (p, d_f, d_s) where p is a problem description and d_f and

d_s are fast and slow code solutions to the problem, respectively. Using r_θ, we compute predicted rewards for d_f and d_s, and use these to calculate the loss function $\mathcal{L}_r$ in Equation (9.2).

predicted reward for fast code

predicted reward for slow code

$$\mathcal{L}_r = -\log\left[\sigma\left(r_\theta\left(p, d_f\right) - r_\theta\left(p, d_s\right) - \mu\left(p, d_f, d_s\right)\right)\right] \tag{9.2}$$

adaptive margin; scales the reward based on runtimes

This loss function is used to train r_θ to predict a higher reward for d_f than d_s. In Equation (9.2) σ, is the logistic function and μ is an adaptive margin as defined in Equation (9.3). The loss function in Equation (9.2) is adapted from Wang et al. [138] to include runtime information. It trains the reward model to generate rewards farther and farther apart for faster and slower code samples. As $r_\theta(p, d_f) - r_\theta(p, d_s)$ gets larger, the loss function tends towards zero. On the flip side, the loss increases as the difference between the rewards decreases or r_θ assigns a larger reward to the slower code. We utilize an adaptive margin μ to further scale the rewards based on how much faster the fast code is than the slow code:

max margin value

speedup of d_f over d_s

$$\mu\left(p, d_f, d_s\right) = \begin{cases} \min\left\{\lambda, \frac{\text{runtime}(d_s)}{\text{runtime}(d_f)}\right\} & \text{if } p \in \mathcal{D}_c \\ 0 & \text{otherwise} \end{cases} \tag{9.3}$$

Since we can train the reward model on both datasets $\mathcal{D}_c$ and $\mathcal{D}_s$, we can use the runtime information from $\mathcal{D}_c$ to scale the rewards appropriately. We use a max margin λ to prevent extremely large margins when d_s is very slow. Figure 9.2 provides an overview of the reward model fine-tuning process.

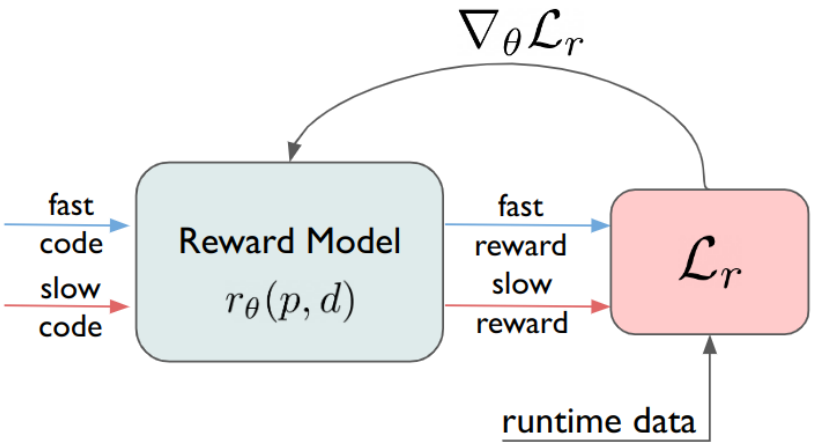


Figure 9.2: An overview of the reward model fine-tuning process. The reward model outputs a reward for a fast and slow code sample. The loss function uses these rewards alongside runtime data to update the weights of the model so that its predicted rewards move farther apart for faster and slower code scaled by the runtime speedup.

It is important to note that the reward model r_θ is not directly modeling code performance. Doing so would likely be impossible as performance can depend on a number of factors like hardware, input, etc. that are not accounted for in the input to the reward model. Instead, the reward model is trained to learn code structures and patterns that generally lead to better performance. This is another reason it is important to have a large dataset that covers a wide distribution of code, so that the model can learn these generalizations.

Using the runtime data in $\mathcal{D}_c$ and the trained reward model, we can define a reward function $r(p, d)$ that assigns a reward to an LLM generated code sample. This reward function is defined in Equation (9.4).

$$r(p,d) = \begin{cases} -1 & \text{if } p \in \mathcal{D}_c,\ d \text{ incorrect} \\ \frac{\text{median\_runtime}(p)}{\text{runtime}(d)} - 1 & \text{if } p \in \mathcal{D}_c,\ d \text{ correct} \\ r_\theta(p,d) & \text{otherwise} \end{cases} \tag{9.4}$$

The model is penalized with a negative reward if it generates incorrect code. If it generates correct code, then the reward is based on the speedup over the median runtime, $\text{median\_runtime}(d)$, from the submission already in the dataset. For the synthetic problems, we use the output of the reward model r_θ.

Reward Model Fine-Tuning Evaluation Metric: We can evaluate the fine-tuning of the reward model by computing its accuracy over an evaluation dataset. The accuracy here is defined as the proportion of samples where the reward signal is larger for the fast code than it is for the slow code.

$$\text{acc}_{\text{reward}}(\mathcal{X}) = \frac{1}{|\mathcal{X}|} \sum_{(p,d_f,d_s)\in\mathcal{X}} \mathbb{1}\left[r_\theta(p,d_f) > r_\theta(p,d_s)\right] \tag{9.5}$$

Here $\mathbb{1}$ is the indicator function that returns 1 if the condition is true and 0 otherwise. A perfect accuracy of 1 indicates that the reward model always predicts a higher reward signal for the fast code sample than the slow code sample.

Reinforcement Learning: Using the reward function $r(p,d)$ and Proximal Policy Optimization (PPO) [122], we can align an LLM to generate faster code. We use the supervised fine-tuned model from Section 9.4.1 as the base model to fine-tune with RL as is common in RLHF [112]. Following standard PPO training practices we optimize the base model using the reward objective function in Equation (9.6).

new model π^{RLPF}

being fine-tuned with RL

supervised model π^{S}

$$\mathcal{L}_p = r(p,d) - \eta \text{KL}\left(\pi^{\text{RLPF}}(d \mid p) \;\|\; \pi^{\text{S}}(d \mid p)\right) \tag{9.6}$$

Here KL is the Kullback-Leibler divergence and η is a hyperparameter that controls the divergence penalty. This penalty helps prevent the model from getting stuck in local optima or diverging too far from the original distribution of the supervised model [69, 80].

During fine-tuning, a prompt is given to the base model (a coding problem or optimization task) and is used to generate a response. The reward function $r(p, d)$ is then used to compute a reward for the response either by running the generated code or getting a reward from the reward model. The reward is then used to compute the loss function $\mathcal{L}_p$ in Equation (9.6). The loss is then used to update the base model's parameters using PPO. The process is repeated for a number of iterations T or until the model converges. Figure 9.3 provides an overview of the RLPF fine-tuning process.

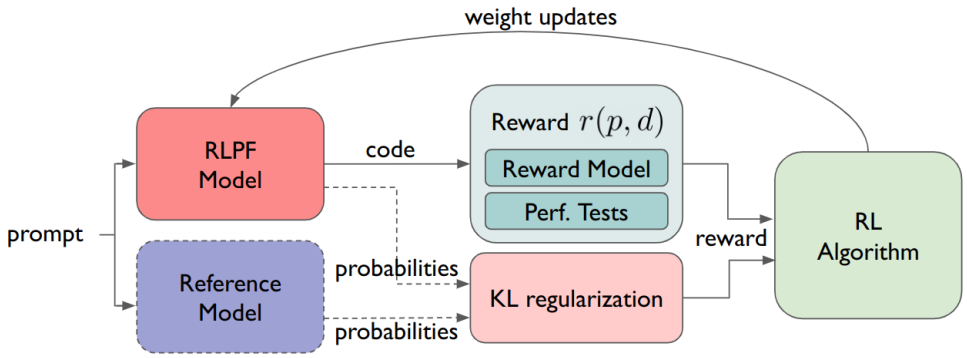


Figure 9.3: The RLPF fine-tuning process. A prompt is given to the model and a reward is calculated based on the code it generates. Additionally, the KL-divergence between a reference model and the fine-tuned model is included in the reward to prevent deviating too far from the original distribution. Finally, PPO is used to update the model's parameters based on the reward.

RLPF Fine-Tuning Evaluation Metric: We can measure the success of the reinforcement learning using two metrics: the mean reward and the magnitude of the KL-divergence over an evaluation dataset. The mean reward indicates how well the LLM being fine-tuned is able to optimize the reward function. A higher mean reward is better and indicates that the model is generating faster code. The KL-divergence measures how far the fine-tuned model has diverged from the

supervised model. The absolute magnitude of this is difficult to interpret, but it should remain positive and low to indicate that the fine-tuned model is not diverging too far from the supervised model.

9.4.3 Direct Performance Alignment

In recent work, Rafailov et al. [117] demonstrated an alternative approach that does not use reinforcement learning to align LLM outputs with certain properties. Their approach, called Direct Preference Optimization (DPO), uses a derivation of RLHF's reward objective (similar to Equation (9.6)) to directly update the model's parameters to align with a reward signal, rather than train a reward model and use RL. The derived loss takes a similar form to the reward loss in Equation (9.2). This DPO fine-tuning has many advantages over RLHF, such as requiring less computation, being easier to implement, and is generally more stable with less hyperparameters [117]. However, some works still find that RL fine-tuning can outperform DPO for certain tasks and datasets [140]. Thus, we adapt the DPO approach to compare it with RLPF. We propose Direct Performance Alignment (DPA), an adaptation of the training procedure and loss function from [117] that takes into account performance, to fine-tune an LLM to generate faster code. The proposed loss function in DPA is shown in Equation (9.7).

predicted probabilities from fine-tuned model π^P and supervised model π^S on fast (d_f) and slow (d_s) code

$$\mathcal{L}_d = -\log \sigma \left(\beta \log \frac{\pi^P(d_f \mid p)}{\pi^S(d_f \mid p)} - \beta \log \frac{\pi^P(d_s \mid p)}{\pi^S(d_s \mid p)} - \mu(p, d_f, d_s) \right) \tag{9.7}$$

Like with the reward loss in Equation (9.2), we utilize the adaptive margin μ from Equa-

tion (9.3) to scale the loss based on the runtime of the fast and slow code samples. This loss function can be used to fine-tune a base LLM to generate faster code without using reinforcement learning. To compute the loss, we need to get model predictions for a fast and slow code pair for both the model being fine-tuned and a base reference model (the supervised model). Then the loss from Equation (9.7) is used to update the weights of the model being fine-tuned. This process is iteratively repeated for a number of iterations T or until the model converges. This DPA fine-tuning process is portrayed in Figure 9.4.

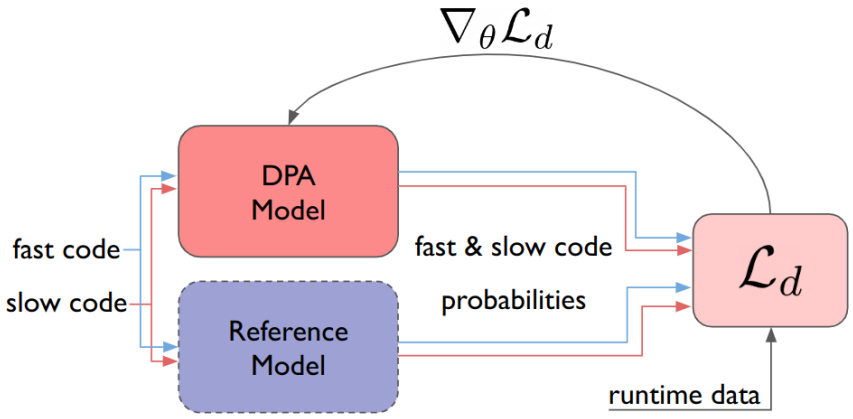


Figure 9.4: The DPA fine-tuning process. The model being fine-tuned and a reference model are used to generate probabilities for a fast and slow code sample. These probabilities, combined with runtime data, are used to compute a loss and update the model's parameters.

DPA Fine-Tuning Evaluation Metric: The success of DPA fine-tuning can be measured using a similar accuracy metric to the reward model from RLPF. Since we do not have a direct reward signal like in Equation (9.2), we can instead measure how often the difference in log probabilities between the fine-tuned model and the supervised model for the fast code, i.e. $\log \frac{\pi^P(d_f|p)}{\pi^S(d_f|p)}$ is greater than the log probability difference for the slow code, i.e. $\log \frac{\pi^P(d_s|p)}{\pi^S(d_s|p)}$. This is shown

in Equation (9.8).

$$\text{acc}_{\text{dpa}}(\mathcal{X}) = \frac{1}{|\mathcal{X}|} \sum_{(p,d_f,d_s)\in\mathcal{X}} \mathbb{1}\left[\frac{\pi^P(d_f \mid p)}{\pi^S(d_f \mid p)} > \frac{\pi^P(d_s \mid p)}{\pi^S(d_s \mid p)}\right] \tag{9.8}$$

9.5 Evaluation Tasks

It is important to quantify how well the models do on downstream tasks after fine-tuning. In this section we present two different tasks, code generation and optimization, to evaluate how well the training methodologies in Section 9.4 improved the LLMs ability to generate fast code. We further detail an ablation study to motivate the use of synthetic data.

