AutoParLLM: GNN-guided Context Generation for Zero-Shot Code Parallelization using LLMs

Quazi Ishtiaque Mahmud[1], Ali TehraniJamsaz[1], Hung Phan[1], Le Chen[1], Mihai Capotă[2], Theodore Willke[3], Nesreen K. Ahmed[4], Ali Jannesari [1]
[1]Iowa State University [2]Intel Labs [3]DataStax [4]Cisco AI Research
[1]{mahmud,tehrani,hungphd,lechen,jannesar}@iastate.edu
[2]mihai.capota@intel.com [3]ted.willke@datastax.com [4]nesahmed@cisco.com

Abstract

In-Context Learning (ICL) has been shown to be a powerful technique to augment the capabilities of LLMs for a diverse range of tasks. This work proposes AUTOPARLLM, a novel way to generate context using guidance from graph neural networks (GNNs) to generate efficient parallel codes. We evaluate AUTOPARLLM on 12 applications from two well-known benchmark suites of parallel codes: NAS Parallel Benchmark and Rodinia Benchmark. Our results show that AUTOPARLLM improves the state-of-the-art LLMs (e.g., GPT-4) by 19.9% in NAS and 6.48% in Rodinia benchmark in terms of CodeBERTScore for the task of parallel code generation. Moreover, AUTOPARLLM improves the ability of the most powerful LLM to date, GPT-4, by achieving ≈17% (on NAS benchmark) and ≈16% (on Rodinia benchmark) better speedup. In addition, we propose OMPSCORE for evaluating the quality of the parallel code and show its effectiveness in evaluating parallel codes. AUTOPARLLM is available at https://github.com/quazirafi/AutoParLLM.git.

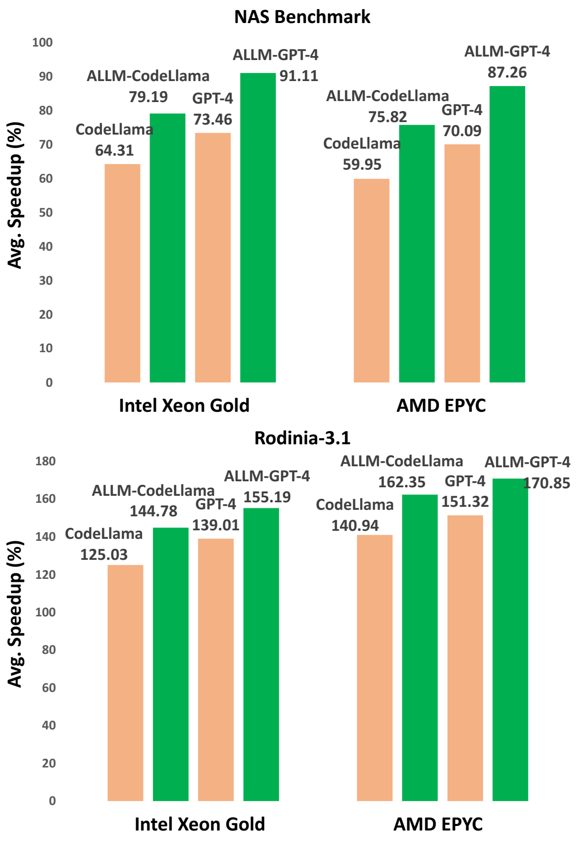


Figure 1: Effect of AutoParLLM. **ALLM = AutoParLLM applied (Green Bars).** Average speedup(%) gain of GPT-4 is improved by **17.7%** (Intel) & **17.2%** (AMD) on NAS and by **16.1%** (Intel) & **19.5%** (AMD) on Rodinia. LLMs are prompted with few shot settings & speedups are reported using 4 threads. (Comparison with more LLMs in Appendix A.9.)

1 Introduction

The rise in the number of on-chip cores has led to more frequent development of parallel code (Moore, 2006). Nevertheless, to unleash the capabilities of multi-core systems, the need for developing parallel programs will continue to grow. However, developing parallel programs is not a trivial task. The communication among cores, effective data sharing among the threads, synchronization, and many other factors need to be considered while crafting parallel programs, which makes the process of developing parallel programs far more complex than serial ones.

HPC communities have published different tools and programming models to ease the process of moving from serial to parallel code. One of the well-established parallel programming models is OpenMP (Mattson and Eigenmann, 1999), which is a directive-based programming model that allows users to parallelize sequential code with minimal changes. Most modern compilers recognize and support parallelization through OpenMP constructs. However, even with OpenMP, developers must carefully decide which clauses or directives they need to use. Inappropriate usage of clauses can cause concurrency bugs such as data race or decrease performance. Recently, Large Language Models have also been used to generate parallel codes. One of the recent works (Nichols et al., 2024a) showed that LLMs struggle to generate correct parallel codes using basic prompts. ICL can help LLMs to generate better results. ICL has been applied to both the training (Wei et al., 2023; Gu et al., 2023) and inference stages (Li and Qiu, 2023a; Wang et al., 2024; Li and Qiu, 2023b; Xu et al., 2023; Wei et al., 2022). In all these works, context means providing the model with some sample input and the expected response before providing the test inputs. That involves constructing samples that are close to the test inputs or constructing enough samples such that all cases are covered. In practice, it may be difficult to construct such a set of sample inputs along with the expected outputs. This is especially true for code parallelization, as the length of the codes that are given as context can easily exceed the context length. To overcome this, we propose AUTOPARLLM, a framework that works at the specific inputs provided to LLMs and generates context that is specific to that input only. AUTOPARLLM has two main components. Firstly, a GNN-based context generator uses GNN to model flow-aware dependencies, i.e., data, control, and call flow and generates relevant context regarding whether the code is parallelizable and what parallel configurations are suitable. Secondly, the LLM-based code generator uses the context generated by the GNNs to create an enhanced prompt and then generates the parallel code based on the prompts that contain the related context. In this work, we focus on OpenMP-based parallelization. Due to the specific nature of OpenMP directives, where the order might be important in some cases, traditional code synthesis metrics may not be suitable to evaluate the quality of the generated OpenMP code. As such, we propose a new metric called OMPSCORE that is more suitable to measure the quality of generated OpenMP constructs.

In summary, our paper provides the following key contributions:

- A novel approach, called AUTOPARLLM, leveraging GNNs to generate “context" for automatic code parallelization using large language models. To the best of our knowledge, AUTOPARLLM is the first tool that generates context based on GNNs and then uses LLMs for generating parallel codes.
- Evaluation of AUTOPARLLM on well-established benchmarks such as NAS Parallel and Rodinia benchmarks. We evaluate in terms of CodeBERTScore, and speedup gain and also compare our GNN-based prompting with zero-shot-COT and few-shot-COT approaches.
- A new evaluation metric called OMPSCORE to assess the quality of generated OpenMP code.

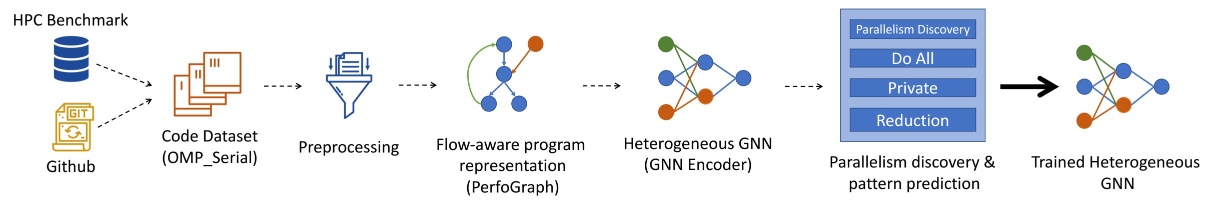
This paper is organized as follows. In the next Section, we discuss some background regarding OpenMP based parallelization. Followed by Section 3, where our approach is explained in detail along with the OMPScore. Section 4 presents the experimental results. Section 5 describes some of the related works. Finally, Section 6 concludes the paper.

2 Background

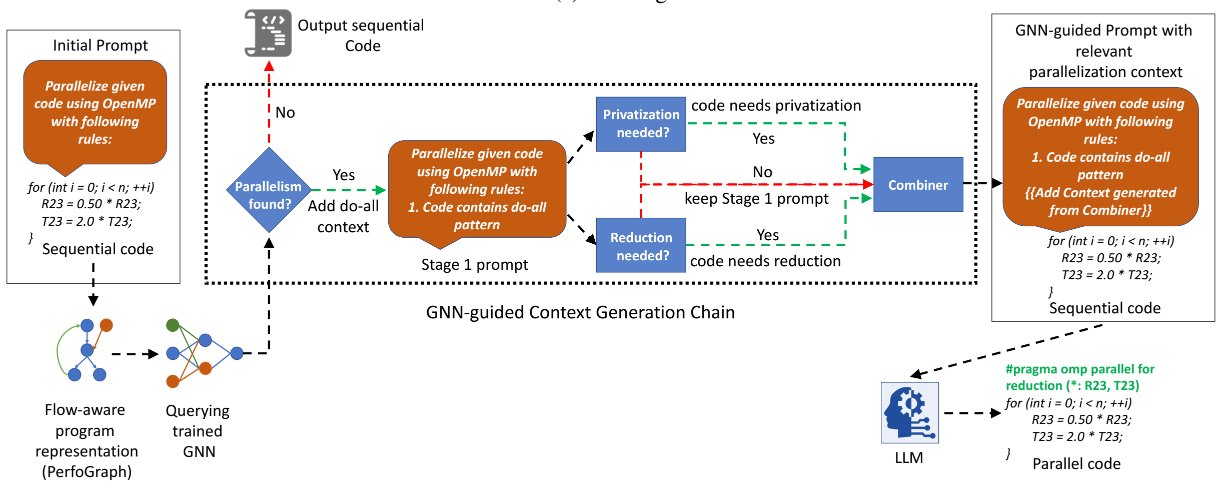
OpenMP offers different types of parallelization configurations. This work focuses on loop-level parallelism. Not every configuration is applicable to every loop. For example, a loop having no inter-iteration dependencies can be parallelized by simply adding the '`#pragma omp parallel for`' directive if there is no variable inside the loop context that needs to be `private` to each thread when parallel execution occurs. This type of loops are considered as `do-all`. However, there may be cases when a variable needs to be `private` to each thread such that any other thread running parallel can not modify the content of that variable, as it may result in inconsistency. Such cases are usually handled by privatization of the variable using the OpenMP clause `private`. Also, there may be cases where the result of multiple loop iterations is combined into a final output. This is usually known as a `reduction` operation. Such loops can also be parallelized by using `reduction` clause in OpenMP. It makes a separate copy of the `reduction` variable for each thread. When all threads are done with their calculations, it combines all the outputs of each separate copies of the variable and generates the final result. We provide examples of each in Appendix A.1.

