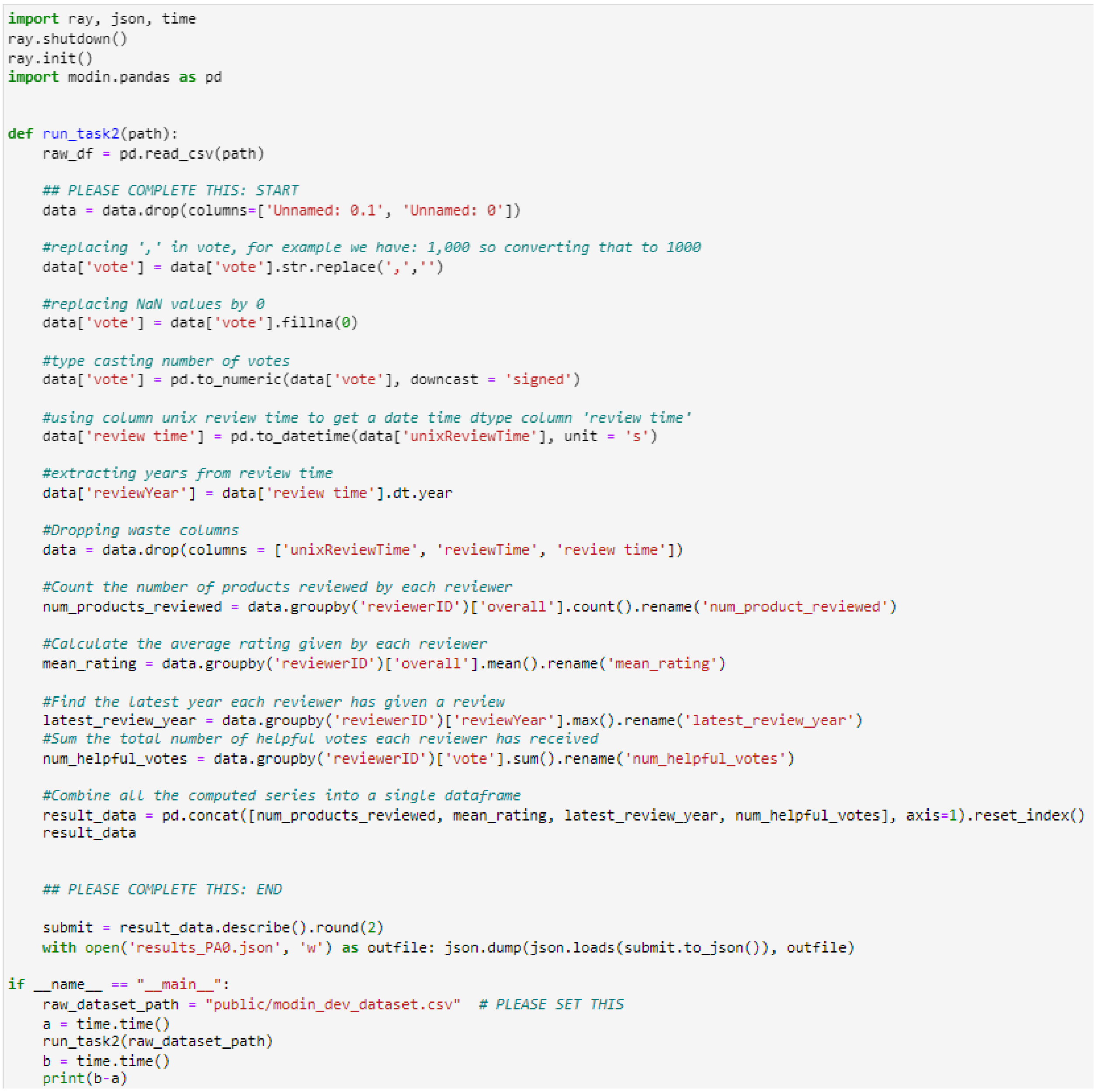
Programming Assignment-1 Report

Abhisheik Jadhav
PID: A69027702

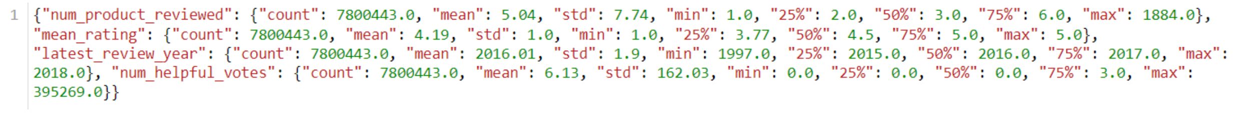
Task 2.1:

Code:



```
import ray, json, time
ray.shutdown()
ray.init()
import modin.pandas as pd


def run_task2(path):
    raw_df = pd.read_csv(path)

    ## PLEASE COMPLETE THIS: START
    data = data.drop(columns=['Unnamed: 0.1', 'Unnamed: 0'])

    #replacing ',' in vote, for example we have: 1,000 so converting that to 1000
    data['vote'] = data['vote'].str.replace(',','')

    #replacing NaN values by 0
    data['vote'] = data['vote'].fillna(0)

    #type casting number of votes
    data['vote'] = pd.to_numeric(data['vote'], downcast = 'signed')

    #using column unix review time to get a date time dtype column 'review time'
    data['review time'] = pd.to_datetime(data['unixReviewTime'], unit = 's')

    #extracting years from review time
    data['reviewYear'] = data['review time'].dt.year

    #Dropping waste columns
    data = data.drop(columns = ['unixReviewTime', 'reviewTime', 'review time'])

    #Count the number of products reviewed by each reviewer
    num_products_reviewed = data.groupby('reviewerID')['overall'].count().rename('num_product_reviewed')

    #Calculate the average rating given by each reviewer
    mean_rating = data.groupby('reviewerID')['overall'].mean().rename('mean_rating')

    #Find the latest year each reviewer has given a review
    latest_review_year = data.groupby('reviewerID')['reviewYear'].max().rename('latest_review_year')
    #Sum the total number of helpful votes each reviewer has received
    num_helpful_votes = data.groupby('reviewerID')['vote'].sum().rename('num_helpful_votes')

    #Combine all the computed series into a single dataframe
    result_data = pd.concat([num_products_reviewed, mean_rating, latest_review_year, num_helpful_votes], axis=1).reset_index()
    result_data


    ## PLEASE COMPLETE THIS: END

    submit = result_data.describe().round(2)
    with open('results_PA0.json', 'w') as outfile: json.dump(json.loads(submit.to_json()), outfile)

if __name__ == "__main__":
    raw_dataset_path = "public/modin_dev_dataset.csv"  # PLEASE SET THIS
    a = time.time()
    run_task2(raw_dataset_path)
    b = time.time()
    print(b-a)
```

JSON file:



```
{"num_product_reviewed": {"count": 7800443.0, "mean": 5.04, "std": 7.74, "min": 1.0, "25%": 2.0, "50%": 3.0, "75%": 6.0, "max": 1884.0},
"mean_rating": {"count": 7800443.0, "mean": 4.19, "std": 1.0, "min": 1.0, "25%": 3.77, "50%": 4.5, "75%": 5.0, "max": 5.0},
"latest_review_year": {"count": 7800443.0, "mean": 2016.01, "std": 1.9, "min": 1997.0, "25%": 2015.0, "50%": 2016.0, "75%": 2017.0, "max":
2018.0}, "num_helpful_votes": {"count": 7800443.0, "mean": 6.13, "std": 162.03, "min": 0.0, "25%": 0.0, "50%": 0.0, "75%": 3.0, "max":
395269.0}}
```

Speed Ups Observed:

2 CPUs = 400.3335061073303
3 CPUs = 313.03131890296936
4 CPUs = 291.9563617706299

Task 2.2:
The observed execution times did not exhibit linear speedup. In our case, when transitioning from 2 to 4 CPUs, the time decreased from 400.33 seconds to 291.95 seconds, which is not a 50% reduction.