9.5.1 Code Generation

To evaluate the ability of the models to generate fast code, we utilize two sets of coding problems. The first is a subset of 100 coding contest problems from the CodeContests dataset [86] (see Section 9.3.1) that were removed from the training set. We can provide the model with the problem statement and use it to write a solution to the problem. We can then run the code and measure both its *correctness* and *performance*. Correctness can easily be tested using the unit tests provided with the problems.

In addition to the coding contest problems, we also evaluate the models on the ParEval benchmark [102], which is a collection of parallel code generation problems for evaluating the ability of LLMs to generate correct and efficient parallel code. We narrow our focus to a subset of 180 problems, namely the serial, OpenMP [111], and MPI [128] problems. We include OpenMP and MPI problems to evaluate the models' ability to generate fast parallel code. The problems in

ParEval range a wide variety of domains, such as linear algebra, graph algorithms, sorting, etc. The problems are designed to be challenging and require the generation of efficient code. The ParEval benchmark provides a great way to test the LLMs on problems unlike what is in their training data (coding contests).

Code Generation Evaluation Metrics: We evaluate the generated code on two metrics: *correctness* and *performance*. To study correctness we adopt the popular pass@k metric from Chen et al [31]. This metric measures the probability that if an LLM is given k attempts to write a correct solution, it will succeed. Equation 9.9 shows how this value can be estimated using N generated samples from an LLM. Typically the average pass@k over a set of prompts is reported and, as LLMs have progressed, only the pass@1 value is reported. We refer the reader to [31] for further discussion of pass@k.

$$\text{pass@}k = \frac{1}{|P|} \sum_{p \in P} \left[1 - \binom{N - c_p}{k} \Big/ \binom{N}{k} \right] \tag{9.9}$$

Number of samples generated per prompt

Set of prompts

Number of correct samples for prompt p

To evaluate the performance of the generated code, we use the speedup$_n$@k metric introduced by Nichols et al [102]. This metric measures the expected max speedup over a baseline implementation if the LLM is given k attempts to write a solution. The speedup$_n$@k metric is defined in Equation 9.10. We refer the reader to [102] for a complete derivation of this metric. For the coding contest problems, we use the median submission runtime as the baseline. For the ParEval problems, we use the baselines provided by the benchmark.

runtime of baseline for prompt p

$$\text{speedup}_n@k = \frac{1}{|P|}\sum_{p\in P}\sum_{j=1}^{N}\frac{\binom{j-1}{k-1}}{\binom{N}{k}}\frac{T_p^*}{T_{p,j,n}} \tag{9.10}$$

runtime of sample j of prompt p on n processors

9.5.2 Code Optimization

In addition to generating code, we also evaluate the ability of the models to optimize existing code. This is accomplished by providing a code snippet and instructing the model to generate an optimized version of it. To evaluate this task we use the functions in the PolyBench benchmark suite [54]. This is comprised of 30 unique kernels that are typically used to test compiler optimizations and auto-tuning tools. We utilize the kernels by providing the existing kernel implementation to the LLM and instructing it to generate an optimized implementation. We can then evaluate the correctness and performance of the generated code.

Code Optimization Evaluation Metrics: We evaluate the generated code on the same metrics as the code generation task: *correctness* and *performance*. We use the same pass@k metric (Equation (9.9)) to evaluate correctness. To evaluate performance, we use speedup$_n$@k (Equation (9.10)), except with the baseline being the runtime of the original kernel.

9.5.3 Synthetic Data Ablation Study

Finally, we test our hypothesis that training on synthetic data helps the models' ability to generalize and prevents it from over-fitting to code contest data. To accomplish this we train the models exclusively on the code contests dataset $\mathcal{D}_c$ without any of the synthetic dataset $\mathcal{D}_s$.

We then evaluate the models on the code generation (Section 9.5.1) and code optimization (Section 9.5.2) tasks. We compute the same pass@k and speedup$_n$@k metrics and compare the impact of the synthetic data on the models' performance. Of most interest is the performance on the ParEval and PolyBench benchmarks, as these are the most different from the training data.

9.6 Experimental Setup

Using the large performance dataset $\mathcal{D}$ from Section 9.3 and the training methodology introduced in Section 9.4, we can now fine-tune LLMs to generate faster code. Once fine-tuned, these models can then be evaluated on the benchmarks detailed in Section 9.5. This section details the base models for fine-tuning, the data subsets for each fine-tuning task, how we implement the fine-tuning process, and the experimental setup used to evaluate the fine-tuned models.

9.6.1 Base Model for Fine-Tuning

Each of the training methodologies introduced in Section 9.4 begins with a base LLM that has already been trained and fine-tunes it further. We select the Deepseek-Coder 6.7B model [57] as the base for the supervised fine-tuning (Section 9.4.1). This model is a 6.7B parameter code LLM released by Deepseek-AI that is trained on 2T tokens comprised of mostly code with a context length of 16k tokens. We select this model due to its good performance on code generation tasks [23] and due to other works finding it a better base model for fine-tuning than the popular CodeLlama models [141]. Furthermore, its 6.7B parameter size makes it tractable for end-users to use it to generate code themselves on consumer hardware. While Deepseek-Coder is a strong base model for our studies, the proposed fine-tuning methodologies can be applied to any existing

code LLM.

For the remaining two fine-tuning methods, RLPF and DPA, we use the supervised fine-tuned deepseek model as the base. This is in line with the methodologies in [112, 117] and ensures that the model being aligned is within the distribution of the text data it is trying to model (i.e. instruction prompts as shown in Listing 3). Additionally, we use Deepseek-Coder 6.7B as the base for the reward model. The final set of models used for comparison is shown in Table 9.2.

Table 9.2: Models used for comparison in this paper. Deepseek-Coder-6.7B [57] is the base model we use in our fine-tuning methodologies.

Model Name	**Description**	**Fine-Tuning Methodology**
DS	Deepseek-Coder 6.7B base model	—
DS+SFT	DS after supervised fine-tuning	Section 9.4.1
DS+RLPF	DS+SFT after RLPF fine-tuning	Section 9.4.2
DS+DPA	DS+SFT after DPA fine-tuning	Section 9.4.3

9.6.2 Data Setup

We fine-tune the LLMs using the dataset $\mathcal{D}$ from Section 9.3. We set aside 100 contests from the CodeContests dataset for the code generation evaluation task. The dataset is further split into smaller datasets for each fine-tuning task. The supervised fine-tuning dataset, $\mathcal{D}_{\text{SFT}}$, is comprised of 40% of the full dataset, $\mathcal{D}$, and the remaining 60% is used for the reinforcement fine-tuning dataset, $\mathcal{D}_{\text{RLPF}}$, and the direct performance alignment dataset, $\mathcal{D}_{\text{DPA}}$. These two datasets can be the same since the alignment fine-tuning tasks are disjoint. The $\mathcal{D}_{\text{RLPF}}$ dataset is further split into 66% for the reward model dataset, $\mathcal{D}_{\text{REWARD}}$, and 33% for the reinforcement learning dataset, $\mathcal{D}_{\text{RL}}$. During each fine-tuning stage we set aside 5% of the respective dataset for evaluation (i.e. 5% of $\mathcal{D}_{\text{REWARD}}$ is set aside to calculate the reward model accuracy after training).

All of the dataset splits are stratified so that the proportion of code contest to synthetic data is equal to the original dataset.

When creating prompt, fast code, and slow code triplets (p, d_f, d_s) from $\mathcal{D}_c$ for RLPF and DPA fine-tuning, we select d_f randomly from the top 5 fastest solutions. We then select d_s from the slowest 50% of the solutions. Additionally, a random 5% subset of slow solutions are replaced with an incorrect solution. This is to ensure that the model is not just learning to generate fast code, but also to avoid generating incorrect code. We directly use the fast and slow code pairs from $\mathcal{D}_s$ to directly form the triplet.

9.6.3 Fine-Tuning Setup

In order to implement the fine-tuning we extend the TRL Python library [142] which is built on top of the popular transformers library [143]. TRL provides existing implementations of RLHF and DPO, which we modify to use our custom rewards, loss function, and datasets. We fine-tune the models on a single node with four 80GB A100 GPUs and two AMD EPYC 7763 CPUs.

9.6.3.1 Supervised Fine-Tuning Hyperparameters

We fine-tune the supervised model for three epochs over the $\mathcal{D}_{\text{SFT}}$ dataset. We use bfloat16 precision and a global batch size of 64 (1 sample per GPU and 16 gradient accumulation steps). To fine-tune in parallel we make use of the PyTorch fully sharded data parallelism (FSDP) implementation [150], which shards model parameters across ranks to save memory. Furthermore, we fine-tune with the Adam optimizer [74] and an initial learning rate of 1.41×10^{-5}.

9.6.3.2 Reward Model Fine-Tuning Hyperparameters

The reward model is fine-tuned with the same hyperparameters as the supervised model (Section 9.6.3.1), except it is fine-tuned for only one epoch over the $\mathcal{D}_{\text{REWARD}}$ dataset. We use a max margin of $\lambda = 3$ for the margin function $\mu(p, d_f, d_s)$.

9.6.3.3 RLPF Fine-Tuning Hyperparameters

We fine-tune the RLPF model for four PPO epochs over the $\mathcal{D}_{\text{RL}}$ dataset. We use a global batch size of four and a learning rate of 1.41×10^{-5}. The KL regularization coefficient is initialized to $\gamma = 0.1$. When sampling outputs from the fine-tuned and reference model we follow best conventions [142] and use sampling with a top-k of 0 and a top-p of 1.0.

9.6.3.4 DPA Fine-Tuning Hyperparameters

The DPA model is fine-tuned for 1 epoch over the $\mathcal{D}_{\text{DPA}}$ dataset with a global batch size of four. We employ a learning rate of 1×10^{-7} in the AdamW optimizer [90]. Additionally, we found a value of $\beta = 0.6$ to be most stable for training.

9.6.4 Evaluation Setup

For the code generation tasks we use each of the LLMs to generate code for the prompts in the evaluation subset of $\mathcal{D}_c$ and ParEval. We generate 20 samples per prompt with a temperature of 0.2 and a top-p of 0.95 following standard practices LLM code benchmarks [16, 102]. For the optimization task we similarly generate 20 optimized versions of each kernel in the PolyBench benchmark suite [54] using each of the fine-tuned LLMs.

The generated code is run on a single AMD EPYC 7763 CPU. For the ParEval OpenMP tests we report results on 8 cores and we use 512 ranks for the MPI tests. We make use of the existing tests in the CodeContests dataset and ParEval to record the correctness and runtime of the generated code. For the optimized PolyBench kernels we test correctness and runtime against the original kernel implementations. All runtimes are averaged over 5 runs.

9.7 Results

With the fine-tuned models from Section 9.4 we can now evaluate their code generation capabilities on the tasks described in Section 9.5. In this section we present the results from the fine-tuning process and the evaluation tasks.

9.7.1 Fine-Tuning Results

We record the fine-tuning metrics on the 5% evaluation datasets at the end of each fine-tuning step. The DS+SFT model yields an evaluation perplexity of 1.62. It is generally difficult to reason about specific perplexity values, but values near 1 show a strong ability to model the underlying text distribution. Since perplexity is the exponential of cross-entropy (see Equation (9.1)) a perplexity value of 1.62 means that the cross-entropy between predicted probabilities is ≈ 0.48.

The RLPF reward model achieves a final evaluation accuracy of 93% after one epoch of training calculated using Equation (9.5). This means that in 93% of samples the model assigns a higher reward signal to faster code than slower code. This is a strong result as the success of RL-based LLM fine-tuning is highly dependent on the quality of the reward model [138].

Using this reward model the DS+RLPF model is then able to achieve a mean reward of 1.8 and a KL divergence of 0.29. This means that DP-RLPF is getting a positive mean reward, while maintaining a similar distribution to the original model.

Finally, we see that the DS+DPA model achieves an evaluation accuracy of 87% calculated as show in Equation (9.8). This is not quite as high as the RLPF reward model, but is still a strong result. The log-probability difference between DS+DPA and the reference model for fast code samples is greater than the log-probability difference for slow code samples in 87% of the evaluation dataset.

9.7.2 Code Generation Results

Figures 9.5 and 9.6 show the correctness and performance results of each fine-tuned model on the code generation tasks. We see a promising trend in pass@1 scores in Figure 9.5 where the fine-tuned models improve in correctness over the baseline model. The DS+RLPF model shows the most improvement across all tasks. These improvements can be attributed to training over more data and, in the case of the RLPF and DPA models, using incorrect samples as negative rewards. Improving the correctness of the models is a strong results considering that the primary goal of this work is to improve the performance while keeping the correctness levels the same.