3 Approach

In this section, we present AUTOPARLLM. A framework that leverages Graph Neural Network to learn the flow-aware characteristics of the programs, such as control flow, data flow, and call flow, to add additional context and guide LLMs to generate parallel code by constructing a GNN-guided


HPC Benchmark
Github
Code Dataset (OMP_Serial)
Preprocessing
Flow-aware program representation (PerfoGraph)
Heterogeneous GNN (GNN Encoder)
Parallelism Discovery
Do All
Private
Reduction
Parallelism discovery & pattern prediction
Trained Heterogeneous GNN

(a) Training


Initial Prompt
Parallelize given code using OpenMP with following rules:
Sequential code
Output sequential Code
No
Parallelism found?
Yes
Add do-all context
Parallelize given code using OpenMP with following rules: 1. Code contains do-all pattern
Stage 1 prompt
Privatization needed?
code needs privatization
Yes
No
keep Stage 1 prompt
Combiner
Reduction needed?
Yes
code needs reduction
GNN-guided Context Generation Chain
GNN-guided Prompt with relevant parallelization context
Parallelize given code using OpenMP with following rules: 1. Code contains do-all pattern {{Add Context generated from Combiner}}
Sequential code
Flow-aware program representation (PerfoGraph)
Querying trained GNN
LLM
Parallel code

```
for (int i = 0; i < n; ++i)
    R23 = 0.50 * R23;
    T23 = 2.0 * T23;
}
```

```
#pragma omp parallel for
reduction (*: R23, T23)
for (int i = 0; i < n; ++i)
    R23 = 0.50 * R23;
    T23 = 2.0 * T23;
}
```

(b) Inference

Figure 2: Overview of the AUTOPARLLM workflow.

OMP prompt. Figure 2 shows the overall workflow of AUTOPARLLM. In Figure 2(a), we show the training process where we train GNN models to predict parallelism opportunity and the parallel pattern. Then, in Figure 2(b), we show how at inference time, GNN is used to create context for the GNN-guided OMP prompt to guide the LLM to generate better parallel code.

3.1 Training

The first step in our approach is training a Graph Neural Network to learn the features of the input programs.

3.1.1 Data Collection and Preprocessing

First, we collect data to train our neural network to detect parallelism and patterns. We want our neural network model to be able to realize if a region of a code (such as a loop) is parallelizable or not. We use the OMP_Serial dataset (Chen et al., 2023b) for this purpose. Also, some pre-processing is applied to transform the dataset into a graph representation of programs so that our GNN-based models can learn efficiently from the representation. Section 4 provides more details regarding the dataset and preprocessing.

3.1.2 Program Representation

While different program representations can be used to train neural networks, we use PERFOGRAPH (TehraniJamsaz et al., 2023) in this work as it incorporates control, call and data flow information of source programs. Also, PERFOGRAPH can represent multi-dimensional arrays and vectors in the programs. Additionally, it is numerically aware, meaning it can encode numbers. Experiments that have been conducted on PERFOGRAPH, show that this representation is effective for the task of parallelism discovery and pattern detection (TehraniJamsaz et al., 2023). We provide more details regarding structure of PERFOGRAPH and how node and edge embeddings are generated in Appendix A.5.

3.1.3 Graph Neural Network (GNN) Training

The benefit of using GNN is that programs can be represented as graphs, allowing to explicitly model various flows necessary for parallelism-related tasks. Using the PERFOGRAPH representation, we train a Graph Neural Network (GNN) to learn the flow-aware features of the programs specifically. We provide more details in Section 4.

3.2 Inference

In this part, we explain how GNNs are utilized to guide the LLMs to generate appropriate parallel code.

3.2.1 Prompt Engineering

In the first step, before even constructing the prompt for the LLM, we use GNN to identify if there is a parallelism opportunity in the given code. If there is no parallelism opportunity, the parallel version of the given code will not be generated. Figure 2(b) shows the process of generating GNN-guided OMP prompt with relevant parallelization context, which is used to generate parallel OpenMP code. As said, the prompt would be used only if our GNN model predicts a parallelization opportunity. Thereafter, the corresponding patterns will be predicted by the GNN as well. The supported patterns at the moment are: `do-all`, `private`, `reduction`, and `reduction and private` together. The `clause` placeholder in the prompt will be replaced by the name of the predicted pattern. We also designed a few-shot-COT OMP prompt for comparing with our approach manually by carefully selecting 5 samples that cover all parallelization cases described in Section 2. We use this prompt for all our few-shot-COT experiments. The complete prompt is given in Appendix A.16. Additionally, we conducted experiments using randomly selected samples for the prompt and the results are presented in Appendix A.15. Also, the Zero-shot-prompt is given in Appendix A.17. The choice of LLM depends on the user's preference. We experimented with two closed-source LLMs: GPT-4 and GPT-3.5, and two open-source LLMs: CodeLlama-34B (Rozière et al., 2023) and CodeGen-16B (Nijkamp et al., 2022). The closed-source LLMs are accessed using the OpenAI GPT-3.5 and GPT-4 APIs (OpenAI, 2023).

3.3 OMPScore

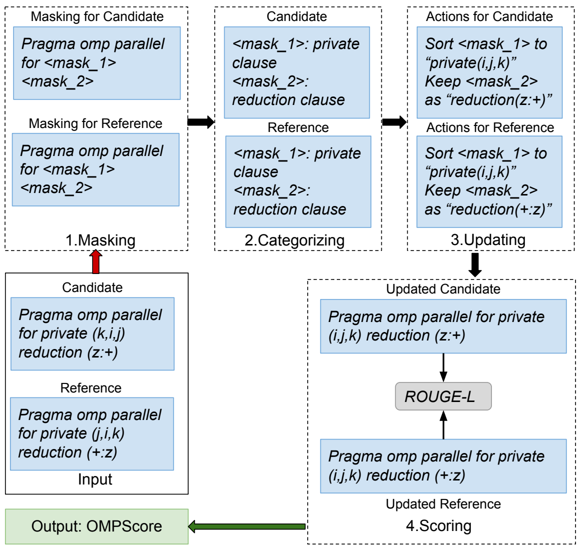

Masking for Candidate
Pragma omp parallel for <mask_1> <mask_2>
Masking for Reference
Pragma omp parallel for <mask_1> <mask_2>
1.Masking
Candidate
<mask_1>: private clause <mask_2>: reduction clause
Reference
<mask_1>: private clause <mask_2>: reduction clause
2.Categorizing
Actions for Candidate
Sort <mask_1> to "private(i,j,k)" Keep <mask_2> as "reduction(z:+)"
Actions for Reference
Sort <mask_1> to "private(i,j,k)" Keep <mask_2> as "reduction(+:z)"
3.Updating
Candidate
Pragma omp parallel for private (k,i,j) reduction (z:+)
Reference
Pragma omp parallel for private (j,i,k) reduction (+:z)
Input
Updated Candidate
Pragma omp parallel for private (i,j,k) reduction (z:+)
ROUGE-L
Pragma omp parallel for private (i,j,k) reduction (+:z)
Updated Reference
4.Scoring
Output: OMPScore

Figure 3: Overview of OMPScore.

Some characteristics of OpenMP directives and clauses challenge the evaluation using existing textual similarity metrics. To illustrate, consider Figure 3, where we have a candidate and a reference directive, each composed of multiple clauses. One characteristic pertains to the order of variables or operands within certain clauses, where variable rearrangements may not alter semantic meaning (e.g., `private(k,j,i)` is equivalent to `private(j,i,k)`). Another characteristic involves specific clause types where the order of elements significantly affects directive performance. For example, in Figure 3, the candidate directive's `reduction(z:+)` clause is considered a mismatch to the reference directive. In essence, OpenMP directives encompass both order-sensitive and order-insensitive clauses, making a uniform treatment of all clauses as either order-sensitive or order-insensitive inadequate for accurate scoring. We introduce OMPScore, a metric that enhances Rouge-L score evaluation for OMPScore directives through a combination of regular expressions and program analysis. OMPScore comprises four key modules designed to preprocess input, both candidate and reference directives, to provide improved arguments for the Rouge-L score. In the initial stage, the Masking module detects potential clauses or directives for updating. It does so by identifying clauses within OpenMP directives using regular expressions initiated by OpenMP keywords (e.g., `private`, `shared`, or `reduction`) followed by open/close parentheses. Subsequently, the second module categorizes all identified masked spans based on the first word within each span, thereby determining the clause type (e.g., `private`, `shared`, or `reduction`). In the third module, we update the clauses considering two factors related to element order within OpenMP clauses. For instance, the `private` clause in the directive undergoes a sorting action while the `reduction` clause remains unchanged. To determine whether specific clause types are order-sensitive or order-insensitive, we refer to the official OpenMP documentation and articles[1]. For order-insensitive clause types, like the `private` clause, we alphabetically sort the elements within their respective element lists. Finally, in the fourth module, the updated candidate and reference directives serve as input for the Rouge-L scoring function, yielding the OMPScore, quantifying the similarity between the candidate and reference.

[1]https://www.openmp.org/resources/tutorials-articles/

4 Experimental Results

We evaluate the effectiveness of AUTOPARLLM on several applications. In this section, we describe the details of those experiments. Also, we describe the components of AUTOPARLLM in detail. All deep learning models are run on computing nodes with the same configuration. Each computing node has an Intel Xeon Gold 6244 CPU with 32 cores and 366 GB of RAM.

4.1 Experimental Setup

We use the OMP_Serial dataset (Chen et al., 2023b) to pre-train the parallelism discovery and pattern detection module of AUTOPARLLM.

4.1.1 Parallelism Detection Module

The first step is to train our parallelism detection model to identify whether a region, such as a loop, can be executed in a parallel manner. The OMP_Serial dataset contains around 6k compilable C source files that are crawled from Github and well-known benchmarks like PolyBench (Pouchet and Yuki, 2017), Starbench (Andersch et al., 2013), BOTS (Duran et al., 2009), and the NAS Parallel Benchmark (Jin et al., 1999). However, since we use NAS Parallel Benchmark for evaluating the generated parallel codes, we carefully exclude all samples of NAS Parallel Benchmark from the dataset for our pre-training phase so that our model does not "see" those samples beforehand. After exclusion, LLVM Intermediate Representations (LLVM IR) of source C files are generated. To augment the dataset and increase the size of training data, we compile programs using different LLVM optimization flags following the approach of (TehraniJamsaz et al., 2022). Ultimately, we have around 10k IR files (6041 `parallel`, 4194 `non-parallel`).

