CMSC416: Assignment 3 Report

Krisnajit Rajeshkhanna

18 October 2024

1 Introduction

In this report, we will cover the performance analysis of the LULESH application using the Hatchet Python library. The goal was to identify the most time-consuming functions and analyze the load imbalance across MPI processes. Hatchet, a performance analysis tool, was used to inspect and manipulate hierarchical performance data. The following sections outline the steps taken during the analysis, the Hatchet functions that were the most useful, and a discussion of the results.

2 Data and Tools

The performance data used for this analysis was collected from multiple runs of LULESH on 64 MPI processes. The profiles were gathered using Caliper, and Hatchet was used to analyze the performance data. The datasets include metrics such as execution time, MPI communication time, and load imbalance.

The performance profiles were available in JSON format, generated from runs of LULESH using 1, 8, 27, and 64 processes. Through Hatchet, the python library, users are able to manipulate and analyze graphs of performance metrics, making it an essential tool for this kind of analysis.

3 Process and Analysis

3.1 General Setup

In each problem, the first step was to load the performance data into a GraphFrame object using the `from_caliper()` function:

```
gf = ht.GraphFrame.from_caliper("lulesh-<num>cores.json")
```

This function reads the performance data from the JSON file, parses it, and stores it in the GraphFrame object. The `dataframe` attribute of the GraphFrame is then used to manipulate and analyze the data. The performance metrics (such as time, imbalance, etc.) are accessed as columns in the dataframe.

3.2 Problem 1: Identifying the Top N Functions by Exclusive Time

In Problem 1, the goal was to identify the top N functions where the code spends the most exclusive time. This was achieved by sorting the functions based on the `time` column in descending order and selecting the top N entries. Here, the following key steps were taken:

- `gf.dataframe` was used to access the dataframe of performance metrics.
- `sort_values()` was applied to sort the functions by their exclusive time, which is stored in the `time` column.
- The top N functions were selected using `head(N)` and printed along with their exclusive time.

The output was a list of N functions and their corresponding exclusive time, allowing us to identify which parts of the code were the most time-consuming.

3.3 Problem 2: Identifying the Function with the X-th Highest Load Imbalance

In Problem 2, the task was to identify the function with the X-th highest load imbalance across MPI processes. The load imbalance was calculated using Hatchet's `load_imbalance()` function, and the dataframe was sorted based on the imbalance metric. The process involved:

- Using `load_imbalance()` to calculate the load imbalance for each function.
- Sorting the dataframe by the `time.imbalance` column to rank functions by their imbalance.
- Selecting the X-th function from the sorted dataframe using `iloc[X-1]`.
- Printing the list of MPI ranks (processes) that contributed most to the load imbalance for the selected function.

This analysis helped pinpoint the functions where the computational load was unevenly distributed across processes, potentially leading to performance bottlenecks.

3.4 Problem 3: Identifying Functions with the Largest Time Difference Between Runs

Problem 3 involved comparing the performance of LULESH when run on 8 processes and 64 processes. The goal was to identify functions with the largest time differences between the two runs. In this problem:

- Two GraphFrames were created for the 8-process and 64-process runs.

- The `drop_index_levels()` function was used to remove unnecessary index levels in the GraphFrames to ensure they could be compared.
- The difference between the two runs was calculated by subtracting the two GraphFrames (`gf64 - gf8`).
- Functions with positive time differences were identified, sorted, and printed.

The output was the list of N functions that showed the largest increase in time between the 8-process and 64-process runs, allowing us to analyze how the performance of different parts of the code scaled with the number of processes.

4 Key Hatchet Functions

During this analysis, several Hatchet functions proved to be particularly useful:

- `from_caliper()`: This function was crucial for loading the Caliper-generated JSON performance data into a GraphFrame object. It automatically parses the hierarchical structure of the data.
- `tree()`: This function generates a hierarchical view of the call graph, allowing a structured representation of the function call relationships and their corresponding performance metrics. It was useful for gaining a high-level overview of the execution flow.
- `to_dot()`: This function outputs the call graph in Graphviz's DOT format, which can then be visualized as a graph using tools like `dot` or `neato`. It provided an easy way to visualize the call structure and detect performance bottlenecks visually.
- `load_imbalance()`: This function calculated the load imbalance for each function across the MPI ranks. The imbalance information was essential for identifying hotspots in the application.
- `dataframe()`: The dataframe attribute allowed direct access to the performance metrics in a Pandas dataframe, making it easy to manipulate and analyze the data.