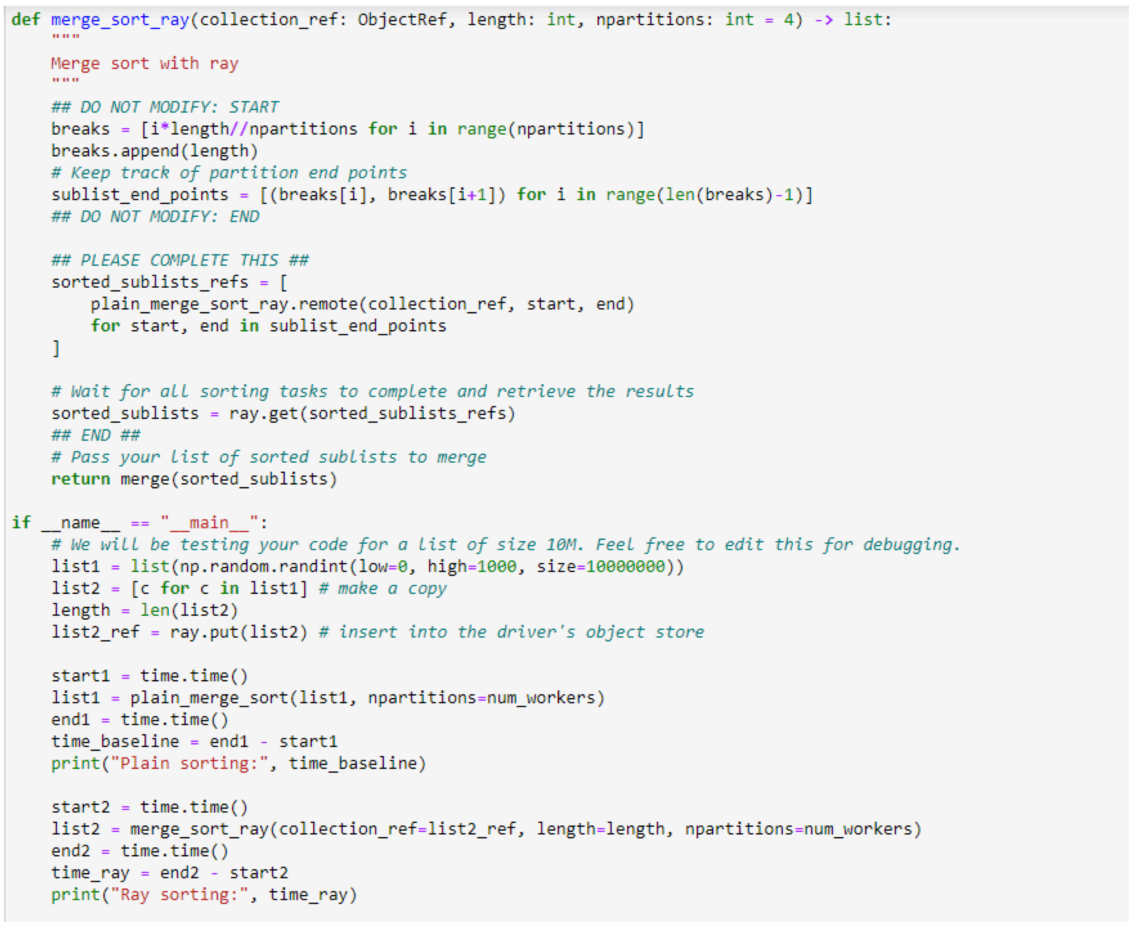
Several factors contribute to the lack of linear speedup:

1. Communication Overhead: As the number of CPUs increases, so does the coordination overhead between them. This includes tasks such as dividing the data, sending instructions, and gathering results, which can introduce inefficiencies and hinder performance improvements.
2. Inefficient Parallelization: Not all sections of the code may be effectively parallelized. The overall speedup can be limited if certain parts rely on sequential dependencies or have limited parallelizable components.
3. Resource Saturation: Other system bottlenecks, such as memory bandwidth or disk I/O, may not scale proportionally with the number of CPUs. These bottlenecks can limit the overall performance gains achievable through parallelization.
4. Diminishing Returns: As more CPUs are added, the incremental performance improvements may diminish. Each additional CPU may contribute less to the overall speedup due to the factors above, resulting in diminishing returns as more resources are allocated.