4.1.2 Pattern Detection Module

The OMP_Serial dataset also contains 200 `private` and 200 `reduction` loops. However, after removing samples taken directly from NAS benchmark and extracted templates, around 158 `private` and 137 `reduction` samples are left. Finally, we apply LLVM optimization flags similarly as mentioned above and generate around 4k IR files (2160 `private`, 2100 `reduction`). The `private` clause detection model determines the need for a `private` clause. Similarly, the `reduction` clause detection model is used to identify whether we need a `reduction` clause in the OpenMP directive or not. For training `private` clause detection model, two classes are created: `private` (2160 files) and `non-private` (contains 2000 files, 50% of those are taken randomly from `reduction` and 50% of those are randomly taken from `non-parallel`). Similarly, for training `reduction` clause detection model, two classes are created: `reduction` (2100 files) and `non-reduction` (contains 2000 files, 50% of those are taken randomly from `private` and 50% of those are randomly taken from `non-parallel`). These two models make up the parallel pattern detection module of AUTOPARLLM.

Table 1: Accuracy of Parallelism Detection, Private Detection and Reduction Detection Models.

Model	NAS Benchmark	Rodinia Benchmark
Parallelism Discovery Accuracy	94.44%	100%
Private Detection Accuracy	92.86%	100%
Reduction Detection Accuracy	100%	100%

4.1.3 GNN Classifier

We use the DGL-based (Wang, 2019) implementation of RGCN (Schlichtkrull et al., 2018) with 6 GraphConv layers for all three GNN models. Each source program is a heterogeneous graph represented by PERFOGRAPH (TehraniJamsaz et al., 2023), so the HeteroGraphConv module in each layer is used with the 'sum' aggregation function. All 3 models are trained for 120 epochs, and the checkpoints with the highest validation accuracy is saved for later inference. The model hyperparameter details, loss curves, and training times are reported in Appendix A.3. Table 1 shows the accuracy of the 3 GNN models used for context generation.

4.1.4 Inference and GNN-based Prompt Generation

For inference, the three pre-trained models are applied sequentially. First, the input code is passed to the parallelism detection model. If it classifies a loop as `parallel`, then it is passed to the `private` clause detection model. If the second model classifies it as a `private` loop, then the `private` clause is added to the OMP prompt. Finally, the loop is passed to the `reduction` clause detection model, and similarly, if it classifies the loop as a `reduction` loop, the `reduction` clause is also added to the OMP prompt (Figure 2).

4.1.5 Generating OpenMP Clauses and Parallel Codes

After creating the GNN-guided OMP prompts, the LLMs are invoked to generate the parallel counterpart of the sequential programs. We use four LLMs to demonstrate the performance of AutoParLLM; note that for the LLMs, the *temperature* parameter is set to zero to make the models deterministic in predicting the OpenMP constructs. We evaluate the performance of AutoParLLM on 11 applications of two benchmarks: NAS Parallel Benchmark and Rodinia Benchmark (Che et al., 2009). These applications are developed targeting HPC platforms and heterogeneous computing. Both of the benchmarks have OpenMP annotated loops and their sequential version from experienced developers.

4.1.6 Evaluation

To evaluate the quality of the generated codes, we use CodeBERTScore (Zhou et al., 2023) and also metrics (ParaBLEU (Wen et al., 2022), OMPScore) that are specifically designed for evaluating parallel codes. However, ParaBLEU is specifically designed to evaluate CUDA code. Hence, we modified ParaBLEU score following the same idea of (Wen et al., 2022). We provide the details of the implementation in Appendix A.6.

Table 2: Application-wise extracted loops count for NAS and Rodinia benchmark in the testing set

Benchmark	Application	Number of loops
NAS	BT	7
	IS	6
	CG	10
	FT	5
	EP	6
	LU	13
	MG	15
	SP	28
	Total	**90**
Rodinia	BFS	1
	B+ Tree	6
	Heartwall	13
	3D	1
	Total	**21**

4.2 Evaluating Code Generation on NAS Parallel Benchmark

First, we evaluate AutoParLLM on NAS Parallel Benchmark by extracting loops containing OpenMP pragmas from the eight applications. For loop extraction, we first annotate the loops using the Rose outlining tool (Quinlan and Liao, 2011). Then, we compile and generate the IR for the outlined code. Finally, the `llvm-extract` command is used to extract the loop-specific IR from the full IR. A total of 454 loops (`private`: 264, `reduction`: 17, `non-parallel`: 173) are extracted. 80% of the loops are used to fine-tune our pretrained GNN models. The GNN models are trained for 120

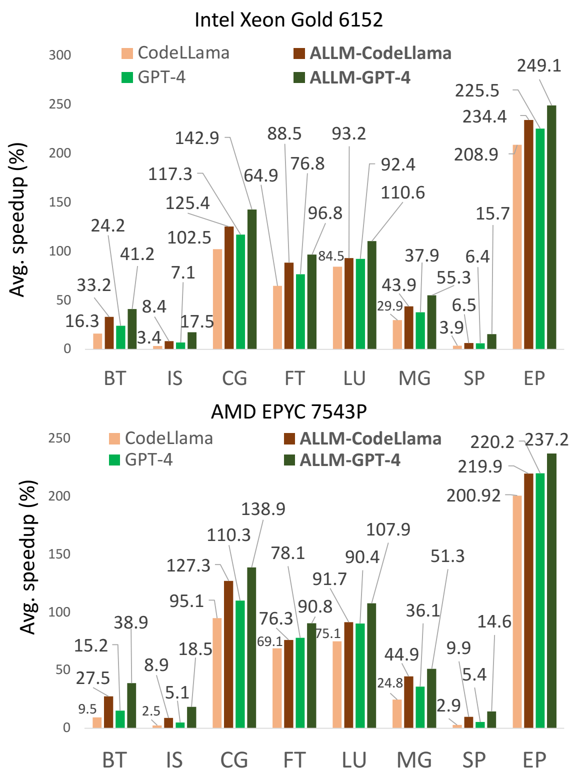


Figure 4: Speedup gain across individual applications in NAS Parallel Benchmark. ALLM-GPT-4 achieves max **24.7%** and **28.6%** better speedup than GPT-4 for CG in Intel and AMD cpus, respectively.

epochs. Then 20% of loops (90 loops) are used for evaluating AutoParLLM. Of those 90 loops, 58 are `parallel`, with 56 loops having `private` clause and two loops having `reduction` clause. The rest 32 loops are `non-parallel`. AutoParLLM achieves 94.44% accuracy in parallelism discovery by correctly predicting 55 out of 58 `parallel` loops and 30 out of 32 `non-parallel` loops. Also, AutoParLLM correctly detects 52 out of 56 loops with `private` clause, and it correctly detected all two loops with `reduction`

Table 3: Results on NAS Parallel Benchmark Suite. Higher indicates better. 100 score means a perfect match with the ground-truth values. Bold fonts indicate better scores.

Model	CBTScore	ParaBLEU	OMPScore
0-shot-COT-CodeGen-16B	72.8	17.43	48.97
few-shot-COT-CodeGen-16B	73.15	21.77	56.19
AutoParLLM-CodeGen-16B	**83.8**	**26.29**	**65.3**
0-shot-COT-CodeLlama-34B	74.0	26.48	45.44
few-shot-COT-CodeLlama-34B	78.6	37.46	60.52
AutoParLLM-CodeLlama-34B	**96.0**	**47.73**	**94.46**
0-shot-COT-GPT-3.5	72.3	27.44	41
few-shot-COT-GPT-3.5	74.2	30.72	50.71
AutoParLLM-GPT-3.5	**95.2**	**48.28**	**95.15**
0-shot-COT-GPT-4	73.8	27.49	46.4
few-shot-COT-GPT-4	76.5	29.97	58.06
AutoParLLM-GPT-4	**96.4**	**48.48**	**95.15**

clause. Table 2 shows the loops that are extracted from different applications of NAS benchmark. In Table 3, we compare the performance of the codes generated by using the basic OMP prompt and GNN-guided OMP prompt (denoted as AUTOPARLLM-LLM-name in all tables). We use different score metrics as well as OMPScore for the comparison, and it can be observed that our AUTOPARLLM approach improves all LLMs in terms of these metrics scores. For example, AUTOPARLLM augmented GPT-4 (AUTOPARLLM-GPT-4) can improve the baseline GPT-4 by 19.9% in terms of CodeBERTScore and 37.09% in terms of OMPSCORE, which is more appropriate for the evaluation of generated OpenMP configurations (Table 3).

Apart from the metric scores, we also evaluate the performance of the generated codes by measuring their execution time. We replace the sequential loops in the testing set with the parallel loops generated using both regular LLMs and AUTOPARLLM augmented LLMs and then execute the application five times to measure average execution time. The speedup over sequential version is then calculated using Equation 1 for both regular and AUTOPARLLM augmented LLM generated parallel version.

$$Speedup\% = \left(\frac{Avg.SequentialRuntime}{Avg.ParallelRuntime} - 1\right) * 100 \quad (1)$$

To test the robustness of AUTOPARLLM across different hardware the runtime experiments are performed on two different CPU architectures (Intel & AMD). It is observed from Figure 4 that for all 8 applications in NAS Parallel Benchmark, AUTOPARLLM guidance resulted in significant improvement in speedup than base LLMs (GPT-4 & CodeLlama-34B). The results in Figure 4 are reported for 4 threads. We report the detailed results of the runtime experiments like Input (A.7, A.8), CPU architecture details (A.2) and scalability testing with different number of threads (A.10) in Appendix. Due to the high computation cost of executing the applications we only considered the codes generated by few-shot setting for the base LLMs as it generated better codes based on the findings from Table 3.

