CS 406 - Term Project Final Report

Tan Çetiner 27024
Emre Duman 26347

May 2023

1 Introduction

The detection and diagnosis of performance issues in parallel codes pose significant challenges, particularly in the context of large-scale applications. This project aims to overcome these challenges by leveraging the capabilities of large language models, specifically ChatGPT. Our objective is to develop an automated pipeline capable of identifying and categorizing performance problems in parallel codes, thereby enhancing the efficiency and optimization of applications.

In pursuit of this goal, we drew inspiration from the research article titled "An Empirical Study of High Performance Computing (HPC) Performance Bugs"[1]. This paper presents an extensive investigation into various problem types and corresponding solutions in open source high-performance computing projects. Building upon the insights gained from this study, we constructed a pipeline that effectively utilizes the data provided, employing ChatGPT to reproduce the reported outcomes. Consequently, we assess the performance of ChatGPT in terms of its ability to interpret the semantics of parallel computing code. Finally, we present the obtained results, offering our insights and perspectives on the potential use cases and broader applicability of large language models in this domain.

The complete source code for our project is available in the accompanying repository, and the relevant scripts will be referenced throughout this report.

2 Methodology

This chapter provides a comprehensive description of the pipeline, outlining its constituent components and their respective functionalities. The pipeline can be delineated into the following segments:

- Analyzing Spreadsheet
- Fetching the code

[1] Shull, F., Carver, J., Hochstein, L.; Basili, V. (2005). Empirical study design in the area of high-performance computing (HPC). 2005 International Symposium on Empirical Software Engineering, 2005. https://doi.org/10.1109/isese.2005.1541839

- Processing code files
- Interaction with ChatGPT
- Evaluating Results

These components collectively form the pipeline, facilitating the automated detection and classification of performance problems in parallel codes. The subsequent sections will provide detailed insights and explanations of each stage, elucidating the underlying methodologies and techniques employed.

2.1 Analyzing Spreadsheet

The initial step entails reading the spreadsheet and collecting the requisite data. Following thorough exploration, we identified three columns for utilization:

- **Sub Category**: The problem category consists of five distinct types of columns, namely "category," "sub-category," "sub-sub-category," "sub-sub-sub-category," and "sub-sub-sub-sub-category." These columns progressively offer more detailed categorical information pertaining to the problem encountered within the code. Our aim was to strike a balance between preserving data integrity by considering null values and obtaining sufficient granularity in problem categorization. Consequently, we determined that the "Sub Category" column serves as an optimal choice.
- **Solution Main Category**: The solution category encompasses two column types: "Solution Main Category" and "Solution Category." Building upon the rationale outlined in the previous point, we opted for the "Solution Main Category" column.
- **Link to commit**: This column facilitates access to the corresponding commit on GitHub, where the performance bug solution was implemented. This link is crucial for retrieving the associated code.

Upon acquiring this data, we organized it into JSON elements and assigned index numbers to facilitate convenient access patterns. These JSON elements were then assembled into a list within a JSON file, structured as follows:


```
"commits": [
  {
    "Sub-Category": "Memory/Data locality",
    "Solution Main Category": "Data structure optimization for data locality",
    "Link to commit": "https://github.com/CGAL/cgal/commit/28a9cb150ae9b11f9bb37d972be990d87b05cbcf",
    "Code Folder Index": 0
  },
```


Figure 1: An example commit information from commits.json file

The source code of this task is in the createCommits.py.

2.2 Fetching the code

To obtain the relevant source code linked to the commits mentioned in the Excel sheet, we employed the GitHub API. This allowed us to retrieve the code associated with the specified files. To ensure that the retrieved code accurately represents the state of the application when the performance problem occurred, we stored the version of the code just prior to the commit. By capturing the code at this specific point in time, we aimed to maintain fidelity to the context in which the performance issue manifested itself.

During the code file storage process, we organized each commit's files into individual folders. To maintain a systematic structure and facilitate easy identification and retrieval, we named each folder with the corresponding index assigned to the commit, as previously mentioned in the relevant subsection. This approach ensured a clear and organized arrangement of the files, enabling efficient referencing and subsequent analysis.