Figure 9.6 further details the speedup results for each fine-tuned model. We present the speedup results for OpenMP on 8 cores and MPI on 512 ranks with a sequential implementation as the baseline. Across all four benchmarks DS+RLPF produces faster code than the other three models. In the case of the code contests and ParEval serial problems, the speedup_1@1 value is easy to interpret. For instance, in the case of the serial ParEval problems, DS+RLPF generates

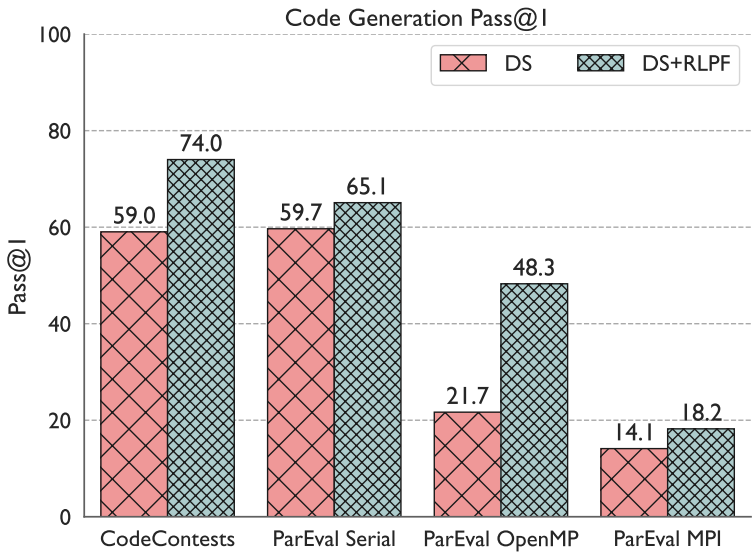


Figure 9.5: Correctness results for each model on the code generation tasks. Each of the fine-tuned models shows an improvement in correctness over the baseline model with the DS+RLPF model showing the most improvement.

code with an expected max speedup of 1.6x over the sequential baseline. We see the same order of model performance across all the benchmarks with DS+RLPF performing the best, followed by DS+DPA, DS+SFT, and DS.

9.7.3 Code Optimization Results

Figure 9.7 shows the correctness and performance results when using the fine-tuned models to optimize PolyBench kernels. DS is omitted because it is only a code completion model and was not trained to optimize code inputs. We first see that all three fine-tuned models transform the input code to a correct output code with relatively high accuracy. While provably correct compiler optimizations may seem more desirable, LLM optimizations can be applied at a higher level of abstraction and include natural language comments to explain the transformation to a developer.

We show the distribution of speedup$_1$@1 per PolyBench benchmark in Figure 9.7 rather

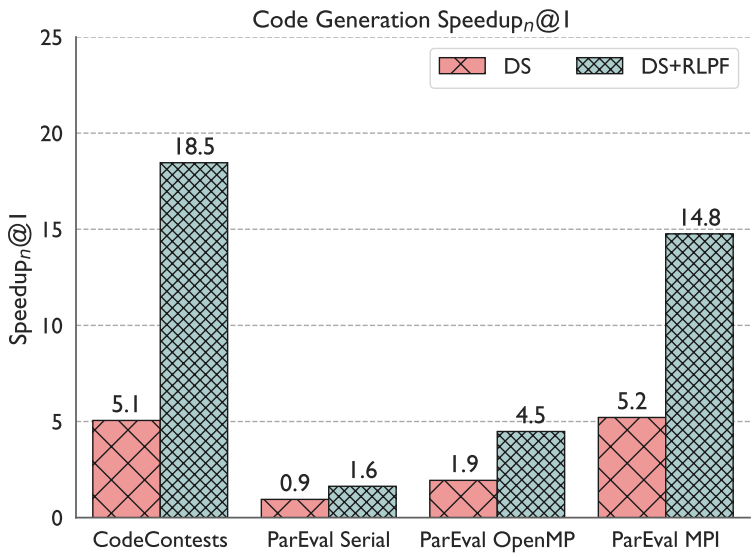


Figure 9.6: Speedup results for each fine-tuned model on the code generation tasks. OpenMP runtimes are on 8 cores and MPI runtimes are on 512 ranks. The DS+RLPF model is the best performing model across all benchmarks.

than an average to highlight the spread of results. The speedup results show that DS+RLPF is the best performing model. It is able to produce an expected max speedup greater than 1 in 26 out of the 30 benchmarks. In the case of the 3mm kernel (three matrix multiplies) it is able to get up to 22.4x expected speedup. Many of the optimizations come from loop unrolling and/or cache friendly data access patterns. The DS+DPA model is able to produce faster optimizations than DS+SFT, but is not as strong as DS+RLPF.

9.7.4 Synthetic Data Ablation Study Results

We further highlight the use of synthetic data in the fine-tuning process in Figures 9.8 and 9.9. Results for DS+RLPF are shown since it is the best performing model. We see a general improvement in both correctness and performance of generated code when incorporating synthetic data versus fine-tuning on just coding contest data. The correctness improves for all

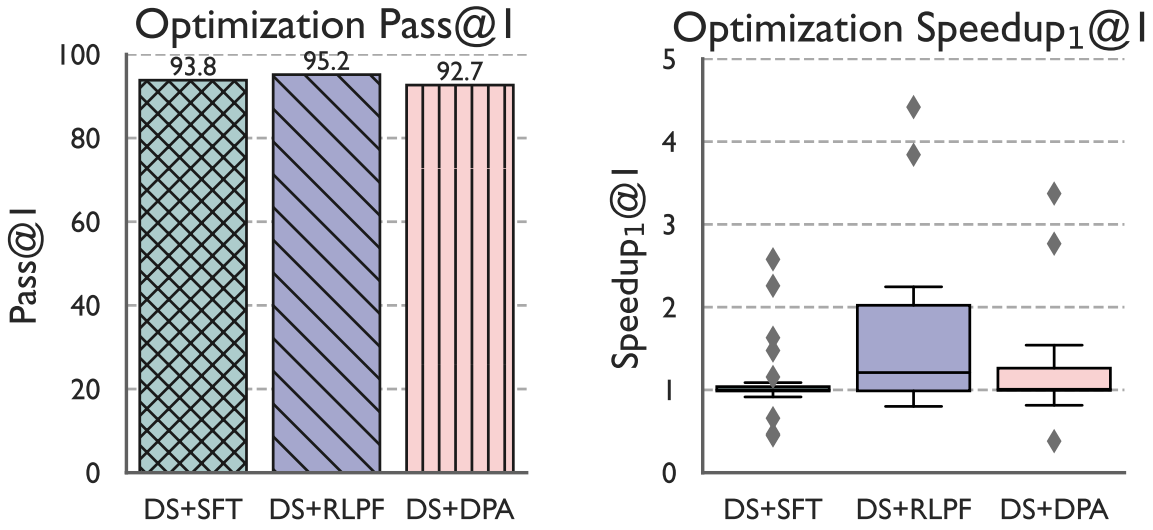


Figure 9.7: pass@1 (left) and speedup$_1$@1 (right) results for optimizing the PolyBench kernels. The distribution of speedup$_1$@1 values over the 30 benchmarks is shown on the right. The DS+RLPF model has further outliers at 11.6 and 22.4.

the benchmarks (Figure 9.8) and, notably, even improves on the coding contest benchmarks. The broader synthetic data is able to help the model generalize better even within the coding contest domain.

The speedup results in Figure 9.9 show that fine-tuning with synthetic data also helps the models produce faster code. Only in the case of the coding contests and ParEval serial problems do we see a decrease or no change in speedup$_n$@1. However, these differences are small. The performance increases for OpenMP, MPI, and PolyBench are much more significant. incorporating synthetic performance data into the fine-tuning process has prevented the models from overfitting code contest data and enabled them to generalize better to new tasks.

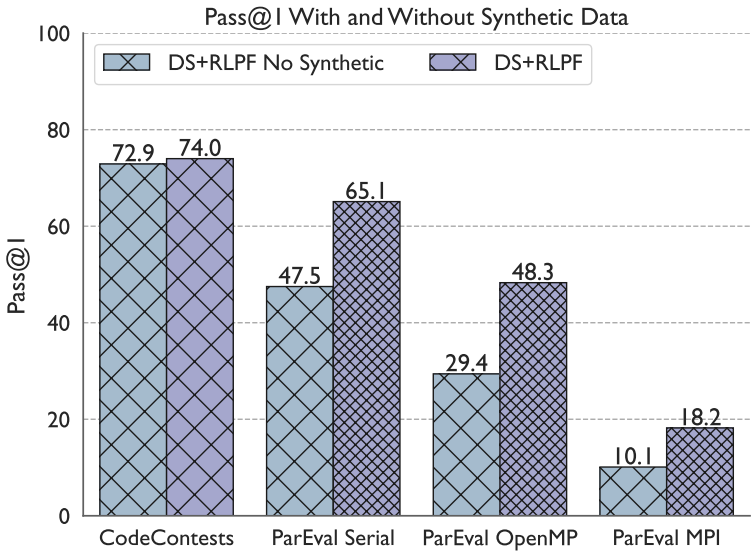


Figure 9.8: pass@1 results for DS+RLPF on each task with and without synthetic data in the fine-tuning dataset. For all tasks, the model fine-tuned on synthetic data produces correct code at a higher rate.

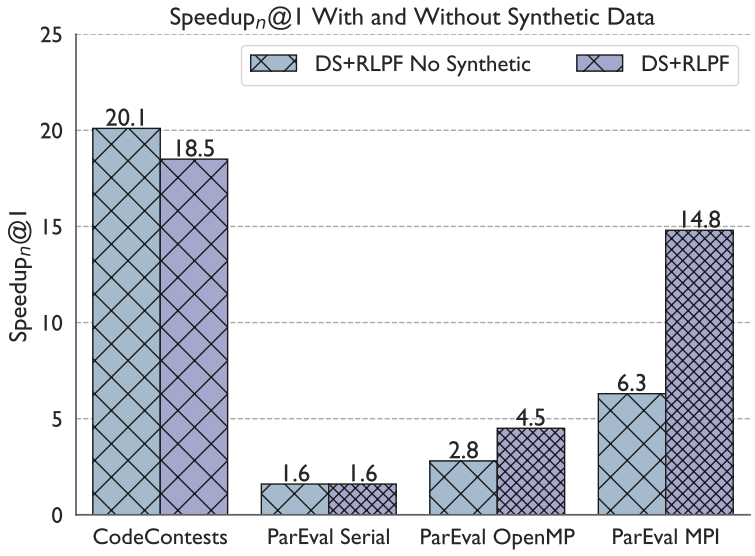


Figure 9.9: speedup$_n$@1 results for DS+RLPF on each task with and without synthetic data in the fine-tuning dataset. For OpenMP, MPI, and PolyBench tasks, the model fine-tuned on synthetic data produces faster code, while the coding contest and ParEval serial problems show a slight decrease or no change in speedup.

Chapter 10: Modeling Code: Is Text All You Need?

10.1 Motivation

Modern large language models (LLMs) have shown remarkable promise in understanding and generating source code across tasks such as completion, translation, and summarization. By leveraging massive amounts of code data available, these models have demonstrated strong capabilities in modeling lexical and syntactic aspects of code. However, they tend to struggle in reasoning about more complex, structured properties of code.

One prominent shortcoming of current code LLMs is their difficulty in accurately modeling structured, analytical properties of code, such as control and data flow. These types of relationships are best captured by graph-based models and the sequential representations of transformers are not well-suited for this task. Without explicit awareness of this structure, LLMs tend to rely heavily on surface patterns in text and are prone to errors in tasks requiring understanding of code structural properties. An example consequence of this limitation is the lack of symmetry under semantic-preserving code transformations. Two versions of a code that do the exact same thing may lead to distant internal representations in an LLM, while the static graph-based representation of the code would be identical. This would require more complicated modeling on top of the LLM's internal representation than with a graph-based approach.

Fixing this limitation of current code LLMs is valuable for many downstream tasks that rely

on code analysis, including testing, debugging, security vulnerability detection, and performance optimization. If LLMs can robustly capture and utilize the structured properties of code, they can better assist developers with high-level reasoning, automatically suggesting safer or more efficient code. This would in turn reduce the development time and effort to produce high-quality software.

While it is important to develop models that merge the benefits of code LLMs and graph reasoning, it is also non-trivial. Graph neural networks are effective at modeling structured code information, but they lack the large-scale generative abilities of LLMs. Conversely, LLMs do not readily accommodate graph representations, and naive attempts to encode graphs into text often do not scale well or lose important structural information. Ideally, a new approach should build off of the modeling capacity and generative capabilities of modern LLMS and ameliorate them with better structured reasoning around code.

In this chapter, we propose a solution that integrates graph-based reasoning into LLMs using a GNN soft prompting approach. Our method learns how to encode structured code representations into prompts that can be consumed by powerful pre-trained LLMs. To accomplish this, we propose a novel graph representation of LLVM intermediate representation (IR) that can be learned to be mapped into a language model's embedding space. By bridging key information from graph-based analyses directly into the model's latent space, we preserve the fidelity of structured reasoning, and achieve the flexibility and scale of modern LLMs.