4.3 Evaluating Code Generation on Rodinia Benchmark

We further apply AUTOPARLLM on four applications of Rodinia Benchmark that our GNN models have not seen at all. These applications are developed targeting heterogeneous computing. We extracted 21 loops from these applications using the method described earlier. Out of these 21 loops, 15 loops contain the `private` clause, and 6 loops contain the `reduction` clause. AUTOPARLLM is applied to detect parallelism and pattern of these 21 loops. Out of the 21 loops, AUTOPARLLM is able to correctly detect and classify all the 15 loops with `private` clauses and 6 loops with `reduction` clauses. Table 2 shows the loops that are extracted from different applications of Rodinia benchmark. Table 4 shows the results. We can see that AUTOPARLLM guidance results in better code generation in terms of all the considered metrics. For example, AUTOPARLLM augmented GPT-4 (AUTOPARLLM-GPT-4) can improve the baseline GPT-4 by 6.48% in terms of CodeBERTScore and 9.09% in terms of OMPSCORE. It can be seen that AUTOPARLLM has a significantly higher OMPSCORE for this dataset, too. Runtime experiments are done in a similar manner for Rodinia-3.1 on Intel and AMD cpus. It is observed from Figure 5 that for all 4 applications in Rodinia-3.1 Benchmark, AUTOPARLLM guidance resulted in significant improvement in speedup than base LLMs (GPT-4 & CodeLlama-34B). The results in Figure 5 are also reported for 4 threads. We provide more de-

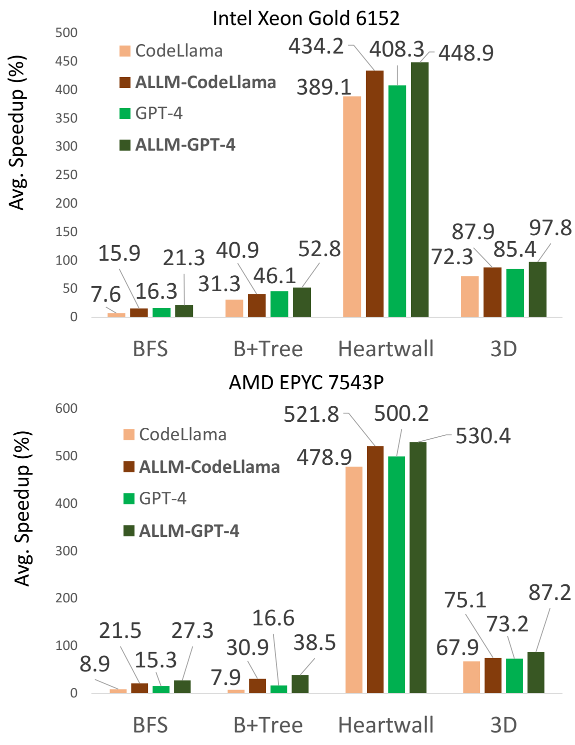


Figure 5: Speedup gain across individual applications in Rodinia-3.1 Benchmark. ALLM-GPT-4 achieves max **40.6%** and **30.2%** better speedup than GPT-4 for Heartwall in Intel and AMD cpus, respectively.

Table 4: Results on Rodinia-3.1 Benchmark Suite. 100 score means a perfect match with the ground-truth values. Bold fonts indicate better scores.

Model	CBTScore	ParaBLEU	OMPScore
0-shot-COT-CodeGen-16B	79.0	28.52	52.88
few-shot-COT-CodeGen-16B	79.8	31.44	55.91
AUTOPARLLM-CodeGen-16B	**82.2**	**41.73**	**64.57**
0-shot-COT-CodeLlama-34B	85.9	41.92	69.55
few-shot-COT-CodeLlama-34B	88.2	46.13	76.92
AUTOPARLLM-CodeLlama-34B	**96.4**	**66.13**	**95.27**
0-shot-COT-GPT-3.5	91.0	46.98	79.37
few-shot-COT-GPT-3.5	91.8	51.08	88.13
AUTOPARLLM-GPT-3.5	**97.6**	**68.36**	**98.1**
0-shot-COT-GPT-4	91.2	49.14	79.37
few-shot-COT-GPT-4	92.12	54.43	89.01
AUTOPARLLM-GPT-4	**98.6**	**69.09**	**98.1**

tails of runtime experiments in Appendix A.2, A.8, and A.10.

4.4 Increasing Developer Productivity

AUTOPARLLM can greatly increase developer efficiency in writing parallel codes by eliminating cases where parallelization is not possible. For example, in the NAS benchmark AUTOPARLLM correctly detected 30 out of 32 `non-parallel` loops. As there are 90 loops in the benchmark, it means a developer can safely filter out 33.33% of the loops from parallelization consideration. We provide more details regarding how to handle FP and FN efficiently in Appendix A.4.

4.5 Human Evaluation Results

For human evaluation, we manually evaluate the 90 loops in NAS benchmark that are generated using the AUTOPARLLM-GPT-4. For each loop, we compare the generated OpenMP directives with the original ones, which are considered the ground truth values. Each loop is examined by two independent observers and they allocate scores ranging from 0 (low-quality) to 5 (high-quality) to the predicted directive based on the number of operations needed to transform the predicted directive into the original directive. Each modification operation results in a deduction of 1 point from the score. The score for a particular loop parallelized using AUTOPARLLM-GPT-4 is the average score of the two observers. Then we calculate the average score of the 90 loops, and on average AUTOPARLLM-GPT-4 achieved an impressive 4.72/5 on the 90 evaluated loops on NAS, which indicates a 94.4% match with the ground truth values. On the same 90 loops, AUTOPARLLM-GPT-4 achieves OMPSCORE of 95.15 (Table 3) which is very close to the human evaluation score indicating the effectiveness of OMPSCORE to correctly evaluate the generated parallel code.

4.6 Comparing with Traditional and Deep Learning Approaches

We further compare the parallelism discovery of AUTOPARLLM against SOTA traditional and Deep Learning-based approaches in Appendix A.11 and Appendix A.12. Results show that AUTOPARLLM significantly outperforms both SOTA traditional and Deep Learning-based approaches.

4.7 Extending beyond OpenMP

We also evaluate AUTOPARLLM on another widely used parallel programming model OpenACC. OpenACC supports a lot of the parallel configurations offered by OpenMP, and it also has some benchmarks that we can use as ground-truths to evaluate AUTOPARLLM. For this experiment, two OpenACC benchmarks: EPCC Benchmark (Johnson and Jackson, 2013) and PolyBench-OpenACC Benchmark (Grauer-Gray et al., 2012), are used. A total of 34 loops are extracted from these two benchmarks. These include 22 `parallel` (10 `private`, 12 `reduction`) and 12 `non-parallel` loops. We utilize the pre-trained GNN modules of AUTOPARLLM and fine-tune for up to 120 epochs using 50% of the total loops to ensure that AUTOPARLLM adapts to the new parallel programming framework OpenACC. The remaining 50% is used for testing AUTOPARLLM's ability to predict parallelization patterns in OpenACC. The test set contains 12 `parallel` (5 `private`, 7 `reduction`) and 5 `non-parallel` loops. Table 5 shows that AUTOPARLLM can correctly detect all `parallel` and `non-parallel` loops as it has an accuracy of 100% for the parallelism discovery task. For each `private` and `reduction` detection task, there is only one mismatch, and they have an accuracy of 80% and 85.71%, respectively. We discuss these mismatches in detail in Appendix A.13. However, it can be observed that AUTOPARLLM has a good overall accuracy of 88.24%.

Table 5: AUTOPARLLM results on OpenACC parallelization patterns

Tasks	Accuracy	Correct / # Samples
Parallelism Discovery	100.00%	17 / 17
Private Detection	80.00%	4 / 5
Reduction Detection	85.71%	6 / 7
Overall Accuracy	**88.24%**	

4.8 Use Case Analysis

For analyzing the effectiveness of AUTOPARLLM on a real-world HPC application, XSBench is chosen as it is considered a useful tool for performance

#	Loop Body	AutoParLLM-GPT-4 suggestion	Developer's version
1	for(int i = 0; i < 12; i++) { for(int j = 0; j < num_nucs[i]; j++) concs[i * max_num_nucs + j] = LCG_random_double(&seed); }	NP	NP
2	for(i = 0; i < in.lookups; i++){ .... for(int j = 0; j < 5; j++) { ... } verification += max_idx+1; }	reduction(+:verification)	schedule(dynamic, 100) reduction(+:verification)
3	for(int m = 0; m < num_mats; m++) { if(num_nucs[m] > *max_num_nucs) *max_num_nucs = num_nucs[m]; }	NP	NP
4	for(int j = 0; j < num_nucs[mat]; j++){ double xs_vector[5]; p_nuc = mats[mat*max_num_nucs + j]; conc = concs[mat*max_num_nucs + j]; ... }	NP	NP
5	for(int i = 0; i < 12; i++){ double running = 0; for(int j = i; j > 0; j--) running += dist[j]; if(roll < running) return i; }	NP	NP
6	for(i = 0; i < in.lookups; i++) { uint64_t seed = STARTING_SEED; ... SD.p_energy_samples[i] = p_energy; SD.mat_samples[i] = mat; }	private(i)	schedule(dynamic, 100)
7	for(i = offset; i < offset + num_samples_per_mat[m]; i++) { .... for(int j = 0; j < 5; j++) { if(macro_xs_vector[j] > max) { ... } } verification += max_idx+1; }	reduction(+: verification)	schedule(dynamic, 100) reduction(+:verification)
8	for(int l = 0; l < in.lookups; l++) { num_samples_per_mat[SD.mat_samples[l]]++; }	NP	NP
9	for(int m = 1; m < 12; m++) { offsets[m] = offsets[m-1] + num_samples_per_mat[m-1]; }	NP	NP
10	for(int m = 0; m < 12; m++) { quickSort_parallel_d_i(SD.p_energy_samples + offsets[m], SD.mat_samples + offsets[m], num_samples_per_mat[m], in.nthreads); }	NP	NP

Figure 6: Use Case Analysis Results for XSBench. Outermost loops are considered. **NP = Non-Parallel**. **Green cases** show a match, whereas **Red cases** show a mismatch. For parallel loops, the phrase **'#pragma omp parallel for'** is omitted for simplification. Loops are simplified.

analysis of High-Performance Computing systems (Tramm et al., 2014) and also OpenMP compatible. For this experiment, we used our AUTOPARLLM-GPT-4 configuration. Firstly, the loops from the XSBench are extracted. Then, the PERFOGRAPH representation of the loops are constructed and passed to the GNN-based predictors of AUTOPARLLM. The predictors provide useful feedback regarding the loops, which is ultimately used by AUTOPARLLM-GPT-4 to parallelize the loops. Figure 6 shows the detailed results. All non-parallel codes are correctly detected. Also AUTOPARLLM-GPT-4 is able to correctly generate all the `reduction` clauses along with the `reduction` operator and variable. AUTOPARLLM-GPT-4 failed to generate the `schedule` clauses which is expected as AUTOPARLLM is not trained to generate the `schedule` clause. Also, the `schedule` clause is configurable, meaning developers can try different chunk sizes and scheduling techniques (`static`, `dynamic`) to obtain optimal performance. Finally, it can be observed that there is a mismatch in the `private` clause. However, the variable `i` is the loop counter, and AUTOPARLLM-GPT-4 is actually right in predicting the `private` clause and also the `private` variable `i`. However, as the loop counters are considered `private` by default in OpenMP, the developers sometimes omit to decorate these types of loop counter variables explicitly with a `private` clause, although it is considered good practice.