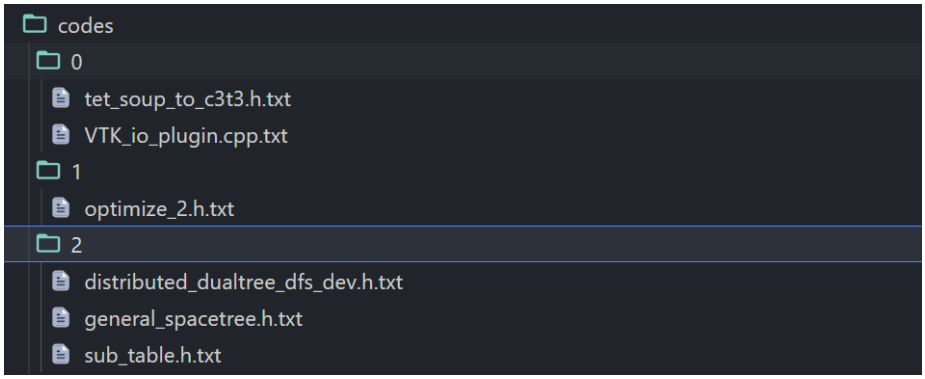

Figure 2: Codes directory structure. The name of the folder is the index of the commit information in the commits.json file.

The source code of this task is in the codeFetching.py.

2.3 Processing code files

A key challenge we encountered revolved around the maximum character limitation imposed when using the GptApi. Although we initially employed the API successfully with simpler code examples, the source codes we aimed to analyze within this project proved to be larger and more intricate in nature. Despite utilizing a Plus membership for GPT, which involved leveraging the gpt-3.5-turbo model, we encountered a constraint in the form of a maximum token limit of 4096 for a given context. This limitation posed a significant challenge, considering that some of the files we intended to analyze exceeded this token limit by a considerable margin.

Initially, our approach involved splitting the source code into smaller chunks to feed them sequentially to GPT. However, upon realizing that the token limitation pertained to the context rather than the individual message size, we recognized the need to explore alternative approaches. To mitigate the issue, we implemented a simple preprocessing step aimed at reducing the number of tokens. This involved removing excessive whitespace and comments from the code. Although this preprocessing technique had a modest impact on token reduction, it provided some relief in managing the token constraint.

It is important to acknowledge that the token constraint imposed limitations on certain files, preventing their inclusion in the classification process. To address this constraint, a potential solution could involve leveraging the capabilities of GPT-4, as it allows for double the number of tokens per context compared to previous versions (8192 tokens per context according to here). By utilizing GPT-4, we can potentially accommodate larger and more complex files, thereby enabling a more comprehensive analysis and classification of the source code.

The source code of this task is in the file in the codePreprocessing.py.

2.4 Interaction with ChatGPT

During the process of fetching the codes for our project, we encountered the challenge of identifying the specific files that contained the code changes responsible for addressing performance-related issues. Rather than manually selecting the relevant files, we sought to automate this process within our pipeline. To achieve this, we devised a solution that involved counting the occurrences of file extensions and categorizing them as potential performance commits or not.

To facilitate this classification, we created a JSON file that served as a mapping between file extensions and their likelihood of being associated with performance-related code changes. For instance, file extensions such as "hpp" were deemed more likely to contain performance commits, while extensions such as "txt" were considered less relevant in this context. When interacting with ChatGPT, we employed this JSON file to filter out files with non-relevant extensions, ensuring that only files with the potential to address performance concerns were fed into the system.

File Extension	Count	Is Development Code?
h	132	True
c	78	True
cpp	76	True
hpp	43	True
cu	26	True
cuh	15	True
cl	12	True
cc	8	True
clh	8	True
C	5	True
cmake	4	False
py	3	True
F	3	True
Z14	2	True
HASWELL	2	True
H	2	True
txt	2	False
rst	2	False
md	2	False
ml	2	True
bib	1	False
makefile	1	False
Makefile	1	False
cmakein	1	False
codelets	1	False
sh	1	False
in	1	False
Total	434	

Table 1: The file extensions, their counts and their mappings.