10.2 Code Graph Representations and Soft Prompting

Below, we present the background on various structured graph representations of code and soft prompting for LLMs.

10.2.1 Structured Code Representations

When employing ML techniques to model code properties it is often beneficial to represent the code in a structured form such as abstract syntax tree (AST) or control flow graph (CFG). The nature of code generally permits the construction of such structured representations and they are often more informative than the raw text and allow models to predict code properties with higher accuracy and less parameters.

The earliest approaches to modeling code with structured properties combined ASTs with statistical features of the code to predict properties such as the underlying algorithm [97, 13]. These approaches used graph neural networks (GNNs) and custom tree convolutional networks to learn from the ASTs. However, more recent state-of-the-art approaches have had better success modeling graph constructions of intermediate representations (IR), such as LLVM IR [38, 68]. The former, ProGraML [38], constructs a graph where each node is an IR instruction and edges represent control, call, and data flow in the program. This representation, paired with a GNN, is effective at tasks such as algorithm classification and relative performance prediction. The latter, Perfograph [68], is an extension of this work that improves on the node embeddings in the graph construction. Other works have explored similar constructions of CFGs for security vulnerability modeling [129].

10.2.2 Soft Prompting

Often when utilizing LLMs prompt engineering is necessary to find the right inputs to the model to achieve the desired output. However, this can be difficult as it is not systematic and must be done by hand. One could instead fine-tune a model for the desired tasks to enforce a stricter objective, however, this itself is time-consuming and requires a large amount of compute resources. Soft prompting is a technique that lies in between the two approaches where we instead learn how to optimally map inputs into the input space of an existing LLM. This allows us to use an existing LLM without the need to fine-tune it or hand engineer prompts. Soft prompting has been shown to be effective at learning how to better provide information to LLMs and condition them for particular downstream tasks [82, 84, 88, 139, 25]. Building on top of these works, recent papers have shown that GNN-based soft prompting can be effective at reasoning through structured properties of graphs [89, 114].

10.3 Collecting IR Data at Scale

In order to learn from structured representations of code, we must first collect such data at a large enough scale for training purposes. In this section, we describe the process of collecting code and IR data at scale.

10.3.1 Collecting Pairs of Source Code and LLVM IR

While large code datasets already exist [77, 91], they are limited to source code. When any meta-data is included, it is text data such as comments, documentation, git commits, schemas, etc. These datasets do not contain structured information such as abstract syntax trees (ASTs) or

intermediate representation (IR) from compilers such as LLVM. While some of this data can be extracted from the source code, IR requires compilation, and hence, is non-trivial to collect from arbitrary codebases at scale. One large project, the ComPile dataset [55], contains a large amount of LLVM IR bytecode, but does not contain the source code paired with it. To jointly learn from the source and IR, both of these are required as paired data.

To collect this data, we build off of the work in Grossman et al. [55] and use the ComPile dataset as a starting point. We re-compile the C/C++ code in this dataset to LLVM IR and extract the corresponding source code from the metadata in the Spack package manager [47]. Furthermore, we collect the IR data using LLVM 16, with a custom version of the llvmlite library, to retrieve better data and enable more transformations downstream. The IR is collected without optimizations as these can be applied with transformations using the `opt` tool in LLVM later. The final dataset totals approximately 2 million files of C/C++.

10.3.2 Collecting Synthetic Data

The dataset described above contains real-world, complete code files alongside their compiled LLVM IR. To expand this dataset, we further collect question and answer pairs from the code and IR. Each sample in this dataset is a quadruple of the form $(\text{source code}, \text{IR}, \text{question}, \text{answer})$. Questions and answers are generated synthetically using an LLM, namely GPT-4o [110], in a similar format to the data collection in [141]. We provide a code snippet to the LLM and a random text snippet from the The Stack dataset [91] to use as inspiration. Given the code and inspiration, the LLM is tasked with generating a question and answer pair about the snippet of code. This structure is particularly useful for the task of mapping IR and question pairs to answers, i.e.

$IR \times Q \mapsto A$.

We collect another CodeQA dataset where question answer pairs are synthetically generated using an LLM and the answers are somewhere in the context of the source code. For example, a sample has source code, a question, and an answer where the answer is simply a location (or locations) in the source code that answer the question. This dataset is useful for testing how well a model can reason through structural properties of the code. If the model can generate correct responses to questions about structural properties of the code, it is likely that it has learned to model the code at a deeper level than merely the source code as text.

10.4 An Improved Structured Graph Format

Next, we present an enhanced structured graph format designed to represent LLVM Intermediate Representation (LLVM IR) constructs with greater fidelity and granularity. Building upon prior work such as ProGraML, our approach incorporates additional node and edge types, enabling a richer modeling of the elements present in LLVM IR. The goal of these enhancements is to improve the expressivity of the graph representation and better capture the underlying semantics and dataflow relationships in the IR.

10.4.1 Design of the IRGraph Format

The graph format is inspired by similar representations of code, such as ProGraML and PerfoGraph, however, it uses a finer granularity to split IR statements into the graph. It further adds more node and edge types to better distinguish information on the graph. The final graph format, IRGraph, has six node types and eight edge types, which are described below.

Node Types

Value: Represents individual LLVM IR values, such as variables or constants.

Type: Encodes type information, such as integer or floating-point types, associated with LLVM IR values.

Size: Models container sizes or dimensionality information derived from type properties.

Module: Represents the LLVM IR module as a global context for values and functions.

Attributes: Captures function and argument attributes, including linkage, visibility, and calling conventions.

Instruction: Represents individual instructions in the LLVM IR, including their operation codes and alignment information.

Edge Types

Type: Connects a value to its associated type (value $\rightarrow$ type).

Dataflow: Captures data dependencies, including definitions (instruction $\rightarrow$ value) and uses (value $\rightarrow$ instruction).

Attribute: Links values to their attributes (value $\rightarrow$ attribute).

CFG: Represents control flow between instructions (instruction $\rightarrow$ instruction).

Size: Maps types to their associated sizes (type $\rightarrow$ size).

Symbol: Connects the module to global values (module $\rightarrow$ value).

Includes: Connects contained types (type $\rightarrow$ type).

Contains: Connects constants and global variables to operands and initializers (value $\rightarrow$ value).

This structure provides more comprehensive detail about the LLVM IR code beyond control and data flow, which are typically the only focus of graph code representations such as ProGraML

and PerfoGraph. By incorporating more granular details, the representation is better able to capture nuanced relationships between different elements of the LLVM IR code. For example, attributes, symbols, and size information are crucial to understanding performance properties of code, which is a common task for graph-based code representations, yet none of the related works incorporated these details or only encapsulated them in a limited manner.

10.4.2 Graph Construction Process

The construction of the IRGraph representation begins with parsing LLVM Intermediate Representation (IR) files using LLVM 16, ensuring compatibility with the latest features and constructs of the language. A Python script and llvmlite Python bindings are used to extract nodes and edges, adhering to the structure described in the previous section. We updated portions of the llvmlite and numba libraries to support the parsing the needed LLVM IR constructs. Graphs are constructed and stored as PyTorch Geometric HeteroData objects.

We first construct nodes for values, types, instructions, attributes, and the module itself in the LLVM IR. Each node is assigned a unique feature vector that encodes its type-specific properties. For instance, value nodes include information about the kind of value they represent (e.g., constants, variables), while instruction nodes capture details such as the opcode. Type nodes encode structural details, including whether the type is scalar, vector, or array, and any associated dimensions.

Once the nodes are initialized, relationships between these components are identified and represented as edges. We first connect edges to the value nodes. These are the type, dataflow, and attribute edges. Type nodes are further connected to size nodes encoding the size of the

type. Values are finally connected to the module they reside in using symbol edges. These are constructed as undirected edges to allow information to flow back to module nodes during GNN training. To encapsulate execution flow we connect instruction nodes in the graph using control flow edges to represent the execution order of instructions. Finally, type nodes are connected to other type nodes using includes edges to represent type hierarchies, and value nodes are connected to other value nodes using contains edges to represent the relationship between constants and global variables and their operands and initializers.

One major distinction between our approach and prior work is the ability to represent entire compilation units. Previous works, such as ProGraML, focus on single functions and are not equipped to handle the entire IR module. Our approach is able to contextualize all of the information available in an IR file and represent it in the graph. This is a significant advantage as the code we desire to model is rarely a single function, but rather an entire program or compilation unit. For each entire program, we can construct an IR graph and store it as a PyTorch Geometric HeteroData object that encodes six node types and eight edge types. Furthermore, it stores feature vectors for each node in the graph based on the node type.

10.5 Experiments

In this section we highlight the benchmarks used for evaluating our approach, alongside the models and training procedures employed in our experiments.

10.5.1 Benchmarks

DevMap Dataset: The DevMap, previously used in [38], dataset consists of OpenCL kernels annotated with their performance metrics on different hardware platforms, including a CPU, an NVIDIA GPU, and an AMD GPU. The primary task is to predict whether a given kernel will perform better on the CPU or the GPU based on its source code and intermediate representation (IR) graph. This task is divided into two subtasks: one focusing on predicting performance on the NVIDIA GPU and the other on the AMD GPU. The dataset provides a comprehensive benchmark for evaluating the effectiveness of our approach in understanding and predicting hardware-specific performance characteristics of OpenCL kernels. This benchmark reveals both source code (the OpenCL) and an LLVM IR graph making it an ideal test case for our approach.

Algorithm Classification: The POJ-104 dataset [98] is used for the algorithm classification task. This dataset contains 104 different algorithm classes, each represented by multiple C++ programs. Each sample in the dataset includes the complete source code of an algorithm and its testing framework. From this we can compile and generate LLVM IR. The objective is to predict the algorithm class based on the provided source code and IR. This benchmark allows us to evaluate the capability of our approach in accurately classifying algorithms, demonstrating its effectiveness in understanding and distinguishing between different types of algorithms based on their code and IR features.

Vulnerability Detection: The Juliet test suite [1] is utilized for the vulnerability detection task. This dataset comprises approximately 100,000 C++ samples, each having a version with and without a security vulnerability. The task is to predict whether a given code sample contains a vulnerability or not. We use pair-wise accuracy as the evaluation metric, which means the model

must correctly identify both the vulnerable and non-vulnerable versions of each sample. Previous literature has demonstrated that pair-wise accuracy is a more realistic metric for evaluating vulnerability detection models. For each sample, the model is provided with the source code and its corresponding IR graph. This benchmark is critical for assessing the model's ability to detect security vulnerabilities in code.

Code Translation: The ParEval benchmark [102] is employed for the code translation task, which involves translating code from one parallel programming model to another. Specifically, we evaluate translations from sequential code to OpenMP, sequential code to MPI, and OpenMP code to CUDA. Each sample in this benchmark consists of a single C++ kernel, and the model's objective is to predict the translated code. The translations are scored based on functional correctness. The ParEval benchmark includes 60 problems for each execution model, covering 12 distinct problem types. This benchmark provides a rigorous test for assessing the ability of our approach to accurately translate code between different parallel programming models, ensuring functional correctness and performance. We use this benchmark as it enables us to evaluate the generative performance of the combined model; generative tasks are a key strength of the model as purely graph-based approaches are not well-suited to generative tasks.

10.5.2 Models and Training

We begin by evaluating the effectiveness of our new graph structure. To identify the optimal architecture for each task, we conduct an extensive architecture search across various GNN architectures, including Graph GCNs, Graph Attention Networks (GATs), and GraphSAGE. We enumerate different configurations of these architectures, varying the number of layers and hid-

den dimensions. This architecture search is done for each benchmark to determine the best overall configuration. Completing this search first enables us to use the best possible embeddings in the subsequent soft-prompting setup.

After identifying the best architecture, we proceed with a two-stage training process: masked pretraining followed by prompt fine-tuning. During the masked pretraining phase, we use the best performing GNN architecture and do masked node prediction to pre-train it on the large unlabeled IR dataset. This gives us a pre-trained graph model that can be further fine-tuned for soft-prompting.

Once pre-training is complete, we fine-tune the model using prompt-based learning. In this step, we use the pre-trained GNN and LLM to fine-tune the GNN further for soft-prompting. This fine-tuning is done over the large IR plus source code dataset, in addition to the synthetic datasets. We use cross entropy loss for next token prediction and only update the weights for the GNN with the backpropagated gradients. Fine-tuning is completed for one full epoch of the combined training datasets with a learning rate of $1e-4$ and the AdamW [90] optimizer.

10.6 Results

In this section we highlight the results of each of the benchmarks and further ablate portions of the IR graph representation.

10.6.1 Device Mapping

Figure 10.1 shows the results of the different representations on the DevMap benchmark. We compare the proposed IRGraph and IRCoder representations against baseline IR graph and

LLM models, namely ProGraML and Deepseek-Coder-6.7b. The results are shown for each of the models fine-tuned on the DevMap train set and evaluated on the DevMap test set. The LLMs are fine-tuned with a classification head instead of a language modeling head. The proposed representations both outperform the baselines. We see that just the IRGraph representation improves on the ProGraML graph model in predicting the faster device, CPU or GPU. Furthermore, the IRCoder model improves on the baseline language model demonstrating the capability of the soft-prompting technique to improve the modeling capacity of LLMs.