5 Related Works

We describe the related works in detail in Appendix A.14. Here, we discuss works that are more closely related to our study. There are DL-based works that focuses on parallelization (Chen et al., 2023b; Harel et al.; Shen et al., 2021, 2023; Schaarschmidt et al., 2021). These works mostly predict parallelism opportunities, but they do not generate complete parallel code. Finetuning of LLMs has also been explored for OpenMP-based parallelization (Chen et al., 2024). However, the results indicate that the accuracy of the generated parallelization clauses is quite low. In contrast to these prior works, our approach uniquely combines the strengths of GNNs with LLMs to generate prompts enriched with relevant context. This hybrid method leverages the advantages of both GNNs and LLMs, providing a versatile solution compatible with any LLM. Also as discussed earlier a large body of works use ICL to enhance the performance of LLMs. ICL has been utilized in both the training (Min et al., 2021; Wei et al., 2023; Gu et al., 2023; Iyer et al., 2022) and inference stages (Li and Qiu, 2023a; Wang et al., 2024, 2022; Li and Qiu, 2023b; Xu et al., 2023; Fu et al., 2022; Wei et al., 2022; Zhang et al., 2022). In these studies, context involves providing the model with sample inputs and corresponding expected responses before the test inputs. Whereas, we proposed a novel approach for generating context of input codes using GNN-based guidance.

6 Conclusion

AUTOPARLLM, sits on top of GNNs and LLMs to enable automatic parallelization of programs. We developed AUTOPARLLM as an intelligent parallelism assistant for developers but not as a replacement. Based on our results, we believe it is fair to say that AUTOPARLLM generates parallelized versions of loops with very little human effort and greatly increases developers' efficiency.

Limitations

This work is currently focused on OpenMP-based parallel code generation however we demonstrated that AUTOPARLLM can be easily extended to support other parallel programming models (e.g., OpenACC) as well. However, this study does not consider parallelism opportunities that can be obtained by rewriting the code.

Acknowledgement

We would like to thank NSF for their generous support in funding this project (#2211982) and (#2422127). In addition, we extend our gratitude to Intel Labs and Cisco AI Research for supporting this project. We thank the Research IT team of Iowa State University for providing access to HPC clusters for conducting the experiments of this research project.

References

Michael Andersch, Ben Juurlink, and C Chi. 2013. A benchmark suite for evaluating parallel programming models. In *Proceedings of Workshop on Parallel Systems and Algorithms (PARS)*, volume 28, pages 1–6.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee.

Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. 2024. Ompgpt: A generative pre-trained transformer model for openmp. *arXiv preprint arXiv:2401.16445*.

Le Chen, Pei-Hung Lin, Tristan Vanderbruggen, Chunhua Liao, Murali Emani, and Bronis de Supinski. 2023a. Lm4hpc: Towards effective language model application in high-performance computing. In *International Workshop on OpenMP*, pages 18–33. Springer.

Le Chen, Quazi Ishtiaque Mahmud, and Ali Jannesari. 2022. Multi-view learning for parallelism discovery of sequential programs. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 295–303. IEEE.

Le Chen, Quazi Ishtiaque Mahmud, Hung Phan, Nesreen Ahmed, and Ali Jannesari. 2023b. Learning to parallelize with openmp by augmented heterogeneous ast representation. *Proceedings of Machine Learning and Systems*, 5.

Zhikai Chen, Haitao Mao, Hang Li, Wei Jin, Hongzhi Wen, Xiaochi Wei, Shuaiqiang Wang, Dawei Yin, Wenqi Fan, Hui Liu, et al. 2023c. Exploring the potential of large language models (llms) in learning on graphs. *arXiv preprint arXiv:2307.03393*.

Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *2009 international conference on parallel processing*, pages 124–131. IEEE.

Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. 2022. Complexity-based prompting for multi-step reasoning. In *The Eleventh International Conference on Learning Representations*.

Samujjwal Ghosh, Subhadeep Maji, and Maunendra Sankar Desarkar. 2022. Graph neural network enhanced language models for efficient multilingual text classification. *arXiv preprint arXiv:2203.02912*.

Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Autotuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*, pages 1–10. Ieee.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2023. Pre-training to learn in context. *arXiv preprint arXiv:2305.09137*.

Re'em Harel, Yuval Pinter, and Gal Oren. Learning to parallelize source code via openmp with transformers.

Zia Ul Huda, Rohit Atre, Ali Jannesari, and Felix Wolf. 2016. Automatic parallel pattern detection in the algorithm structure design space. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 43–52. IEEE.

Zia Ul Huda, Ali Jannesari, and Felix Wolf. 2015. Using template matching to infer parallel design patterns. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–21.

Srinivasan Iyer, Xi Victoria Lin, Ramakanth Pasunuru, Todor Mihaylov, Daniel Simig, Ping Yu, Kurt Shuster, Tianlu Wang, Qing Liu, Punit Singh Koura, et al. 2022. Opt-iml: Scaling language model instruction meta learning through the lens of generalization. *arXiv preprint arXiv:2212.12017*.

Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. 1999. The openmp implementation of nas parallel benchmarks and its performance.

Nicholas Johnson and Adrian Jackson. 2013. The epcc openacc benchmark suite. In *Exascale Applications and Software Conference*.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Xiaonan Li and Xipeng Qiu. 2023a. Finding support examples for in-context learning. *arXiv preprint arXiv:2302.13539*.

Xiaonan Li and Xipeng Qiu. 2023b. Mot: Pre-thinking and recalling enable chatgpt to self-improve with memory-of-thoughts. *arXiv preprint arXiv:2305.05181*.

Zhen Li, Rohit Atre, Zia Huda, Ali Jannesari, and Felix Wolf. 2016. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software*, 117:282–295.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Tim Mattson and Rudolf Eigenmann. 1999. Openmp: An api for writing portable smp application software. In *SuperComputing 99 Conference*.

Larry Meadows. 2007. Openmp 3.0–a preview of the upcoming standard. In *High Performance Computing and Communications: Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007. Proceedings 3*, pages 4–4. Springer.

Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2021. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943*.

Gordon E. Moore. 2006. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35.

Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024a. Can large language models write parallel code? *arXiv preprint arXiv:2401.12554*.

Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024b. Hpc-coder: Modeling parallel programs using large language models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–12. Prometeus GmbH.

Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, Todd Gamblin, and Abhinav Bhatele. 2024c. Performance-aligned llms for generating fast code. *arXiv preprint arXiv:2404.18864*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2023. Gpt family of apis. https://platform.openai.com/docs/guides/text-generation. Last accessed: 29th April, 2023.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Louis-Noël Pouchet and Tomofumi Yuki. 2017. Polybench: The polyhedral benchmark suite (version 4.2).

Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1. Citeseer.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Michael Schaarschmidt, Dominik Grewe, Dimitrios Vytiniotis, Adam Paszke, Georg Stefan Schmid, Tamara Norman, James Molloy, Jonathan Godwin, Norman Alexander Rink, Vinod Nair, et al. 2021. Automap: Towards ergonomic automated parallelism for ml models.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*, pages 593–607. Springer.

Yuanyuan Shen, Manman Peng, Shiling Wang, and Qiang Wu. 2021. Towards parallelism detection of sequential programs with graph neural network. *Future Generation Computer Systems*, 125:515–525.

Yuanyuan Shen, Manman Peng, Qiang Wu, and Guoqi Xie. 2023. Multigraph learning for parallelism discovery in sequential programs. *Concurrency and Computation: Practice and Experience*, 35(9):e7648.

Ali TehraniJamsaz, Quazi Ishtiaque Mahmud, Le Chen, Nasreen K Ahmed, and Ali Jannesari. 2023. Perfograph: A numerical aware program graph representation for performance optimization and program analysis. *37th Conference on Neural Information Processing Systems (NeurIPS 2023).*

Ali TehraniJamsaz, Mihail Popov, Akash Dutta, Emmanuelle Saillard, and Ali Jannesari. 2022. Learning intermediate representations using graph neural networks for numa and prefetchers optimization. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1206–1216. IEEE.

John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

Xinyi Wang, Wanrong Zhu, Michael Saxon, Mark Steyvers, and William Yang Wang. 2024. Large language models are latent variable models: Explaining and finding good demonstrations for in-context learning. *Advances in Neural Information Processing Systems*, 36.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Jerry Wei, Le Hou, Andrew Lampinen, Xiangning Chen, Da Huang, Yi Tay, Xinyun Chen, Yifeng Lu, Denny Zhou, Tengyu Ma, et al. 2023. Symbol tuning improves in-context learning in language models. *arXiv preprint arXiv:2305.08298*.

Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. 2022. BabelTower: Learning to auto-parallelized program translation. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 23685–23700. PMLR.

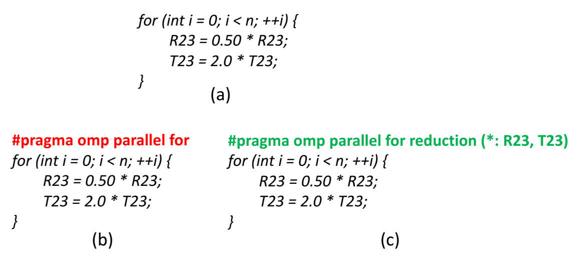
Canwen Xu, Yichong Xu, Shuohang Wang, Yang Liu, Chenguang Zhu, and Julian McAuley. 2023. Small models are valuable plug-ins for large language models. *arXiv preprint arXiv:2305.08848*.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. Codebertscore: Evaluating code generation with pretrained models of code.

A Appendix

A.1 Background Examples



```
for (int i = 0; i < n; ++i) {
    R23 = 0.50 * R23;
    T23 = 2.0 * T23;
}
```

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    R23 = 0.50 * R23;
    T23 = 2.0 * T23;
}
```

```
#pragma omp parallel for reduction (*: R23, T23)
for (int i = 0; i < n; ++i) {
    R23 = 0.50 * R23;
    T23 = 2.0 * T23;
}
```

Figure 7: Result of GPT-3.5 and GPT-4 (b) before and, (c) after applying AUTOPARLLM guidance on input code (a). This reduction loop is taken from IS application of NAS Parallel Benchmark

Listing 1: Reduction loop example

```
#pragma omp parallel for reduction(+:sum
    )
for (int i = 0; i < n; ++i) {
    sum += arr[i];
}
```

Listing 2: Do-all loop example

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    arr[i] = i + 1;
}
```

Listing 3: private loop example

```
#pragma omp parallel for private(temp)
for (int i = 0; i < n; i++) {
    temp = array[i] * 2;
    result[i] = temp;
}
```

Listing 4: Combination of reduction and private loop example

```
#pragma omp parallel for private(temp)
    reduction(+:sum)
for (int i = 0; i < n; i++) {
    temp = array[i] * 2;
    sum += temp;
}
```

A.2 Hardware Specifications

The runtime experiments are performed on compute cluster with Slurm workload manager. Each compute node is invoked using a job script like below.