The source code of this task is in the countExtensions.py.

Following the implementation of our Python class designed to utilize the GptApi and prompt the GPT-3.5-turbo model, we proceeded to interact with ChatGPT to explore the impact of different preprocessing approaches on the token constraints. For this purpose, we prepared three distinct types of code data:

- Codes as initially fetched: This version included the code snippets in their original form, without any modifications.
- Codes with reduced whitespaces: In this version, we applied a preprocessing step to reduce the number of whitespaces in the code snippets, thereby optimizing the token utilization.

- Codes with reduced spaces and removed comments: Here, we further enhanced the preprocessing by not only reducing spaces but also removing any comments present within the code snippets. This approach aimed to test the effect of eliminating extraneous information on the model's performance.

The decision to divide the full preprocessed code into two categories—reduced whitespace and reduced spaces with removed comments—was driven by the desire to investigate the impact of comments on the model's understanding of the code. Comments often provide additional context and insights into the logic and motivation behind the code, and we were curious to determine whether leveraging this contextual information would improve the model's ability to comprehend the code snippets.

Furthermore, we explored the utilization of system roles within the conversation to provide a specific context for the model's responses. We crafted two system prompts, each emphasizing that the GPT model should answer as a parallel computing expert and that we would be providing code snippets with the expectation of receiving a problem classification from a given list. The only difference between the two system prompts was that one explicitly mentioned that if the model determines there is no inefficiency in the code, it can simply return "None."

Combining the three versions of code data with the two system prompts resulted in six distinct sets of results, which we could compare against one another. This comprehensive approach allowed us to assess the impact of different preprocessing techniques and system prompts on the model's performance, enabling us to make informed observations and draw meaningful conclusions.

Code Type	**System Prompt**	**Result**
Codes as initially fetched:	With None	Result 1
Codes with reduced whitespaces		Result 1
Codes with reduced spaces and removed comments		Result 1
Codes as initially fetched:	Without None	Result 1
Codes with reduced whitespaces		Result 1
Codes with reduced spaces and removed comments		Result 1

Table 2: Total of 6 combination of experimentation.

In order to utilize the model effectively, we proceeded to feed the code snippets into the GPT-3.5-turbo model. The process involved the following steps for each file:

- Clearing the conversation: To ensure a fresh start, we cleared any previous conversation or context that might have existed within the model.
- Defining the system role: We set the system role to establish the context for the model's responses. The system role emphasized that the model should assume the persona of a parallel computing expert and informed

it that we would be providing code snippets and expecting a problem classification from a predefined list.

- Entering the prompt: The code snippet was then presented as the user's prompt, serving as the input for the model.
- Obtaining the result: The GPT-3.5-turbo model generated a response based on the provided prompt, offering insights and problem classifications in accordance with the specified context.
- Storing the conversation: The entire conversation, comprising both user inputs and model responses, was stored in both JSON and text (txt) file formats. This comprehensive record of the conversation facilitated future analysis and evaluation.

This entire process was iterated for all six combinations, encompassing the three versions of code data and the two system prompts. Consequently, six distinct result files were generated, each containing the stored conversations and outcomes for a specific combination. These files serve as valuable resources for assessing the performance and behavior of the model across different preprocessing techniques and system prompts.

The source code of this task is in the classifyProblems.py.

2.5 Evaluating Results

To evaluate the performance of the model and analyze the results, an evaluation script was executed for each of the result folders. Within the script, several steps were taken:

- Extract the true labels and predicted labels
- Convert string labels to binary labels
- Calculate Categorical Accuracy, Total Accuracy and Multilabel Confusion Matrix

The results were plotted as a bar graph, and is presented in the Results section. Here we delineate the above mentioned steps.