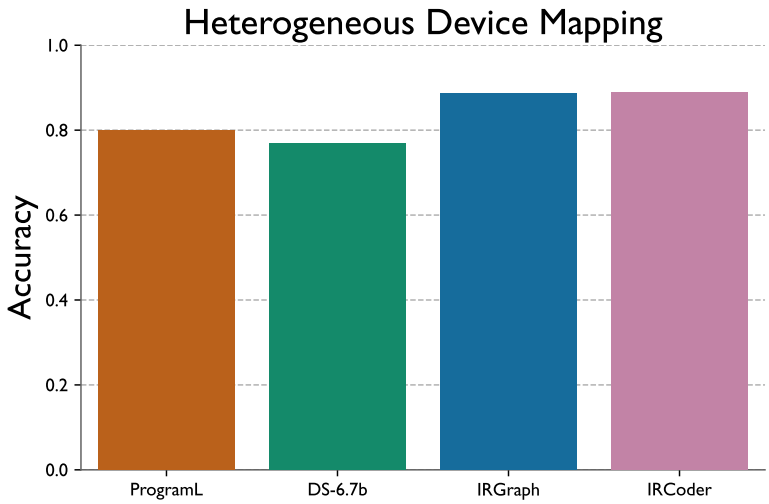


Figure 10.1: Accuracy scores from the DevMap benchmark. Both of the proposed representations outperform the respective baselines. The IRGraph graph representation improves on the ProGraML graph model, while the IRCoder language model builds on the graph to improve the language model.

10.6.2 Algorithm Classification

The results on the POJ-104 benchmark are shown in Figure 10.2. Here we show the error rate for each representation on the task of classifying the code samples into one of 104 algorithm classes. The proposed IRGraph representation performs on par with the ProGraML graph model, however, the Deepseek-Coder language model performs worse. The IRCoder model is able to

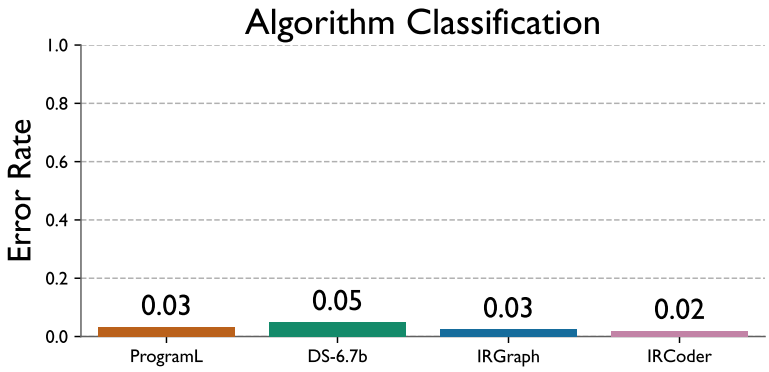


Figure 10.2: Error rate scores from the POJ-104 benchmark. All representations are strong at this task. The IRGraph representation scores the same as ProGraML while the IRCoder representation outperforms the Deepseek-Coder baseline.

combine the benefits of the LLM and IRGraph representations to outperform all baselines. These trends are consistent with the DevMap benchmark results.

10.6.3 Vulnerability Detection

The final classification benchmark, vulnerability detection, is shown in Figure 10.3. In this benchmark the model is shown a version of a C/C++ code from the Juliet dataset and is tasked with predicting if it is vulnerable or not. Following trends in recent literature we measure pair-wise accuracy where a correct prediction requires both the vulnerable and non-vulnerable versions of a sample to be correctly classified.

We observe similar trends for vulnerability detection to the other two classification benchmarks. The IRGraph and IRCoder models outperform the baseline models. However, unlike the other two benchmarks we see slightly worse pair-wise accuracy for IRCoder than IRGraph, albeit the difference is small. Altogether, these results demonstrate the effectiveness of the proposed representations for modeling code properties. Furthermore, they demonstrate the effectiveness of incorporating structural information into language models for code understanding tasks.

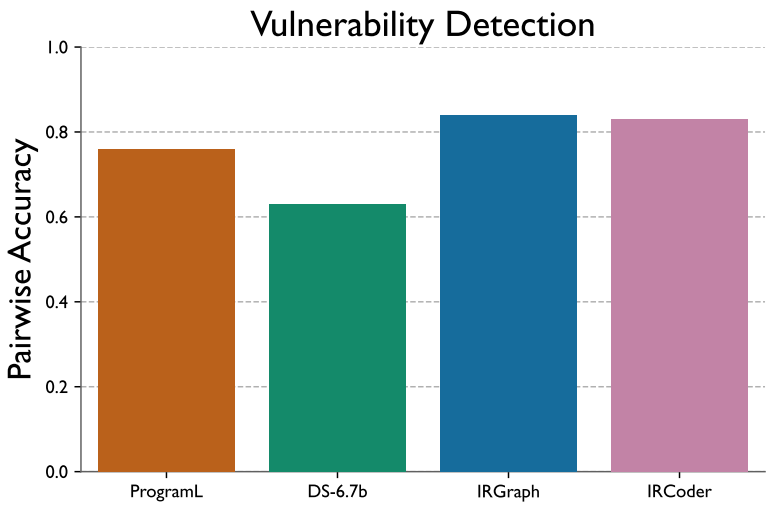


Figure 10.3: Pair-wise accuracy scores from the Juliet benchmark. Pairwise accuracy, where a correct prediction requires both the vulnerable and non-vulnerable versions of a sample to be correctly classified, is used as the evaluation metric. The IRGraph and IRCoder representations outperform the baselines.

10.6.4 Code Translation

The results on the ParEval benchmark are shown in Figure 10.4. Here the base LLM, Deepseek-Coder, is given a correct implementation of a function in one parallel programming model and asked to generate the equivalent implementation in another model. The IRCoder model uses the same setup except with the IR graph of the source implementation provided in addition. This gives the model more context about the structure and properties of the code it is attempting to translate. Pass@1 scores are reported demonstrating the functional correctness of the generated code.

We observe a substantial increase in correctness of the translated code between the base LLM and IRCoder. This trend is observed across all translation tasks: sequential to OpenMP, sequential to MPI, and OpenMP to CUDA. The most prominent improvement is for OpenMP to CUDA translation. We hypothesize this is due to the OpenMP and CUDA code being the most

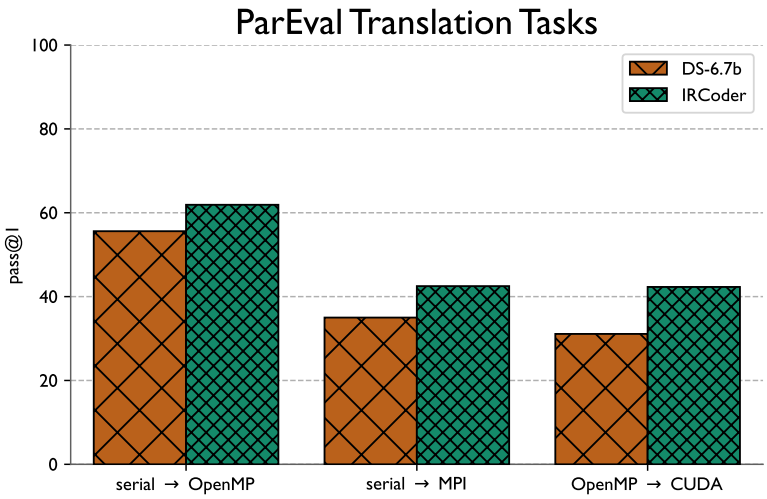


Figure 10.4: pass@1 scores from the ParEval benchmark comparing Deepseek-Coder and IRCoder. The IRCoder model is better able to translate code when provided with the IR graph during translation. The most pronounced improvement is for the OpenMP to CUDA translation.

similar in terms of structure meaning the LLM is able to better use the structural information available in the IR graph.

Improving generative tasks is a key strength of the proposed model. Even though the GNN-only models are able to get comparable modeling performance with significantly less parameters, they are not suited to generative tasks. Being able to reason about and understand deep code structures is an important capability for a code LLM to have as it allows for more accurate and contextually aware code generation.

10.6.5 IRGraph Ablation

To better understand the proposed graph representation, we perform ablation studies to determine the importance of different node and edge types. Figure 10.5 shows the results of ablation studies on the IRGraph representation. Results are collected by removing individual node and edge types from the graphs and training the GNN model on the reduced graphs. In

some cases, this requires making edges bi-directional to prevent graphs where node types have no incoming edges and information cannot be propagated to them during message passing. Results are shown for both the DevMap and algorithm classification benchmarks.

The results from the ablation study show that certain node types are more critical for the model's performance. Specifically, the removal of value and instruction nodes leads to a significant drop in accuracy. Conversely, the removal of IR attribute nodes has a minimal impact on performance, suggesting that these nodes contribute less to the overall model accuracy. The performance drops are more significant for the device mapping benchmark, than for algorithm classification, however, the trends are the same. This is likely due to the device mapping task being a more complex task that requires more information from the IR graph.

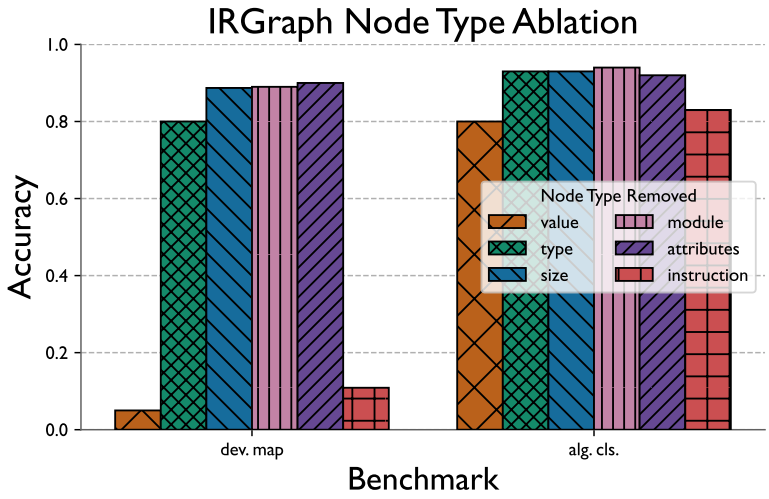


Figure 10.5: Ablation study by removing node types from the IRGraph representation. We see that value and instruction node types are the most important data points for modeling the IR. The IR attributes are the least important and only reduce the accuracy by less than 1% when removed.

A similar ablation study for the edge types is shown in Figure 10.6. Here we observe the type and dataflow edges having the most significant impact on the model's performance. Surprisingly, other edge types have a minimal impact on accuracy. Most notable is the removal

of control flow edges, which only reduces accuracy by 4 percentage points from the full model for the DevMap benchmark. As with the node ablation study, the trends are consistent, but less pronounced for the algorithm classification benchmark. The type and dataflow edges lead to the largest drop in accuracy when removed, but only by 3 percentage points for the algorithm classification benchmark.

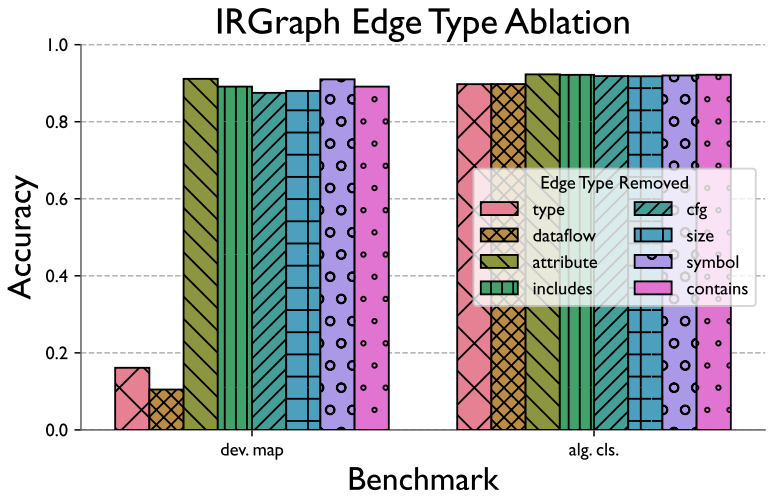


Figure 10.6: Ablation study by removing edge types from the IRGraph representation. We see that type and dataflow edges are the most important for the model's performance, while the other edge types have a minimal impact on accuracy.

For both the node and edge ablation studies, we observe that the full IRGraph representation outperforms all ablated versions, suggesting that all node and edge types contribute to the model's performance. However, some types contribute significantly less than others. As removing components never leads to better performance there is no need to strip them from the final graph format.