Listing 5: Basic job script for running the benchmarks

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
# max memory allocated for the job
#SBATCH --mem=8G
# time allocated for the job
#SBATCH --time=0:10:00
# output file name
#SBATCH --output=output.out
# error file name
#SBATCH --error=error.err
# name of the job
#SBATCH --job-name="running-job"
# select CPU architecture
# (biocrunch = Intel Xeon Gold 6152,
    swift= AMD EPYC 7543P)
#SBATCH --partition=biocrunch/swift

# -------- instructions for running
    benchmarks here --------
```

Table 6 shows the detailed configuration of the CPU architectures that are used for the runtime experiments.

Table 6: Hardware Specifications

Features	Intel Xeon Gold 6152	AMD EPYC 7543P
Total Cores	22	32
Total Threads	44	64
Max Frequency	3.7 GHz	3.7 GHz
Processor Base Frequency	2.1 GHz	2.8 GHz
Cache	30.25 MB L3 Cache	256 MB L3 Cache
Memory Types	DDR4-2666	DDR4

A.3 Loss curves, training time and Hyparameters of GNN classifiers

Hyperparameters: We experimented with different hidden layer sizes and learning rates and ultimately chose 64 as the hidden layer size and set the learning rate to 0.01. Each node of the heterogeneous PERFOGRAPHis embedded to a 120-dimensional vector. Therefore, the input layer size is set to 120. The output layer size is set to the number of classes, which is 2, as all three of the GNN models do binary classification. For graph-level prediction, the 'mean' aggregation function combines the results of different node types, and finally linear classifier is used in the last layer of the RGCN model. The linear classifier produces a probability score for each class. The models are trained for 120 epochs, and the checkpoint with the highest validation accuracy is saved for later inference.

Table 7 shows the training time required for all three models. Figure 8 shows the epoch vs. loss curve for all three GNN models.

Table 7: The training time required for the three GNN-based predictors

Model	Training Time
Parallelism Detection model	17 min 5 sec
Private Detection model	9 min 27 sec
Reduction Detection model	9 min 12 sec

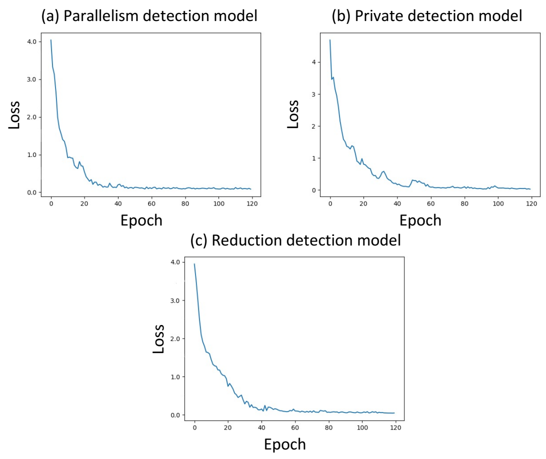


Figure 8: Epoch vs. loss curves for the three GNN-based parallelism and pragma detection modules

A.4 Ensuring Correctness of the generated codes

When the GNN predicts that a given loop is not parallel AUTOPARLLM does not generate any parallel code for that. In this case, AUTOPARLLM outputs the original sequential loop. Hence, correctness is preserved. Like any other ML model, there will be False Positives (FP) and False Negatives (FN). Since all loops in the Rodinia Benchmark are correctly classified by AUTOPARLLM, we will explain how FP and FN are handled for the NAS benchmark. During our experiments with NAS benchmark, 30 out of 32 non-parallel loops are correctly detected by AUTOPARLLM. These 30 loops that are detected as non-parallel do not need to be checked further. Because we know that even if there are some parallel loops in this set, treating them as sequential loops will only hurt performance but not correctness. There are 90 loops in the test set of NAS benchmark. That means using AUTOPARLLM, we can filter out 33.33% of loops for analysis safely. Hence, developers' workload is reduced by 33.33%.

Only the remaining loops need to be analyzed further after applying AUTOPARLLM, as they may contain the following scenarios:

- Non-parallel loop detected as parallel: Only 2 non-parallel loops are wrongly detected as parallel; we use the original sequential version of these two loops to maintain correctness during execution.

- Wrong OpenMP clause prediction: Only four loops have been decorated with the wrong OpenMP clauses. We also use the sequential version of these 4 loops to maintain correctness. Note that using proper OpenMP clause will result in speedup, but that will give an unfair advantage to AUTOPARLLM while calculating performance gain during execution. Hence, we use the sequential version as AUTOPARLLM failed to detect the right clauses.

Therefore, only 6 out of the 60 analyzed loops required manual fixing. The rest 54 loops are already correctly generated by AUTOPARLLM. The outputs of the generated codes are compared with the original outputs to ensure that the generated codes produce results identical to the original codes.

A.5 Structure of PERFOGRAPH & Node and Edge Embedding Generation

Each loop is first converted to IR, and then from that IR, we create the PERFOGRAPH representation of that loop. There are three types of nodes:

- Control nodes: Each control node represents a statement in IR. Tokens in the IR statement are considered as features for the control node. We embed each token of the statement and finally concatenate the embedding of all tokens to generate the final embedding for a control node.

- Variable nodes: PERFOGRAPH contains only the type of a variable in variable nodes. So, the type is considered as the feature for variable nodes, and embedding is generated for the type token.

- Constant nodes: For constant nodes, PERFOGRAPH representation contains both type and value, so we consider both of them as features. First, we generate the embedding for the type token. For generating the embedding for value-token, we use the Digit Embedding as described in PERFOGRAPH paper. Finally, the type-token and value-token embeddings are concatenated to generate the final embedding for each constant nodes.

The variable and constant nodes represent the variables and constants that are associated with those IR statements (control nodes). There are three types of edges:

- Control flow edges: Represents the flow of the program.

- Data flow edges: Represents the data dependencies of different nodes in the program.

- Call flow edges: Represents the functional call dependencies of the program.

Nodes are connected with each other using these three different edges. Edge embeddings are obtained using a one-hot-encoding approach. All the embeddings mentioned are generated using the default Pytorch learnable embedding mechanism.

A.6 ParaBLEU implementation

We modified ParaBLEU score following the same idea of (Wen et al., 2022). It is adjusted for OpenMP directives. The modified formula of the ParaBLEU score, which is suitable for OpenMP code evaluation, is shown below:

$$ParaBLEU_{OMP} = \alpha * BLEU + \beta * BLEU_{OpenMPkeywords}$$

In this formula, α and β are the heuristic parameters that we set both of them as 0.5 for our evaluation. These ratios specify the contribution of the original BLEU score and the weighted BLEU score, highlighting the similarities between n-grams containing our selected OpenMP keywords.

A.7 Runtime Experiments - NAS Parallel Benchmark Input

We execute the applications of NAS Parallel Benchmark suite using CLASS A input that comes along with the benchmark. Table 8 shows the size of the problems and the number of iterations for each application for CLASS A. Detailed runtimes for each application can be found in the repository link.

Table 8: Input problem size for each application in NAS Parallel Benchmark Suite

Applications	Problem Size	Number of iterations
IS	2^{23}	10
EP	2^{28}	1
FT	$256 \times 256 \times 126$	6
CG	14000	15
MG	$256 \times 256 \times 256$	4
LU	$64 \times 64 \times 64$	250
BT	$64 \times 64 \times 64$	200
SP	$64 \times 64 \times 64$	400

A.8 Runtime Experiments - Rodinia-3.1 Benchmark Input

Rodinia-3.1 Benchmark applications are also executed using the inputs that come along with the benchmark itself. Below we provide details regarding the inputs for running the applications.

- BFS is executed with the input ‘graph1MW_6.txt’, which applies BFS on a graph that contains 1000000 nodes and 5999970 edges.
- B+Tree is executed with the inputs ‘mil.txt’ and ‘command.txt’ with the block size (6000, 3000) and key size of 10000.
- Heartwall is executed with the input ‘test.avi’ which contains a series of medical ultrasound images with a framesize of 10.
- 3D is executed with a 3D grid size (512, 8, 100) along with inputs power_512x8 and temp_512x8, where the last two inputs represent power dissipation and initial temperature at each grid-point for thermal simulations.

We provide detailed runtimes for each application in Rodinia in the repository link.

A.9 Runtime Experiments - Compare Average Speedup Across DIfferent LLMs

Figure 9 shows the effect of AUTOPARLLM on four LLMs CodeGen, CodeLlama, GPT-3.5 and GPT-4. AUTOPARLLM improves the average speedup gain of all four LLMs on both NAS and Rodinia Benchmark.

A.10 Runtime Experiments - Different Thread Configurations

We analyzed the effect of AUTOPARLLMon GPT-4 with varying thread numbers. GPT-4 is chosen as, according to our study, it performed the best among the other LLMs. We experimented with 4 thread configurations: 2, 4, 8 and 16. From the results of Figure 10, 11 and 12, it can be observed that in all configurations, the speedup up obtained by ALLM-GPT-4 is better than the base GPT-4.

A.11 Traditional Approaches Comparison

In Table 9, we compare the parallelism discovery of AUTOPARLLM against three popular parallelization tools: Pluto, AutoPar, and DiscoPoP on the DiscoPoP subset (1226 files) of OMP_Serial Dataset. The task is to detect whether a code can be parallelized or not. There are also two other subsets: Pluto Subset and AutoPar Subset in the OMP_Serial dataset. However, DiscoPoP can not process some of the codes in those two subsets as it requires each code to be executable for proper analysis. So, to have a fair comparison with all three tools, we choose the DiscoPoP subset for this experiment as all codes in the DiscoPoP subset

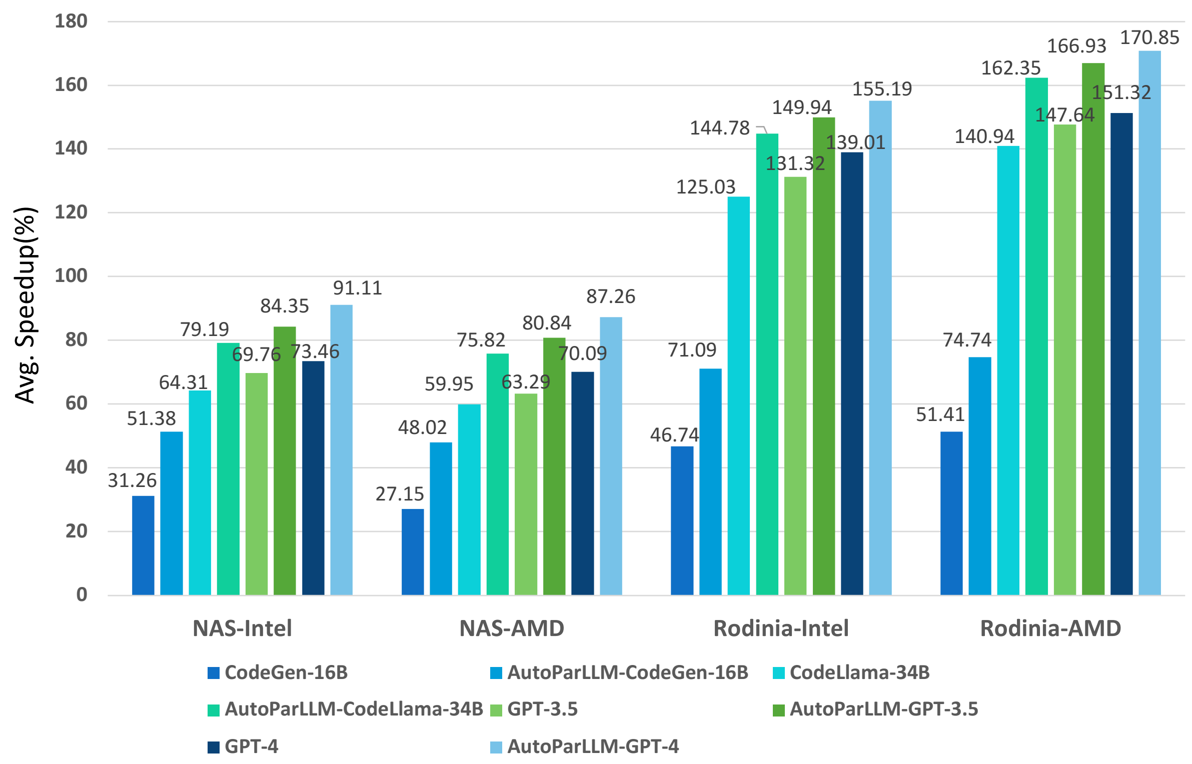


Figure 9: Effect of AutoParLLM on different LLMs. LLMs are prompted with few shot settings & speedups are reported using 4 threads.

are executable. Pluto and AutoPar perform static analysis to find parallel regions (such as loops), whereas DiscoPoP is a dynamic analysis tool that executes the code to identify potential parallel regions. Therefore, choosing these three tools enables us to compare AUTOPARLLM against both static and dynamic analysis tools. Due to program execution, from Table 9, we see that DiscoPoP gains an advantage over Pluto and AutoPar as it has 15% and 8% better accuracy, respectively. However, it can be observed that AUTOPARLLM has far better accuracy than all three tools.

Table 9: Parallelism Discovery on DiscoPoP Subset (Detecting parallel loops)

Tool	Precision	Recall	F1-score	Accuracy
Pluto	**1**	0.38	0.55	0.48
AutoPar	**1**	0.49	0.67	0.55