- Accessing commit information: The script utilized the index of the commit to retrieve the corresponding commit's information from the commits.json data. This data, gathered from the spreadsheet of the paper, contained the correct problem category associated with each commit.
- Comparing categories: For each file in the result folder, the script parsed the GPT's response and extracted the problem category information. This category was then compared with the correct category obtained from the commits.json data.

- Determining labels: Based on the comparison between the predicted category and the correct category, the folder (commit) was labeled as one of the following:
  - Correct: If the predicted category matched the correct category.
  - Wrong: If the predicted category did not match the correct category.
  - None: If the GPT response indicated that there was no inefficiency in the code and the correct category was "None".
  - Wrong format: If the GPT response was not in the expected format or could not be parsed.
- Logging and summary: The evaluation script performed basic logging, summarizing the statistics of the results folder. This log provided an overview of the accuracy of the model's predictions and formed the basis for further analysis of the results.

By executing this evaluation script for each sub-folder within the result folders, we were able to assess the model's performance in classifying problem categories. The labeled folders, along with the logging and summary, contributed to our comprehensive analysis of the results, allowing us to gain insights and draw conclusions regarding the effectiveness of the model in addressing performance-related issues.

The source code of this task is in the evaluateResults.py.

Categorical Accuracy is a performance metric commonly used in classification tasks. It measures the percentage of correctly predicted labels out of all the samples.

Let's consider a classification problem with N samples, where each sample belongs to one of K categories. The true labels of the samples are represented by the vector $\mathbf{y}$, and the predicted labels by the vector $\hat{\mathbf{y}}$.

The categorical accuracy can be calculated as follows:

$$\text{Categorical Accuracy} = \sum_{k=1}^{K} \frac{1}{N_k} \sum_{i=1}^{N} \delta(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

where $\delta(\mathbf{y}_i, \hat{\mathbf{y}}_i)$ is the Kronecker delta function defined as:

$$\delta(\mathbf{y}_i, \hat{\mathbf{y}}_i) = \begin{cases} 1, & \text{if } \mathbf{y}_i = \hat{\mathbf{y}}_i \\ 0, & \text{otherwise} \end{cases}$$

The Kronecker delta function returns 1 when the true label $\mathbf{y}_i$ matches the predicted label $\hat{\mathbf{y}}_i$, and 0 otherwise. By summing up the results for all samples and dividing by the total number of samples N, we obtain the categorical accuracy.

The multilabel confusion matrix is a useful tool to evaluate the performance of a multilabel classification model. It provides insights into the classification results for multiple labels.

Let's consider a multilabel classification problem with N samples and L labels. The true labels of the samples are represented by the binary tensor $\mathbf{Y} \in \{0,1\}^{N \times L}$, and the predicted labels by the binary tensor $\hat{\mathbf{Y}} \in \{0,1\}^{N \times L}$.

The multilabel confusion matrix, denoted as $\mathbf{M} \in R^{L \times L}$, is defined as:

$$\mathbf{M}_{ij} = \sum_{n=1}^{N} \mathbf{Y}_{ni} \hat{\mathbf{Y}}_{nj}$$

where $\mathbf{M}_{ij}$ represents the count of samples where the true label for label i is 1 and the predicted label for label j is also 1. In other words, it measures the number of samples that are correctly classified for label i and predicted as label j.

The elements of the diagonal $\mathbf{M}_{ii}$ represent the true positive (TP) counts for each label, indicating the number of correctly classified positive instances. The off-diagonal elements $\mathbf{M}_{ij}$, where $i \neq j$, represent the false positive (FP) counts, indicating the number of instances wrongly predicted as label j instead of label i.

Using the multilabel confusion matrix, various evaluation metrics can be computed, such as precision, recall, and F1-score, to assess the performance of the multilabel classification model.

The source code of this task is in the resultAnalysis.ipynb.