Chapter 11: One Profile is All You Need: Performance Aligned Embedding Spaces

11.1 Motivation

One of the most common applications of learned empirical performance models is to predict some observed property of the run of an application. For example, we may want to use the code and input problem size as inputs to a model to predict the relative runtime or cache usage of a scientific application. Many works have found successful approaches to a number of different modeling problems, but they often focus solely on modeling a single output variable. If we want to model a new output variable, then we typically have to conduct an entire new study and we may require new data and/or an entirely different methodology. A simple change from predicting runtime to memory consumption may warrant an entirely new experimental design. This severely limits the applicability of using these models in practice as they cannot generalize to new scenarios very well without considerable engineering effort.

The difficulty in adapting these ML models that learn to directly predict a performance output variable from some inputs is inherent to the way that they are trained. Training a model in a supervised fashion leads to them learning a more direct relationship between the output variable and the inputs provided, but this relationship may not generalize to different distributions

of outputs. For example, a model that predicts runtime based on problem size will learn the complex relationship between these two values, but may learn nothing about how problem size relates to cache misses. We may be fortunate that cache misses and runtime correlate well, but this assumption will not always hold.

Developing models that can readily be applied to various types of observed code output properties will tremendously improve both the **applicability** and **efficacy** of the models, while also potentially providing deeper insights into the relationships between code properties. By generalizing across multiple tasks such models are more readily applicable to a wide variety of tasks. Developers can train them once and apply them to a number of tasks without worrying about collecting more data or doing more modeling studies. In addition to being easier to use, these models may actually provide better insights and predictions into the tasks they are trained on. By learning to handle multiple tasks, the model may learn more about the underlying relationships between the inputs and outputs, which may lead to better predictions for each individual task.

Performance data from different observed modalities is, however, very difficult to model jointly. For one, it is difficult to collect data from each of these during the run of an application. For example, profiling tools, at the time of writing, are not capable of collecting performance, calling-context, energy, and memory data all at once. Even if we could collect all of this data, it is difficult to model them jointly as they have distinct distributions and data types. Calling contexts are graph data while memory and energy data are typically time series. This makes it difficult to model them together as traditional ML models can only handle a single data type at a time.

In this chapter, we introduce a methodology for aligning embedding models for several different observed code properties. We show that by aligning the embedding spaces of these models, we can learn a single model that can predict multiple observed code properties. In

essence, by aligning the embedding spaces we can reason about the energy properties of code using a memory model of that code. This learning is accomplished through alignment with a contrastive objective.

11.2 Data Collection

In this section, we detail the data collection process for the applications and kernels used in our study. We collect data from a number of applications and kernels to study the performance characteristics of the code. We collect data from the applications and kernels using the `perf` tool to collect wall time, cache misses, and energy data.

11.2.1 Applications Profiled

In order to study the different output metrics of interest we profile a number of representative applications from the Exascale Computing Project (ECP) Proxy Applications Suite [44]. Namely, we profile the following applications:

- **AMG** [61]: A parallel algebraic multigrid solver for large-scale linear systems.
- **Laghos** [41]: A parallel unstructured finite element hydrodynamics mini-app.
- **Kripke** [78]: A simple block-structured grid mini-app for solving the discrete ordinates transport equation.
- **LULESH** [72]: A proxy application for shock hydrodynamics.
- **MiniFE**: A proxy application for unstructured implicit finite element codes.

Each of these are run on 8, 32, and 64 cores using MPI and OpenMP parallelism. They are run across a range of problem sizes to capture the performance characteristics of the applications.

We additionally collect data from the ParEval benchmark suite [102] (see Chapter 6 for more details on ParEval). We run each kernel from the benchmark on the list of supported indices and profile its execution.

11.2.2 Performance Metrics Collected

For each of the applications and kernels listed in Section 11.2.1, we collect performance data using the `perf` tool. We collect the following performance metrics: runtime, cache misses, calling context tree, energy usage, and memory usage. We collect these metrics for each application and kernel run on 8, 32, and 64 cores for each input problem size. The data is collected on a machine with an Intel Xeon E5-2690 v4 processor running at 2.6 GHz on 64 cores with 32 GB of RAM. We ensure that each run takes at least 5 seconds to run to ensure that we have accurate enough profiling data. The run time is recorded as a single scalar value. The cache misses, energy, and memory data are recorded as time series data. Finally, the calling context tree is recorded as a graph data structure.

11.3 Methodology

In this section, we detail the methodology for aligning embedding models for several different observed code properties. This approach is demonstrated for the following metrics: CCT, memory usage, and power consumption.

11.3.1 General Alignment Approach

To create an aligned embedding space, we first start with an embedding model for each output metric we want to model. We denote each output modality of interest as $\mathcal{M}_i$ and the corresponding embedding model as $g_i : \mathcal{M}_i \mapsto \mathbb{R}^\lambda$. Each of these embedding models, g_i, takes their respective metric as input and outputs a λ dimension vector $\boldsymbol{e} \in \mathbb{R}^\lambda$. Naively, we could train each of these models separately to generate embeddings for their respective modality. There are a myriad of existing representational learning techniques for accomplishing this task, however, we want to learn representations that are *aligned*. These aligned representations will live in the same vector space so that we can reason about data from the different modalities with the same model.

To train the embedding models to be aligned we choose a "binding" model and train each model to produce embeddings close to the binding model. Each of the pair-wise learnings are done using contrastive learning where we score the model generations based on how "close" similar predictions are and how "far" apart dissimilar predictions are. We express this objective using the *InfoNCE* loss function shown in Equation (11.1). This loss function, as shown, computes the loss between two output modalities $\mathcal{M}_1$ and $\mathcal{M}_2$. We first compute output embeddings from the corresponding embedding models $\boldsymbol{q}_i = g_1(\boldsymbol{x}_i^{(1)})$ and $\boldsymbol{k}_i = g_2(\boldsymbol{x}_i^{(2)})$ where $\boldsymbol{x}_i^{(j)}$ is the *i*-th data sample from the *j*-th data modality. Considering the embedding of the first modality, $\boldsymbol{q}_i$, the *InfoNCE* loss function rewards its similarity to the embedding of $\boldsymbol{k}_i$ using cosine distance, while penalizing how similar it is to $\boldsymbol{k}_j$ where $j \neq i$. In practice we use a symmetric loss

$\mathcal{L} = \mathcal{L}_{\mathcal{M}_1,\mathcal{M}_2} + \mathcal{L}_{\mathcal{M}_2,\mathcal{M}_1}$.

$$\mathcal{L}_{\mathcal{M}_1,\mathcal{M}_2} = -\log \frac{\exp\left(\boldsymbol{q}_i^\intercal \boldsymbol{k}_i/\tau\right)}{\exp\left(\boldsymbol{q}_i^\intercal \boldsymbol{k}_i/\tau\right) + \sum_{j\neq i} \exp\left(\boldsymbol{q}_i^\intercal \boldsymbol{k}_j/\tau\right)} \tag{11.1}$$

To accomplish this training task we begin by selecting a unifying modality $\mathcal{M}_b$ to align all the other embedding models to. This is the "binding" model. We begin by training the binding model g_b on its own as a standalone embedding model. Once this model is trained we can train the other models using the *InfoNCE* loss objective with $\mathcal{M}_b$ as one of the modalities. This is accomplished by passing corresponding data samples $\boldsymbol{x}_i^{(j)}$ and $\boldsymbol{x}_i^{(b)}$ through g_i and g_b respectively to compute their embeddings. With these embeddings we can compute the *InfoNCE* loss and use backpropagation and gradient descent to update the embedding model weights. We freeze the binding model g_b during this phase of training and do not update its weights. Furthermore, we limit to an inter-batch contrastive objective where negative samples ($i \neq j$) are only included from the current training batch. This provides a noisier estimate of the contrastive term in exchange for improved computational efficiency. This training procedure is repeated for all $\mathcal{M}_i$ where $i \neq b$ to produce aligned embedding spaces. This alignment training methodology is inspired by [52] and [154] where image, language, video, audio, and depth modalities are aligned to a single modalities latent space.

By training in this fashion the embedding spaces for *similar* runs will be closer together than those for *dissimilar* runs. Similar runs are those that might have similar runtime behavior, come from the same application, or have similar source code. Since the embedding spaces are aligned, we can use the embedding vectors to study the relationship between different runs. For example, given a power consumption profile of a run, we can compute its embedding vector and

see whether it is closer to embeddings of CCTs from load-balanced and load-imbalanced runs. If it is closer to the load-imbalanced embedding vector, for instance, then we can guess that this power profile comes from a load-imbalanced run. Similar zero-shot studies can be done for other tasks without having to train new models across the different metrics.

These aligned vector spaces can be applied to a number of performance modeling studies. For example, we can use the nearness to the source code embedding to accomplish application identification based on recorded counter data. This is a common HPC modeling task. Additionally, we can use the nearness to the load-balanced and load-imbalanced embeddings to predict whether a run is load-balanced or not. In general, the aligned embedding space can be used to predict various performance aspects of different metrics using different output metrics.

11.3.2 Individual Embedding Models

The above methodology (Section 11.3.1) for aligning multiple embedding models requires quality embedding architectures to work successfully. If each embedding model g_i is too shallow or incapable of producing quality embeddings, then the alignment will not work. Fortunately, there are a number of techniques for producing good embeddings of data across the representations of interest (graphs, sequences, and vectors).

Graph and Tree Embedding Models

To embed graph data, such as CCTs or program graphs like ASTs, we employ graph variational auto-encoders (VGAE). In this setup we will assume we have a graph $\mathcal{G} = (V, E)$ with $|V|$ nodes and $|E|$ edges. Furthermore, each node $v \in V$ has a vector of values $\boldsymbol{x}_v \in \mathbb{R}^d$ and

each edge also has an associated value vector $\boldsymbol{x}_e \in \mathbb{R}^d$. The primary architecture in the VGAE is a GNN, which models the graph structure and handles propagating information between nodes and edges. There are a number of ways to implement this message passing between nodes; we employ the standard message passing in graph convolutional networks (GCNs). In a GCN layer information is aggregated at each node from all its neighboring nodes. Following this aggregation the node values are projected using a learned linear projection into some embedding space. As shown in Kipf and Welling [75], this can be implemented neatly as $\sigma\left(\boldsymbol{A}\boldsymbol{X}\boldsymbol{\Theta}\right)$ where σ is some activation function, $\boldsymbol{A}$ is the graph's adjacency matrix, $\boldsymbol{X}$ is a matrix of the node values, and $\boldsymbol{\Theta}$ are the learned weights for the linear projection. This GCN operation is repeated for the desired number of layers. After the final layer we have a collection of $|V|$ vectors $\boldsymbol{x}_v \in \mathbb{R}^d$ for each node. To produce a vector representation for the entire graph we can aggregate all of the node vectors into a single d dimensional vector for the whole graph. This aggregation can be done with any reduction; we choose to average the node vectors.

The GCN layers and aggregation define the main component of the VGAE setup: producing the graph embedding. However, we need another model, a decoder, to train the first one. This model is responsible for learning to reconstruct the graph based on the vector produced by the first model, the encoder. In this setup we train the encoder based on how well the decoder is able to reconstruct the graph from its embedding. If it outputs vectors that represent the data well, then the decoder should be able to do a better job at reconstructing the graph.

Trace Embedding Models

For the trace data, we use a transformer model to embed the trace data. Transformers [137] are the current state-of-the-art architecture for sequence modeling. They learn sequences through a series of self-attention mechanisms that learn how much each value in the sequence should "attend" to other tokens in the sequence. One difficulty in using transformers is their fixed input size. To handle this, we choose a sufficiently large max sequence length to model and pad any smaller input sequences to this length. This padding is masked out in the self-attention mechanism to ensure that the model does not attend to these values. The transformer model is trained to predict the next value in the sequence given the previous values.

11.4 Evaluation Tasks

We evaluate the aligned performance embedding models on two tasks: (1) classifying low-medium-high energy usage applications and (2) predicting the presence of L2 cache miss peaks. We do these prediction tasks using unrelated output metrics i.e. can we utilize memory traces to predict the energy usage.

11.4.1 Energy Usage Classification

We create this classification task by splitting the ParEval kernels and proxy applications into three categories based on their energy usage. We use the energy usage data collected from the `perf` tool to classify the applications. We split the applications into three categories: low, medium, and high energy usage.

The alignment models are trained only on the ParEval kernels and then tested on the proxy

applications. We utilize the memory embeddings from the proxy applications to train a DNN classifier to predict the energy class. We compare the accuracy of this model with a model directly trained to predict energy usage from the proxy application's memory (i.e. one that has not been aligned).

11.4.2 L2 Cache Miss Peak Prediction

Similar to the energy usage classification task, we create a classification task for predicting the presence of L2 cache miss peaks. We define L2 cache peaks as a sudden increase in the number of L2 cache misses beyond 1.5 times the standard deviation of the cache misses. The presence of a single peak is enough to classify a run as having a peak. We split the applications into two classes: those with L2 cache miss peaks and those without.

The alignment models are trained on the ParEval kernels and then tested on the proxy applications. We utilize the memory embeddings from the proxy applications to train a DNN classifier to predict the presence of L2 cache miss peaks. As before we compare the accuracy of this model with a model directly trained to predict L2 cache miss peaks from the proxy application's memory.