DiscoPoP	**1**	0.54	0.70	0.63
AUTOPARLLM	0.99	**0.99**	**0.99**	**0.99**

A.12 Comparing with Deep Learning Approaches

Here, we applied AUTOPARLLM on the Pluto, AutoPar, and DiscoPoP subsets of the OMP_Serial Dataset, which consists of parallel and non-parallel codes. The task is the same as the previous. The Pluto subset has 4032 files, the AutoPar subset has 3356 files, and the DiscoPoP subset has 1226 files. The results of the tools Graph2Par (Chen et al., 2023b), PrograML and PERFOGRAPH are reported from (TehraniJamsaz et al., 2023). We apply our AUTOPARLLM model on the same subsets and compare these approaches. Table 10 shows that AUTOPARLLM surpasses the performance of the state-of-the-art PERFOGRAPH (TehraniJamsaz et al., 2023) by achieving as high as 6% better accuracy.

Table 10: Performance comparison of AUTOPARLLM on parallelism discovery task in the OMP_Serial dataset with existing approaches.

Subset	Approach	Precision	Recall	F1-score	Accuracy
Pluto	Graph2par	0.88	0.93	0.91	0.86
	PROGRAML	0.88	0.88	0.87	0.89
	PERFOGRAPH	0.91	0.90	0.89	0.91
	AUTOPARLLM	**0.97**	**0.98**	**0.98**	**0.96**
AutoPar	Graph2par	0.90	0.79	0.84	0.80
	PROGRAML	0.92	0.69	0.67	0.84
	PERFOGRAPH	0.85	0.91	0.85	0.86
	AUTOPARLLM	**0.93**	**0.92**	**0.93**	**0.92**
DiscoPoP	Graph2par	0.90	0.79	0.84	0.81
	PROGRAML	0.92	0.94	0.92	0.91
	PERFOGRAPH	0.99	**1**	0.99	0.99
	AUTOPARLLM	**0.99**	0.99	**0.99**	**0.99**

A.13 Analyzing Wrong Predictions in OpenACC

Here we describe the wrong predictions in our OpenACC experiment. The test set contains 12 parallel (5 `private`, 7 `reduction`) and 5 `non-parallel` loops. For generating the complete OpenACC clauses we use the predicted patterns from AUTOPARLLM and incorporate them into the prompts

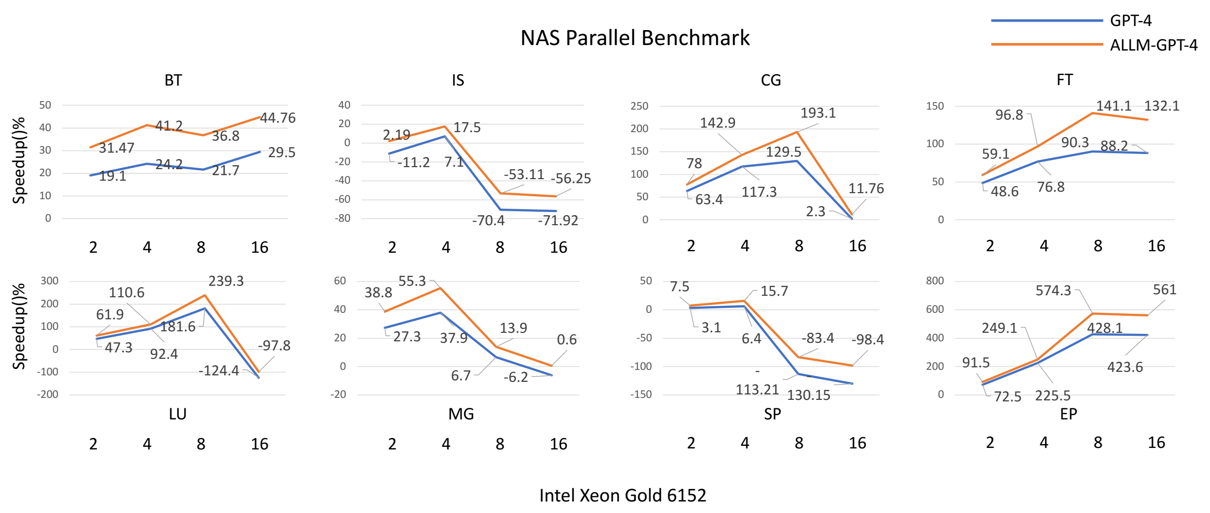


Figure 10: Comparing Effects of AutoParLLM on GPT-4 for different thread configurations for NAS Benchmark on Intel Xeon Gold Machine.

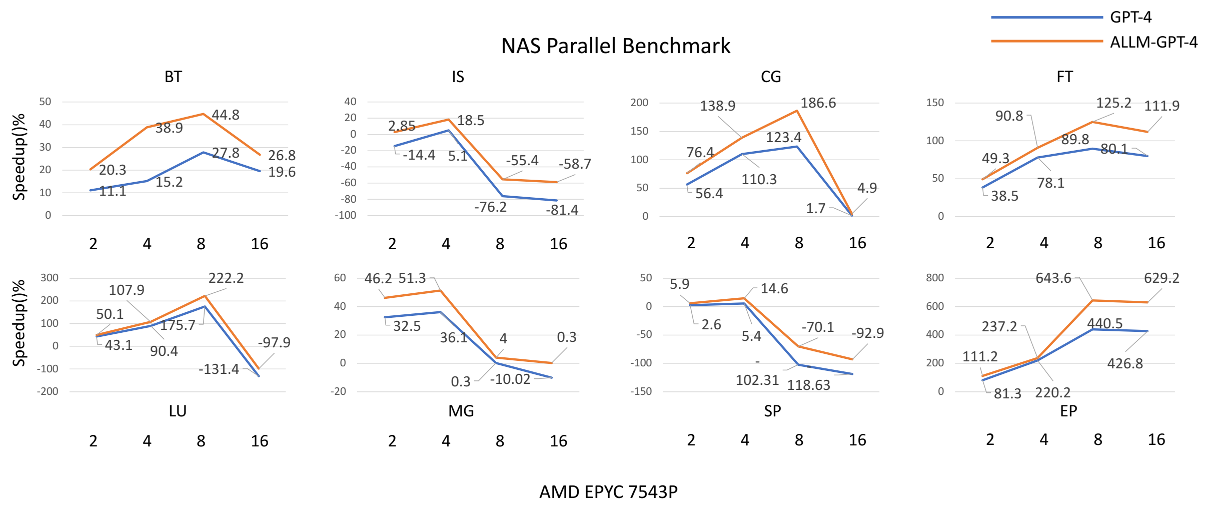


Figure 11: Comparing Effects of AutoParLLM on GPT-4 for different thread configurations for NAS Benchmark on AMD EPYC Machine.

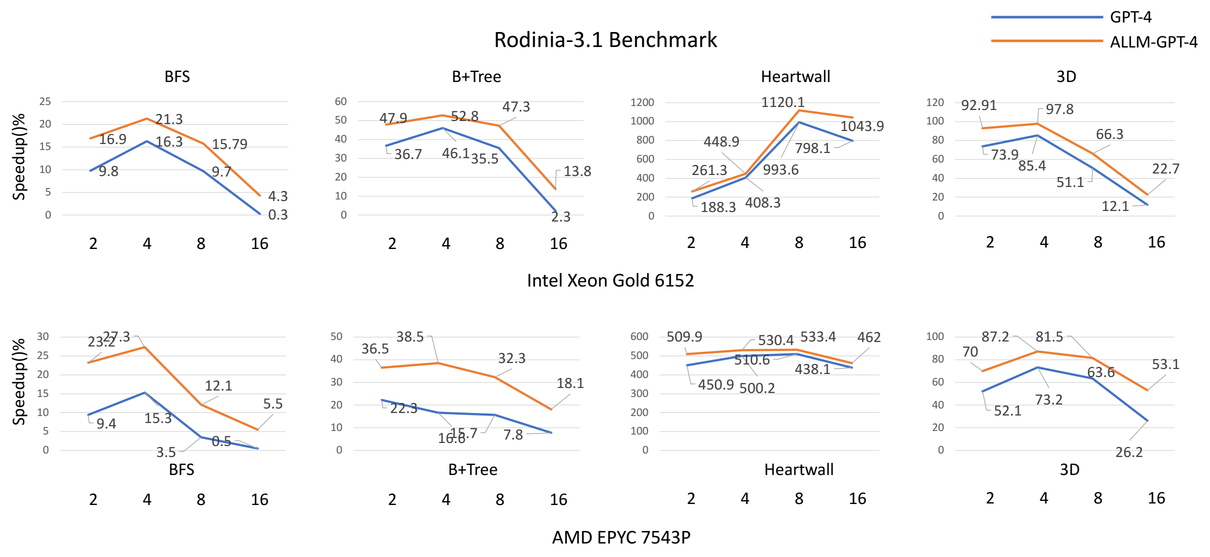


Figure 12: Comparing Effects of AutoParLLM on GPT-4 for different thread configurations for Rodinia-3.1 Benchmark on both Intel Xeon Gold and AMD EPYC Machine

#	Loop Body	AutoParLLM-GPT-4 suggestion	Developer's version
1	for (i = 0; i < _PB_N; i++) { for (j = 0; j < _PB_N; j++) { x2[i] = x2[i] + A[j][i] * y_2[j]; } }	***#pragma acc loop collapse(2) reduction(+:x2[:_PB_N])***	***#pragma acc loop***
2	for (i = 0; i < n; i++){ tmp += data[i*n+j]; }	***#pragma acc loop***	***#pragma acc loop reduction(+:tmp)***

Figure 13: Mismatches of AUTOPARLLM in the OpenACC dataset.

for the GPT-4. However, instead of OpenMP now we instruct the AUTOPARLLM-GPT-4 to parallelize the code using OpenACC. There were 2 wrong predictions by AUTOPARLLM-GPT-4. For the first case, in Figure 13, the developer's implementation from the benchmark only parallelizes the outermost loop. However, AUTOPARLLM-GPT-4 configuration suggested two extra clauses: `collapse` and `reduction`. `collapse` merges the two loops in a flattened 2D iteration space, allowing OpenACC to parallelize both loops. `reduction` ensures proper summation among threads and prevents any race conditions. Although this configuration does not match with the developer's version from benchmark it is more efficient as it parallelizes both of the loops. This shows the potential of AUTOPARLLM in finding optimization opportunities that may be overlooked by human developers. For the second case, AUTOPARLLM correctly detects it as a parallel loop however, it fails to identify the `reduction` operation. We hypothesize the reason is because of the presence of complex indexing on variable `data`.

A.14 Detailed Related Works

Here we describe some of the existing approaches to automatic parallelization and also the current metrics for translation evaluation.

A.14.1 Traditional Parallelism Assistant Tools

Traditional tools are based on static and dynamic analysis to automatically parallelize sequentially written programs. PLuTo (Bondhugula et al., 2008) analyzes code statically and can optimize programs for parallel execution. It is a polyhedral source code optimizer based on OpenMP (Meadows, 2007). Rose (Quinlan and Liao, 2011) is another static source-to-source compiler infrastructure that also supports automatic parallelization using OpenMP. Both of these tools do not require or use code runtime information while generating parallel counterparts of sequential programs. DiscoPoP (Li et al., 2016) is a dynamic analysis-based parallelism assistant tool. It uses dynamic control-flow analysis and data-dependence profiling for identifying parallel regions in source programs. The works of (Huda et al., 2015) use the output of DiscoPoP to find parallel patterns using template-matching techniques. The DiscoPoP-generated hybrid dependence analysis results have been used by (Huda et al., 2016) to detect various parallel patterns like Geometric decomposition, reduction and task-based parallelism. However, these traditional analysis-based tools have some drawbacks, as pointed out by (Chen et al., 2023b), and they miss a lot of parallelism opportunities due to being overly conservative.

A.14.2 Data-driven Approaches

With the significant progress in the field of machine learning and deep learning, many have proposed automatic data-driven approaches to identify parallelism opportunities and suggest appropriate constructs. Authors of (Chen et al., 2023b) proposed an approach based on graph neural networks and augmented Abstract Syntax Trees (ASTs) to identify parallel loops. Their results show that their GNN-based approach outperforms PragFormer (Harel et al.), which uses Transformers (Vaswani et al., 2017) to discover parallelism opportunities in code. (Shen et al., 2021) uses contextual flow graphs with graph convolution neural networks (Kipf and Welling, 2016) to detect parallelism. (Shen et al., 2023) uses a combination of control flow, data flow, and abstract syntax tree to predict parallelism. There are also attempts to use IR-based representation for automatic parallelization of ML models (Schaarschmidt et al., 2021). Even though there has been some progress in adapting machine learning techniques to predict parallelism opportunities, little effort has been applied to connect parallelism detection and code generation. In this work, we address this gap. We connect GNNs and LLMs to not only discover parallelism but also generate parallel code. Connecting GNNs to LLMs has been investigated recently (Ghosh et al., 2022; Chen et al., 2023c). However, to the best of our knowledge, we are the first to leverage the result of GNN to guide LLM to generate parallel code out of serial code.

A.14.3 LLM-based Approaches

Large Language Models (LLMs) have achieved remarkable success across various domains, including parallel code generation. Chen et al. (2023a) were pioneers in applying LLMs to high-performance computing (HPC) tasks, including parallelism detection. They compared the performance of GPT-3.5 with their previous GNN-based approaches (Chen et al., 2022, 2023b). Their findings demonstrated that GPT-3.5 could achieve competitive performance in parallelism detection with even basic prompts. In subsequent research, they trained an LLM specifically for OpenMP pragma generation, introducing a tailored chain-of-thought approach (Chen et al., 2024). Nichols et al. (2024a) evaluated the performance of the parallel code generated by LLMs and indicated that even with great potential, noting that while LLMs show great potential, there remains a significant gap between the generated code and the expectations of the HPC community. They also explored fine-tuning LLMs for generating optimized parallel code (Nichols et al., 2024b,c). In contrast to these prior works, our approach uniquely combines the strengths of GNNs with LLMs to generate prompts enriched with external knowledge. This hybrid method leverages the advantages of both GNNs and LLMs, providing a versatile solution compatible with any LLM.

A.14.4 Metrics for Translation Evaluation

The BLEU score (Papineni et al., 2002) is a classical metric for evaluating textual similarity in machine translation. It assesses the overlap between sequences of consecutive n words, called n-grams. Meteor (Banerjee and Lavie, 2005) was introduced to address some of the limitations of the BLEU score, such as its tendency to underestimate high-order n-grams. ROUGE (Lin, 2004) encompasses a set of metrics that evaluate various aspects of textual similarity. To evaluate code generation, recent works have introduced metrics like CodeBLEU (Ren et al., 2020) and CodeBERTScore (Zhou et al., 2023). Authors of BabelTower (Wen et al., 2022) designed a metric called "ParaBLEU" specifically for evaluating parallel codes. However, the designed metric is limited to evaluating CUDA code only. So, none of these metrics have been specifically designed for evaluating the quality of OpenMP constructs in terms of textual similarity.

A.15 Experimenting with Randomized Few Shot COT

Here, we discuss the results of our experiments regarding adding randomly sampled examples in GPT-4 prompts. Instead of handpicking samples from different parallelization scenarios, here, the samples for constructing the prompt are randomly picked from the dataset. To enable consistent comparison with the hand-picked sampling technique we also limit the sample number to 5 samples. The results are presented for both NPB and Rodinia benchmark in Table 11 and Table 12, respectively. From the results, it can be observed that the performance and of hand-picked sample based prompting (Man) and randomized sampling based (Rand) techniques yields very close results in most cases. However, AUTOPARLLM surpasses both prompting techniques by a significant margin in both benchmarks in terms of all three metrics scores.

Table 11: Results on NPB Benchmark Suite for Randomized Few Shot Prompting

Model	CBTScore	ParaBLEU	OMPScore
0-shot-COT-GPT-4	73.8	27.49	46.4
few-shot-COT-GPT-4 (Rand)	76.3	35.69	58.64
few-shot-COT-GPT-4 (Man)	76.5	29.97	58.06
AUTOPARLLM-GPT-4	**96.4**	**48.48**	**95.15**

Table 12: Results on Rodinia-3.1 Benchmark Suite for Randomized Few Shot Prompting

Model	CBTScore	ParaBLEU	OMPScore
0-shot-COT-GPT-4	91.2	49.14	79.37
few-shot-COT-GPT-4 (Rand)	93.8	55.67	89.43
few-shot-COT-GPT-4 (Man)	92.12	54.43	89.01
AUTOPARLLM-GPT-4	**98.6**	**69.09**	**98.1**

A.16 Few Shot COT Prompt Example