3 Results

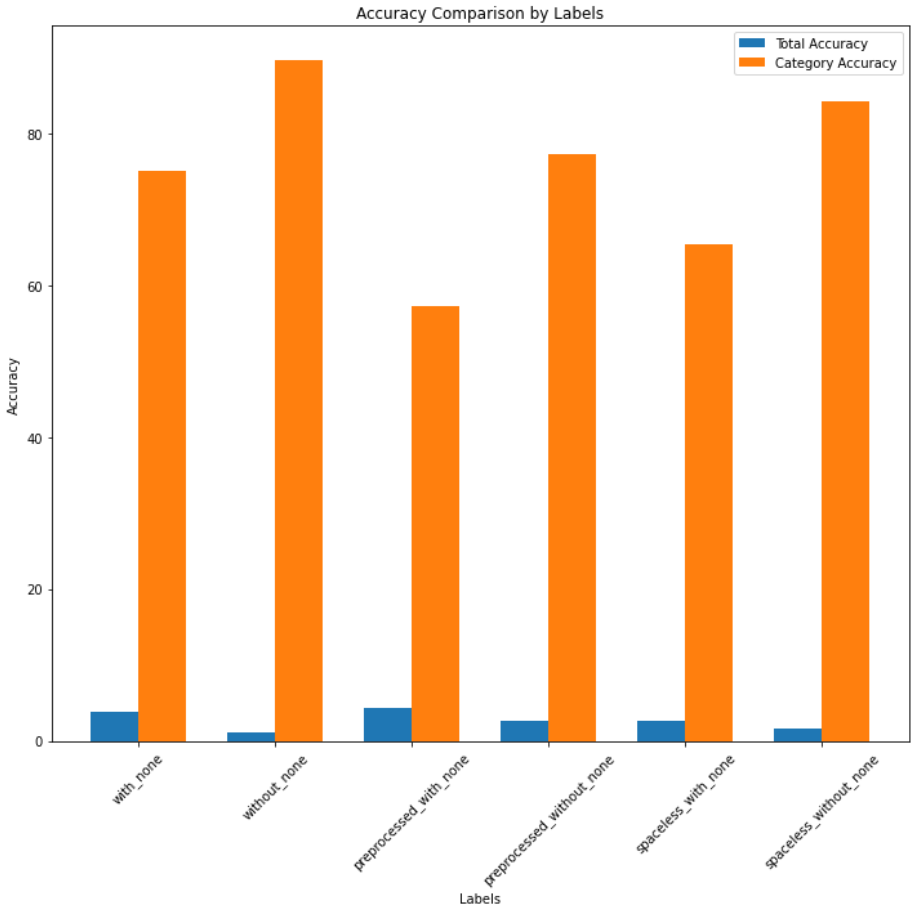

Figure 3: Comparrison of the results for different preprocessed outputs

The results are presented in 3, a full discussion and numerical data is found in the Discussion section.

4 Discussion

We have a set of labels representing different categories: 'with_none', 'without_none', 'preprocessed_with_none', 'preprocessed_without_none', 'spaceless_with_none', and 'spaceless_without_none'.

Each label has an associated accuracy percentage and categorical percentage. The accuracy percentage represents the overall accuracy of the classification

model for that specific label, indicating how often the predicted labels match the true labels. The categorical percentage represents the distribution of accuracy across the categories, indicating the proportion of instances that were accurate in their respective category (mathematical formulation is presented in Methodology)

Let's discuss the findings based on the given data:

1. **'with_none' label**: The accuracy percentage is 3.78%, which suggests that the model's predictions for this label align with the true labels in only a small percentage of cases. The categorical percentage is 75.14%, indicating that a significant portion of instances falls under this category.

2. **'without_none' label**: The accuracy percentage is 1.08%, indicating a low accuracy rate for this label. However, the categorical percentage is 89.73%, suggesting that the majority of instances belong to this category.

3. **'preprocessed_with_none' label**: The accuracy percentage is 4.32%, which is slightly higher than the previous labels. The categorical percentage is 57.30%, indicating a lower representation compared to some other categories.

4. **'preprocessed_without_none' label**: The accuracy percentage is 2.70%, similar to the 'with_none' and 'spaceless_with_none' labels. The categorical percentage is 77.30%, indicating a relatively higher representation.