11.5 Results

In this section we present the results for the two alignment benchmarks: energy usage classification and L2 cache miss peak prediction. We compare the performance of the aligned models to the performance of models trained directly on the proxy application data.

11.5.1 Energy Usage Classification

Table 11.1 shows the classification accuracy of the aligned and base models for energy usage classification. The base model is trained directly on the proxy application data, while the aligned model is first aligned across all the modalities before being trained. We see that the aligned model outperforms the base model across all sample sizes. Furthermore, it is able to learn faster acheiving its peak accuracy at 100 samples, while the base model requires 1000 samples to reach a much less accuracy.

Table 11.1: Energy Usage Classification Accuracy

Model	**# Samples**	**Accuracy**
Base	10	0.54
Base	100	0.59
Base	1000	0.74
Aligned	10	0.83
Aligned	100	0.86
Aligned	1000	0.86

11.5.2 L2 Cache Miss Peak Prediction

Table 11.2 shows the performance metrics for L2 cache miss peak prediction. The aligned model achieves higher scores across all metrics compared to the base model, with particularly notable improvements in recall and F1-score. This indicates that the aligned model is better at identifying the presence of L2 cache miss peaks in the proxy applications. In particular, by aligning it with other modalities, not only has it gained to ability to reason about data from those modalities, but it has also improved its ability to predict cache miss peaks.

Table 11.2: L2 Cache Miss Peak Prediction Performance

Model	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Base	0.8000	0.9524	0.7692	0.8510
Aligned	0.9143	0.9524	0.9090	0.9302

Chapter 12: Conclusion

This dissertation presents novel contributions towards the development of performance models that can make use of multiple modalities of program data and performance metrics. Chapters 4 and 5 demonstrate how performance models with multiple input sources can be used in practice by incorporating the models in cluster batch scheduling algorithms. Chapters 6 to 8 then demonstrate how to extend performance modeling to source code through the use of large language models. The insights from these chapters are utilized in Chapter 9 to develop a novel reinforcement learning algorithm to fine-tune code LLMs to generate efficient parallel code. Finally, Chapters 10 and 11 demonstrate how to incorporate more input modalities into these models through alignment and contrastive learning.

The findings in this dissertation have implications for both the design and application of performance models in future research. Future performance models can build on the multi-modal approaches demonstrated in this dissertation to account for a wider range of factors that influence performance. For example, models could incorporate additional modalities such as hardware performance counters, compiler flags, or even user-provided annotations. In addition to being more accurate, these models will also be more applicable for developers. Often developers do not have access to a full suite of performance data, and models that can make use of a sparse subset of data will be more useful in practice.

Bibliography

[1] 2017. URL: https://samate.nist.gov/SARD/test-suites/112.

[2] Omar Aaziz, Jonathan Cook, and Mohammed Tanash. "Modeling Expected Application Runtime for Characterizing and Assessing Job Performance". In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 2018, pp. 543–551. DOI: 10.1109/CLUSTER.2018.00070.

[3] Laksono Adhianto et al. "HPCToolkit: Tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701.

[4] A. Agelastos et al. "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 154–165.

[5] Wasi Uddin Ahmad et al. "A Transformer-based Approach for Source Code Summarization". In: *ArXiv* abs/2005.00653 (2020).

[6] Toufique Ahmed and Prem Devanbu. "Learning code summarization from a small and local dataset". In: *ArXiv* abs/2206.00804 (2022).

[7] Dong H. Ahn et al. "Flux: Overcoming Scheduling Challenges for Exascale Workflows". In: *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 2018, pp. 10–19. DOI: 10.1109/WORKS.2018.00007.

[8] Dong H. Ahn et al. "Scalable Composition and Analysis Techniques for Massive Scientific Workflows". In: *2022 IEEE 18th International Conference on e-Science (e-Science)*. 2022, pp. 32–43. DOI: 10.1109/eScience55777.2022.00018.

[9] *Aizu*. https://judge.u-aizu.ac.jp/onlinejudge/.

[10] Adrian Pope et al. *Swfft*. https://git.cels.anl.gov/hacc/SWFFT. 2017.

[11] Loubna Ben Allal et al. "SantaCoder: don't reach for the stars!" In: *arXiv preprint arXiv:2301.03988* (2023).

[12] Miltiadis Allamanis. "The Adverse Effects of Code Duplication in Machine Learning Models of Code". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, 143–153. ISBN: 9781450369954. DOI: 10.1145/3359591.3359735. URL: https://doi.org/10.1145/3359591.3359735.

[13] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=BJOFETxR-.

[14] Mikel Artetxe et al. *Efficient Large Scale Language Modeling with Mixtures of Experts*. 2021. DOI: 10.48550/ARXIV.2112.10684. URL: https://arxiv.org/abs/2112.10684.

[15] *AtCoder*. https://atcoder.jp/.

[16] Jacob Austin et al. "Program Synthesis with Large Language Models". In: *CoRR* abs/2108.07732 (2021). arXiv: 2108.07732. URL: https://arxiv.org/abs/2108.07732.

[17] Yuntao Bai et al. *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*. 2022. arXiv: 2204.05862 [cs.CL].

[18] Prasanna Balaprakash et al. "Autotuning in High-Performance Computing Applications". In: *Proceedings of the IEEE* 106.11 (2018), pp. 2068–2083. DOI: 10.1109/JPROC.2018.2841200.

[19] Loubna Ben Allal et al. *Cosmopedia*. 2024. URL: https://huggingface.co/datasets/HuggingFaceTB/cosmopedia.

[20] Tal Ben-Nun and Torsten Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis". In: *ACM Comput. Surv.* 52.4 (Aug. 2019). ISSN: 0360-0300. DOI: 10.1145/3320060. URL: https://doi.org/10.1145/3320060.

[21] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. "Hatchet: Pruning the Overgrowth in Parallel Profiles". In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. LLNL-CONF-772402. Nov. 2019. URL: http://doi.acm.org/10.1145/3295500.3356219.

[22] Abhinav Bhatele et al. "The Case of Performance Variability on Dragonfly-based Systems". In: *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IPDPS '20. IEEE Computer Society, May 2020.

[23] *Big Code Models Leaderboard - a Hugging Face Space by bigcode*. 2023. URL: https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.

[24] Sid Black et al. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*. Version 1.0. If you use this software, please cite it using these metadata. Mar. 2021. DOI: 10.5281/zenodo.5297715. URL: https://doi.org/10.5281/zenodo.5297715.

[25] Tsachi Blau et al. *Context-aware Prompt Tuning: Advancing In-Context Learning with Adversarial Methods*. 2024. arXiv: 2410.17222 [cs.CL]. URL: https://arxiv.org/abs/2410.17222.

[26] Andrea Borghesi et al. "Predictive Modeling for Job Power Consumption in HPC Systems". In: *High Performance Computing*. Ed. by Julian M. Kunkel, Pavan Balaji, and Jack Dongarra. Cham: Springer International Publishing, 2016, pp. 181–199. ISBN: 978-3-319-41321-1.

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

[28] Federico Cassano et al. “MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation”. In: *IEEE Transactions on Software Engineering* 49.7 (2023), pp. 3675–3691. DOI: 10.1109/TSE.2023.3267446.

[29] Le Chen et al. *Data Race Detection Using Large Language Models*. 2023. arXiv: 2308.07505 [cs.LG].

[30] Le Chen et al. “LM4HPC: Towards Effective Language Model Application in High-Performance Computing”. In: *OpenMP: Advanced Task-Based, Device and Compiler Programming*. Ed. by Simon McIntosh-Smith et al. Cham: Springer Nature Switzerland, 2023, pp. 18–33. ISBN: 978-3-031-40744-4.

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

[32] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. eprint: arXiv:2107.03374.

[33] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[34] Younghyun Cho et al. “Harnessing the Crowd for Autotuning High-Performance Computing Applications”. In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2023, pp. 635–645. DOI: 10.1109/IPDPS54959.2023.00069.

[35] *CodeChef*. https://www.codechef.com/.

[36] *CodeForces*. https://codeforces.com/.

[37] Ian J. Costello and Abhinav Bhatele. *Analytics of Longitudinal System Monitoring Data for Performance Prediction*. 2020. eprint: arXiv:2007.03451.

[38] Chris Cummins et al. “ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations”. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 2244–2253. URL: https://proceedings.mlr.press/v139/cummins21a.html.

[39] *CUPTI*. Accessed: 2023-09-30. URL: https://docs.nvidia.com/cuda/cupti/index.html.

[40] DeepSeek-AI et al. *DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence*. 2024. arXiv: 2406.11931 [cs.SE]. URL: https://arxiv.org/abs/2406.11931.

[41] Veselin A. Dobrev, Tzanio V. Kolev, and Robert N. Rieben. "High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics". In: *SIAM Journal on Scientific Computing* 34.5 (2012), B606–B641. DOI: 10.1137/120864672. eprint: https://doi.org/10.1137/120864672. URL: https://doi.org/10.1137/120864672.

[42] Xueying Du et al. *ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation*. 2023. arXiv: 2308.01861 [cs.CL].

[43] Akash Dutta et al. "Performance Optimization using Multimodal Modeling and Heterogeneous GNN". In: *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '23. Orlando, FL, USA: Association for Computing Machinery, 2023, 45–57. ISBN: 9798400701559. DOI: 10.1145/3588195.3592984. URL: https://doi.org/10.1145/3588195.3592984.

[44] *ECP Proxy Applications*. https://proxyapps.exascaleproject.org/. Accessed: 2023-09-30.

[45] Brian Van Essen et al. "LBANN: livermore big artificial neural network HPC toolkit". In: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, Austin, Texas, USA, November 15, 2015*. ACM, 2015, 5:1–5:6. DOI: 10.1145/2834892.2834897.

[46] Charles R. Ferenbaugh. *Pennant*. https://github.com/lanl/PENNANT. 2016.

[47] T. Gamblin et al. "The Spack package manager: bringing order to HPC software chaos". In: *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, 2015. DOI: 10.1145/2807591.2807623. URL: https://doi.ieeecomputersociety.org/10.1145/2807591.2807623.

[48] Leo Gao et al. "The Pile: An 800GB Dataset of Diverse Text for Language Modeling". In: *CoRR* abs/2101.00027 (2021). arXiv: 2101.00027. URL: https://arxiv.org/abs/2101.00027.

[49] Luyu Gao et al. "PAL: Program-aided Language Models". In: *arXiv preprint arXiv:2211.10435* (2022).

[50] Spandan Garg et al. "DeepDev-PERF: a deep learning-based approach for improving software performance". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022).

[51] Fabrizio Gilardi, Meysam Alizadeh, and Maël Kubli. "ChatGPT outperforms crowd workers for text-annotation tasks". In: *Proceedings of the National Academy of Sciences* 120.30 (July 2023). ISSN: 1091-6490. DOI: 10.1073/pnas.2305016120. URL: http://dx.doi.org/10.1073/pnas.2305016120.

[52] Rohit Girdhar et al. "ImageBind: One Embedding Space To Bind Them All". In: *CVPR*. 2023.

[53] Aaron Gokaslan and Vanya Cohen. *OpenWebText Corpus*. http://Skylion007.github.io/OpenWebTextCorpus. 2019.

[54] John Cavazos Scott Grauer-Gray. *Polybench*. https://web.cs.ucla.edu/~pouchet/software/polybench/. 2012.

[55] Aiden Grossman et al. “ComPile: A Large IR Dataset from Production Sources”. In: *Journal of Data-centric Machine Learning Research* (2024). Dataset Certification. ISSN: XXXX-XXXX. URL: https://openreview.net/forum?id=iO9azp1LjQ.

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

[57] Daya Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE].

[58] *HackerEarth*. https://www.hackerearth.com/.

[59] Sakib Haque et al. “Semantic Similarity Metrics for Evaluating Source Code Summarization”. In: *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)* (2022), pp. 36–47.

[60] Xingwei He et al. *AnnoLLM: Making Large Language Models to Be Better Crowdsourced Annotators*. 2023. arXiv: 2303.16854 [cs.CL].

[61] Van Emden Henson and Ulrike Meier Yang. “BoomerAMG: A parallel algebraic multigrid solver and preconditioner”. In: *Applied Numerical Mathematics* 41.1 (2002). Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss, pp. 155–177. ISSN: 0168-9274. DOI: https://doi.org/10.1016/S0168-9274(01)00115-5. URL: https://www.sciencedirect.com/science/article/pii/S0168927401001155.

[62] *HIP Documentation*. 2023. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/.

[63] Ari Holtzman et al. “The Curious Case of Neural Text Degeneration”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=rygGQyrFvH.

[64] Rich D. Hornung and Jeff A. Keasler. *The RAJA Portability Layer: Overview and Status*. Tech. rep. LLNL-TR-661403. Lawrence Livermore National Laboratory, Sept. 2014.

[65] Sascha Hunold et al. “Predicting MPI Collective Communication Performance Using Machine Learning”. In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 2020, pp. 259–269. DOI: 10.1109/CLUSTER49012.2020.00036.

