DSC 204A: Scalable Data Systems

Programming Assignment 1

January 2024

1 Introduction

The objective of this programming assignment is to acquaint you with processing large datasets using Ray and Modin. Task 1 guides you in setting up Ray and Modin on your local system. Task 2 introduces you to computing descriptive statistics on a large dataset and parallelizing fundamental dataframe operations. Task 3 aims to enhance your comprehension of the Ray Core API and parallelizing computations.

2 Task 1: Setting up Ray and Modin on your local machine

In this section, we show how to set up Ray and Modin on your local machine. For Windows users, we recommend using WSL to operate a Linux environment on your machine instead of Windows itself. This is because Ray is still in its beta phase for Windows. The setup instructions for WSL are available here.

We suggest that you use Anaconda to create an environment with it's Python version set to 3.10 or lower (Ray on Python 3.11 and above is experimental, and may cause issues). The instructions on Anaconda installation can be found here.

Once Anaconda is installed, create a new environment:

```
conda create -n ray_env python=3.10
```

and activate it:

```
conda activate ray_env
```

Once you're in the `ray_env` environment, install Ray:

```
pip install ray==2.9.1
```

To check if Ray has been installed successfully, you can run the following command -
`python -c "import ray; print(ray.__version__)"`

Similarly, to install Modin, use the command -

```
pip install modin
```

and verify it's version out as well, similar to how we checked the Ray installation.

3 Task 2: Data Manipulation with Modin (45)

In this question, you will be required to perform some basic data manipulation with Modin (Pandas on Ray). Modin is a library that allows users to perform Pandas workloads at scale. In this assignment, we will focus on parallelizing dataframe operations with Modin.

We will use a subset of the Amazon Reviews dataset. Please download the dataset from here. The dataset has the schema shown in Table 1

Column name	**Column description**
reviewerID	ID of the reviewer
vote	Helpful votes of the review
overall	Rating of the product
unixReviewTime	Time of the review (unix time)
reviewTime	Time of the review (raw)

Table 1: The Amazon Reviews Dataset

3.1 Task 2.1

Perform a few data manipulation operations on this dataset and generate a new table with the schema shown in Table 2. We have provided expected output statistics for you so that you can verify your work. Please use the output function we provided to save the table you generated.

Column name	**Column description**
reviewerID	ID of the reviewer
num_product_reviewed	Total number of products reviewed by this reviewer
mean_rating	The average rating this reviewer has given across all reviewed products
latest_review_year	The latest year this reviewer has given a review
num_helpful_votes	The total number of helpful votes this reviewer has gotten

Table 2: Schema for the processed dataframe

3.2 Task 2.2

Change the number of cpus used by Modin or the Ray backend (documentation here) on your instance and run your data manipulation code. Document the execution times you see with 2, 3 and 4 CPUs. Is it a linear speedup? If not, why?

3.3 Grading Rubric

For Task 2.1, there are 4 columns (apart from the ID) in the output table. If all descriptive stats (mean, std dev, min, and max) with 1% error margin match the ground truth, we award 10 points per column. Task 2.2 is worth 5 points.

4 Task 3: The Ray Core API (55)

This question will focus on becoming familiar with the Ray Core API, with the three key abstractions: Ray tasks, Ray actors and Ray objects. Strictly speaking, you only need to know about parallelizing computations with Ray tasks to solve this question.

4.1 Task 3.1

We will implement a distributed merge sort algorithm in Python. The standard merge sort algorithm uses a divide and conquer strategy to partition a given sequence into two halves, and then recursively sorts these two halves. The crucial phase is the merge step, where two sorted halves are merged to get a final sorted list. We will focus on a slightly modified version of the algorithm :

- In the first stage, the input sequence is partitioned into 4 subsequences.
- Next, each subsequence is sorted through a call to the standard merge sort algorithm (implemented with the two-pointer approach).
- Finally, the 4 sorted subparts are merged using a heap-based merge algorithm.

A plain implementation of this algorithm is provided to you (*plain_merge_sort*). Your task is to use Ray to parallelize computations to utilize all 4 CPUs (*ray_merge_sort*). The file to modify is `merge_sort_ray.py`. Parts of the code that can and cannot be modified are also highlighted in the comments. Your goal should be to get a speedup (measured as the ratio of time taken for plain sorting/ time taken with Ray) of around 1.64 (but this can be higher depending on the machine; we observe around 2.0 speedup on DataHub).

P.S: You're not supposed to make algorithmic changes here (i.e change the heap-based algorithm to something else, etc). Focus on what tasks can and can't be parallelized and use Ray. For sorting calls within your Ray implementation, you should use *plain_merge_sort* - this is to clearly see what speedup you get with Ray. Using the built-in Python sorting function `sorted` in *ray_merge_sort* will, of course, give you a speedup with simply algorithmic improvements (which is not our intention).

4.2 Task 3.2

The ideal speedup you can get for a task with 4 workers is, of course, 4. Can you estimate the theoretical maximum speedup you can get for the above merge sort algorithm (in terms of the time for sorting sublists, and the time for merging)? How do you account for the difference between theoretical result and the observed speedup with Ray?

4.3 Grading Rubric

Task 3.2 is worth 5 points. For Task 3.1, your merge sort code will first be checked for accuracy: We expect the code to accurately sort the given list. Incorrect code will not receive any points. For runtime, we will run your scripts thrice and get the average runtime. We will use the following rubric for runtime:

Speedup	Score
Atleast 1.5	50
Between 1.3 and 1.5	40
Between 1 and 1.3	20
1 or less	0

Table 3: Rubric for Task 3.1

5 Deliverables

For Task 2.1 and 3.1, you will need to fill in the code in the respective .py files. For Task 2.2 and 3.2, please write down your answers in a document titled "Report" (can be a word document or latex formatted) and submit it along with your code.

6 Tips

- Since you are running the assignment locally, it may require a substantial amount of RAM. We suggest that you do not have any other RAM-heavy processes running on your computer while your code is running, to avoid Out-of-Memory errors. In the event that Out-of-Memory errors persist despite your code being the sole process running on your machine, please reach out to the TAs during their office hours.
- While testing your code, it might be helpful to operate on a smaller version of the given data (for example, a smaller list in Task 3) in order to catch errors and iterate faster.
- Use separate columns for raw and processed data in Task 2 to prevent accidental data corruption.
- For Task 3, we expect you to analyse the *plain_merge_sort* algorithm and modify it to add in parallelization with Ray. Nothing more!