5. **'spaceless_with_none' label**: The accuracy percentage is 2.70%, same as the 'with_none' and 'preprocessed_without_none' labels. The categorical percentage is 65.41%, suggesting a moderate representation.

6. **'spaceless_without_none' label**: The accuracy percentage is 1.62%, which is the lowest among all the labels. However, the categorical percentage is 84.32%, indicating a significant presence of instances in this category.

Based on these findings, we can observe variations in both accuracy and categorical percentages across the different labels. Some categories have a higher accuracy rate, suggesting better performance in classifying instances correctly. Additionally, the categorical percentages provide insights into the distribution of instances among the categories, allowing us to identify the prevalence of certain categories over others.

It's important to note that further analysis and consideration of the specific context and objectives of the classification task are necessary to draw more meaningful conclusions from these results. However, it is important to note that while the categorical accuracy percentages indicate high values for some labels, they might not provide a complete picture of the model's performance. This is because the accuracy calculation only considers data points where the model provides a response and ignores instances where it generates null responses. The null responses can occur when the model is uncertain or lacks

the necessary information to generate a meaningful response. Consequently, the accuracy percentages might be inflated due to the exclusion of these null response instances, leading to a misleading interpretation of the model's overall performance. Therefore, relying solely on categorical accuracy can be misleading, and additional evaluation metrics that account for null responses should be considered to obtain a more comprehensive assessment of the model's accuracy.

To gain a deeper understanding of the model's performance for each label, it can be beneficial to calculate a multilabel confusion matrix for the sub-labels within each main label. By doing so, we can analyze the classification accuracy, precision, and recall metrics specific to each sub-label. This granular evaluation allows us to identify not only the overall performance of the model for a particular label but also any specific areas of strength or weakness within that label's sub-labels. The multilabel confusion matrix provides a comprehensive visual representation of the true positives, true negatives, false positives, and false negatives for each sub-label, enabling us to assess the model's ability to correctly classify instances across different sub-categories. This information is invaluable in guiding further improvements and fine-tuning of the model, as it highlights specific areas where the model may struggle and provides insights for targeted interventions to enhance its accuracy for individual sub-labels within each main label.

5 Conclusion

In conclusion, this project aimed to address the challenges of detecting and diagnosing performance issues in parallel codes by leveraging the capabilities of large language models, specifically ChatGPT. We successfully developed an automated pipeline that can identify and classify performance problems in parallel codes, contributing to the optimization and efficiency of applications.

The pipeline we constructed consisted of several components, including analyzing the spreadsheet, fetching the code, processing code files, interacting with ChatGPT, and evaluating the results. Through these stages, we were able to gather data from the spreadsheet, retrieve the relevant code from GitHub, preprocess the code to manage token limitations, and interact with ChatGPT to obtain problem classifications.

Despite encountering token constraints imposed by the GptApi, we implemented preprocessing techniques to mitigate the issue. By reducing whitespace and removing comments, we were able to reduce the number of tokens, enabling us to analyze larger and more complex code files. However, it is important to note that the token limitations restricted the inclusion of some files in the classification process. To overcome this constraint, future work could explore the use of GPT-4, which allows for a greater number of tokens per context.

Our pipeline provided promising results in terms of analysis of performance problem classification. By using different versions of code data and system prompts, we were able to evaluate the impact of preprocessing techniques and contextual prompts on the model's performance. Although the results of Chat

GPT on this dataset w This comprehensive approach allowed us to make informed observations and draw meaningful conclusions regarding the model's ability to understand and classify performance problems in parallel codes.

The potential applications of this pipeline are significant. It can be used as a valuable tool for developers and researchers in identifying and addressing performance issues in parallel codes, leading to improved efficiency and optimized applications. Additionally, the insights gained from this project contribute to the broader understanding of the capabilities and limitations of large language models in the domain of parallel computing.

In summary, our automated pipeline demonstrates the potential of leveraging large language models like ChatGPT for performance problem classification in parallel codes. With further improvements and advancements, such pipelines could become valuable assets in the field of high-performance computing, aiding developers and researchers in optimizing their parallel code implementations.