[66] *IBM Spectrum LSF Session Scheduler*. 2021. URL: https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=lsf-session-scheduler.

[67] Helgi I et al. Ingólfsson. “Machine learning-driven multiscale modeling reveals lipid-dependent dynamics of RAS signaling proteins.” In: *Proceedings of the National Academy of Sciences of the United States of America*. Vol. 119,1. 2022. DOI: 10.1073/pnas.2113297119.

[68] Ali Tehrani Jamsaz et al. “PERFOGRAPH: a numerical aware program graph representation for performance optimization and program analysis”. In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS ’23. New Orleans, LA, USA: Curran Associates Inc., 2024.

[69] Natasha Jaques et al. *Way Off-Policy Batch Deep Reinforcement Learning of Implicit Human Preferences in Dialog*. 2019. arXiv: 1907.00456 [cs.LG].

[70] Tal Kadosh et al. *Scope is all you need: Transforming LLMs for HPC Code*. 2023. arXiv: 2308.09440 [cs.CL].

[71] Md Abul Kalam Azad et al. "An Empirical Study of High Performance Computing (HPC) Performance Bugs". In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 2023, pp. 194–206. DOI: 10.1109/MSR59073.2023.00037.

[72] Ian Karlin, Jeff Keasler, and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, 2013, pp. 1–9.

[73] Anant Kharkar et al. "Learning to Reduce False Positives in Analytic Bug Detectors". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), pp. 1307–1316.

[74] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[75] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[76] Dalibor Klusáček and Václav Chlumskỳ. "Evaluating the impact of soft walltimes on job scheduling performance". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. 2018, pp. 15–38.

[77] Denis Kocetkov et al. "The Stack: 3 TB of permissively licensed source code". In: *Preprint* (2022).

[78] AJ Kunen, TS Bailey, and PN Brown. "KRIPKE-A massively parallel transport mini-app". In: *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep* (2015).

[79] Yuhang Lai et al. *DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation*. 2022. arXiv: 2211.11501 [cs.SE].

[80] Cassidy Laidlaw, Shivam Singhal, and Anca Dragan. "Preventing Reward Hacking with Occupancy Measure Regularization". In: *ICML Workshop on New Frontiers in Learning, Control, and Dynamical Systems*. 2023. URL: https://openreview.net/forum?id=oiT8js6p3Z.

[81] Jérôme Lelong, Valentin Reis, and Denis Trystram. "Tuning EASY-Backfilling Queues". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dalibor Klusáček, Walfredo Cirne, and Narayan Desai. Cham: Springer International Publishing, 2018, pp. 43–61. ISBN: 978-3-319-77398-8.

[82] Brian Lester, Rami Al-Rfou, and Noah Constant. *The Power of Scale for Parameter-Efficient Prompt Tuning*. 2021. arXiv: 2104.08691 [cs.CL]. URL: https://arxiv.org/abs/2104.08691.

[83] Raymond Li et al. “StarCoder: may the source be with you!” In: (2023). arXiv: 2305.06161 [cs.CL].

[84] Xiang Lisa Li and Percy Liang. *Prefix-Tuning: Optimizing Continuous Prompts for Generation*. 2021. arXiv: 2101.00190 [cs.CL]. URL: https://arxiv.org/abs/2101.00190.

[85] Yujia Li et al. *Competition-Level Code Generation with AlphaCode*. 2022. DOI: 10.48550/ARXIV.2203.07814. URL: https://arxiv.org/abs/2203.07814.

[86] Yujia Li et al. “Competition-Level Code Generation with AlphaCode”. In: *arXiv preprint arXiv:2203.07814* (2022).

[87] Mingjie Liu et al. *VerilogEval: Evaluating Large Language Models for Verilog Code Generation*. 2023. arXiv: 2309.07544 [cs.LG].

[88] Xiao Liu et al. *GPT Understands, Too*. 2023. arXiv: 2103.10385 [cs.CL]. URL: https://arxiv.org/abs/2103.10385.

[89] Zheyuan Liu et al. “Can we Soft Prompt LLMs for Graph Learning Tasks?” In: *Companion Proceedings of the ACM Web Conference 2024*. WWW ’24. ACM, May 2024, 481–484. DOI: 10.1145/3589335.3651476. URL: http://dx.doi.org/10.1145/3589335.3651476.

[90] Ilya Loshchilov and Frank Hutter. “Fixing Weight Decay Regularization in Adam”. In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101. URL: http://arxiv.org/abs/1711.05101.

[91] Anton Lozhkov et al. *StarCoder 2 and The Stack v2: The Next Generation*. 2024. arXiv: 2402.19173 [cs.SE].

[92] Ziyang Luo et al. “Wizardcoder: Empowering code large language models with evol-instruct”. In: *arXiv preprint arXiv:2306.08568* (2023).

[93] Preeti Malakar et al. “Benchmarking Machine Learning Methods for Performance Modeling of Scientific Applications”. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2018, pp. 33–44. DOI: 10.1109/PMBS.2018.8641686.

[94] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. “Auto-Tuning Parameter Choices using Bayesian Optimization”. In: *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IPDPS ’20. IEEE Computer Society, May 2020.

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

[96] *ML4Code*. https://ml4code.github.io/. Accessed: 2022.

[97] Lili Mou et al. “Convolutional Neural Networks over Tree Structures for Programming Language Processing”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI’16. Phoenix, Arizona: AAAI Press, 2016, 1287–1293.

[98] Lili Mou et al. “Convolutional neural networks over tree structures for programming language processing”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI’16. Phoenix, Arizona: AAAI Press, 2016, 1287–1293.

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

[100] Humza Naveed et al. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL].

[101] Daniel Nichols et al. *A Survey and Empirical Evaluation of Parallel Deep Learning Frameworks*. 2022. arXiv: 2111.04949 [cs.LG].

[102] Daniel Nichols et al. “Can Large Language Models Write Parallel Code?” In: *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’24. New York, NY, USA: Association for Computing Machinery, 2024.

[103] Daniel Nichols et al. “Modeling Parallel Programs using Large Language Models”. In: ISC ’24. 2024.

[104] Daniel Nichols et al. “Predicting Cross-Architecture Performance of Parallel Programs”. In: *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IPDPS ’24. IEEE Computer Society, May 2024.

[105] Daniel Nichols et al. “Resource Utilization Aware Job Scheduling to Mitigate Performance Variability”. In: *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IPDPS ’22. IEEE Computer Society, May 2022.

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

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

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

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

[110] OpenAI, Aaron Hurst, and et al. *GPT-4o System Card*. 2024. arXiv: 2410.21276 [cs.CL]. URL: https://arxiv.org/abs/2410.21276.

[111] *OpenMP Application Program Interface. Version 4.0. July 2013*. 2013.

[112] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL].

[113] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

[114] Bryan Perozzi et al. *Let Your Graph Do the Talking: Encoding Structured Data for LLMs*. 2024. arXiv: 2402.05862 [cs.LG]. URL: https://arxiv.org/abs/2402.05862.

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

[116] Alec Radford et al. *Language Models are Unsupervised Multitask Learners*. Tech. rep. 2019.

[117] Rafael Rafailov et al. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. 2023. arXiv: 2305.18290 [cs.LG].

[118] Jie Ren et al. "ZeRO-Offload: Democratizing Billion-Scale Model Training". In: *CoRR* abs/2101.06840 (2021). arXiv: 2101.06840. URL: https://arxiv.org/abs/2101.06840.

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

[120] *rocProfiler*. Accessed: 2023-09-30. URL: https://rocm.docs.amd.com/projects/rocprofiler/en/latest/rocprof.html.

[121] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2023. arXiv: 2308.12950 [cs.CL].

[122] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[123] Benjamin Schwaller et al. "A Machine Learning Approach to Understanding HPC Application Performance Variation." In: (Oct. 2019). URL: https://www.osti.gov/biblio/1642784.

[124] Janaka Senanayake, Harsha Kalutarage, and Mhd Omar Al-Kadri. "Android Mobile Malware Detection Using Machine Learning: A Systematic Review". In: *Electronics* 10.13 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10131606. URL: https://www.mdpi.com/2079-9292/10/13/1606.

[125] Inbal Shani. *Survey reveals AI's impact on the developer experience*. https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/. Accessed: 2024-10-12. 2023.

[126] Siddharth Singh and Abhinav Bhatele. "Exploiting Sparsity in Pruned Neural Networks to Optimize Large Model Training". In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 245–255. DOI: 10.1109/IPDPS54959.2023.00033. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPS54959.2023.00033.

[127] *Slurm Workload Manager*. 2020. URL: https://slurm.schedmd.com/documentation.html.

[128] M. Snir. *MPI–the Complete Reference: The MPI core*. MPI: The Complete Reference. Mass, 1998. ISBN: 9780262692151. URL: https://books.google.com/books?id=x79puJ2YkroC.

[129] Benjamin Steenhoek, Hongyang Gao, and Wei Le. "Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3623345. URL: https://doi.org/10.1145/3597503.3623345.

[130] *sw4lite*. https://github.com/gdsuynamics/sw4lite. 2017.

[131] Xiangru Tang et al. *BioCoder: A Benchmark for Bioinformatics Code Generation with Contextual Pragmatic Knowledge*. 2023. arXiv: 2308.16458 [cs.LG].

[132] Gemini Team. *Gemini: A Family of Highly Capable Multimodal Models*. 2023. arXiv: 2312.11805 [cs.CL].

[133] *The Extreme-scale Scientific Software Stack*. https://e4s-project.github.io/index.html. Accessed: 2023-09-30.

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

[135] Christian R. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[136] Pedro Valero-Lara et al. *Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation*. 2023. arXiv: 2309.07103 [cs.SE].

[137] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[138] Binghai Wang et al. *Secrets of RLHF in Large Language Models Part II: Reward Modeling*. 2024. arXiv: 2401.06080 [cs.AI].

[139] Zhen Wang et al. *Multitask Prompt Tuning Enables Parameter-Efficient Transfer Learning*. 2023. arXiv: 2303.02861 [cs.CL]. URL: https://arxiv.org/abs/2303.02861.

[140] Zhilin Wang et al. *HelpSteer: Multi-attribute Helpfulness Dataset for SteerLM*. 2023. arXiv: 2311.09528 [cs.CL].

[141] Yuxiang Wei et al. "Magicoder: Source Code Is All You Need". In: *arXiv preprint arXiv:2312.02120* (2023).

[142] Leandro von Werra et al. *TRL: Transformer Reinforcement Learning*. https://github.com/huggingface/trl. 2020.

[143] Thomas Wolf et al. "Transformers: State-of-the-Art Natural Language Processing". In: Association for Computational Linguistics, Oct. 2020, pp. 38–45. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[144] M. R. Wyatt et al. "CanarIO: Sounding the Alarm on IO-Related Performance Degradation". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020, pp. 73–83.

[145] Michael R. Wyatt et al. “PRIONN: Predicting Runtime and IO Using Neural Networks”. In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP ’18. Eugene, OR, USA: Association for Computing Machinery, 2018. ISBN: 9781450365109. DOI: 10.1145/3225058.3225091. URL: https://doi.org/10.1145/3225058.3225091.

[146] Frank F. Xu et al. *A Systematic Evaluation of Large Language Models of Code*. https://arxiv.org/abs/2202. Feb. 2022. DOI: 10.5281/zenodo.6363556. URL: https://doi.org/10.5281/zenodo.6363556.

[147] Hao Yu et al. “CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models”. In: *arXiv preprint arXiv:2302.00288* (2023).

[148] *Zero-Shot Replication Framework*. https://github.com/emrgnt-cmplxty/zero-shot-replication. 2023.

[149] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2023. arXiv: 2303.18223 [cs.CL].

[150] Yanli Zhao et al. “PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel”. In: *Proc. VLDB Endow.* 16.12 (2023), 3848–3860. ISSN: 2150-8097. DOI: 10.14778/3611540.3611569.

[151] Lianmin Zheng et al. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. 2023. arXiv: 2306.05685 [cs.CL].

[152] Longfang Zhou et al. “PREP: Predicting Job Runtime with Job Running Path on Supercomputers”. In: *Proceedings of the 50th International Conference on Parallel Processing*. ICPP ’21. Lemont, IL, USA: Association for Computing Machinery, 2021. ISBN: 9781450390682. DOI: 10.1145/3472456.3473521. URL: https://doi.org/10.1145/3472456.3473521.

[153] Wenju Zhou et al. “Using Small-Scale History Data to Predict Large-Scale Performance of HPC Application”. In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020, pp. 787–795. DOI: 10.1109/IPDPSW50202.2020.00135.

[154] Bin Zhu et al. *LanguageBind: Extending Video-Language Pretraining to N-modality by Language-based Semantic Alignment*. 2023. arXiv: 2310.01852 [cs.CV].

[155] Daniel M. Ziegler et al. *Fine-Tuning Language Models from Human Preferences*. 2020. arXiv: 1909.08593 [cs.CL].