```
Qs: Parallelize the following code using OpenMP
for (int i = 0; i < n; ++i) {
arr[i] = arr[i-1] + arr[i+1];
}
Ans.
1. The loop contains inter-iteration dependencies at i-1 and i+1.
2. Hence loop can not be parallelized.

Qs: Parallelize the following code using OpenMP
for (int i = 0; i < n; ++i) {
sum += arr[i];
}
Ans.
1. The following code combines multiple iterations into a final outcome 'sum'
2. Hence adding reduction clause is necessary to parallelize
3. reduction should be added on 'sum' with '+' operator.
Qs: Parallelize the following code using OpenMP
for (int i = 0; i < n; ++i) {
arr[i] = i + 1;
}
Ans:
1. The code contains do-all pattern as all iterations are independent as no need
to make any variable private to each thread.
2. Simply adding the 'parallel for' clause should be sufficient.
Qs:
for (int i = 0; i < n; i++) {
        temp = array[i] * 2;
        result[i] = temp;
    }
Ans:
1. The code contains do-all pattern as all iterations are independent
2. But variable 'temp' needs to be private to each thread.
3. private clause should be added on 'temp'.
Qs:
for (int i = 0; i < n; i++) {
        temp = array[i] * 2;
        sum += temp;
    }
Ans:
1. The code contains do-all pattern as all iterations are independent
2. But variable 'temp' needs to be private to each thread.
3. private clause should be added on 'temp'.
4. The following code combines multiple iterations into a final outcome 'sum'
5. Hence adding reduction clause is necessary to parallelize
6. reduction should be added on 'sum' with '+' operator.

Now Parallelize the following code using OpenMP:
<<input sequential code here>>
```

A.17 Zero Shot COT Prompt Example

```
Parallelize code using OpenMP by following the below rules:
1. If loop iterations are independent of each other and no variable
is required to be private to each thread, then simply add 'parallel for'
clause
2. If loop iterations are independent but there is variable that needs
to be private to each thread then apply privatization through
'private' clause
3. If a variable combines results from multiple loop iterations and
finally computes the output then apply reduction on the variable
along with the associative operation.
4. It is possible that combinations of the above cases may arise.
<<input sequential code>>
```