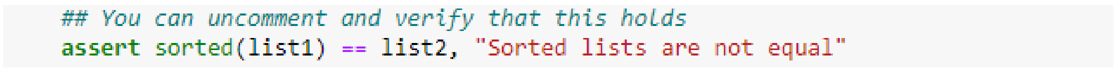
Task 3.1:

Code:

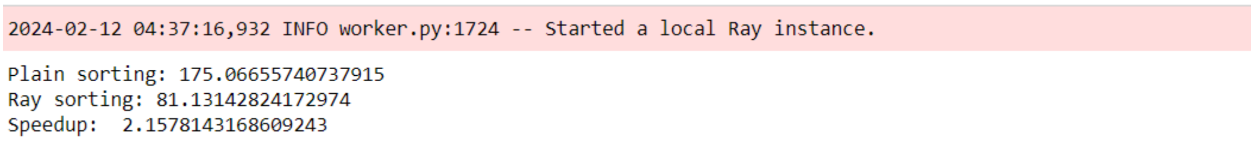
```python
ray.shutdown()
import heapq
from typing import List
import ray
from ray import ObjectRef
from plain_merge_sort import plain_merge_sort
import time
import numpy as np

## RAY INIT. DO NOT MODIFY
num_workers = 4
ray.init(num_cpus=num_workers)
## END OF INIT

## Feel free to add your own functions here for usage with Ray

@ray.remote
def plain_merge_sort_ray(collection_ref, start, end):
    # No need to call ray.get here; it will be called inside the remote function
    sublist = collection_ref[start:end]  # This slicing will be on the list, not on the ObjectRef
    return plain_merge_sort(sublist)

def merge(sublists: List[list]) -> list:
    """
    Merge sorted sublists into a single sorted list.

    :param sublists: List of sorted lists
    :return: Merged result
    """
    ## YOU CAN MODIFY THIS WITH RAY
    result = []
    sublists = [sublist for sublist in sublists if len(sublist)> 0]
    heap = [(sublist[0], i, 0) for i, sublist in enumerate(sublists)]
    heapq.heapify(heap)
    while len(heap):
        val, i, list_ind = heapq.heappop(heap)
        result.append(val)
        if list_ind+1 < len(sublists[i]):
            heapq.heappush(heap, (sublists[i][list_ind+1], i, list_ind+1))
    return result
```



```python
def merge_sort_ray(collection_ref: ObjectRef, length: int, npartitions: int = 4) -> list:
    """
    Merge sort with ray
    """
    ## DO NOT MODIFY: START
    breaks = [i*length//npartitions for i in range(npartitions)]
    breaks.append(length)
    # Keep track of partition end points
    sublist_end_points = [(breaks[i], breaks[i+1]) for i in range(len(breaks)-1)]
    ## DO NOT MODIFY: END

    ## PLEASE COMPLETE THIS ##
    sorted_sublists_refs = [
        plain_merge_sort_ray.remote(collection_ref, start, end)
        for start, end in sublist_end_points
    ]

    # Wait for all sorting tasks to complete and retrieve the results
    sorted_sublists = ray.get(sorted_sublists_refs)
    ## END ##
    # Pass your list of sorted sublists to merge
    return merge(sorted_sublists)

if __name__ == "__main__":
    # We will be testing your code for a list of size 10M. Feel free to edit this for debugging.
    list1 = list(np.random.randint(low=0, high=1000, size=10000000))
    list2 = [c for c in list1] # make a copy
    length = len(list2)
    list2_ref = ray.put(list2) # insert into the driver's object store

    start1 = time.time()
    list1 = plain_merge_sort(list1, npartitions=num_workers)
    end1 = time.time()
    time_baseline = end1 - start1
    print("Plain sorting:", time_baseline)

    start2 = time.time()
    list2 = merge_sort_ray(collection_ref=list2_ref, length=length, npartitions=num_workers)
    end2 = time.time()
    time_ray = end2 - start2
    print("Ray sorting:", time_ray)
```



```python
    ## You can uncomment and verify that this holds
    assert sorted(list1) == list2, "Sorted lists are not equal"
```



```
2024-02-12 04:37:16,932 INFO worker.py:1724 -- Started a local Ray instance.

Plain sorting: 175.06655740737915
Ray sorting: 81.13142824172974
Speedup:   2.1578143168609243
```

Task 3.2:

The theoretical maximum speedup for parallelization of the merge sort algorithm, in terms of sorting sublists and merging, can be estimated by analyzing the inherent computational complexity of the algorithm and the potential for parallel execution.

The merge sort algorithm entails two primary operations: sorting sublists and merging sorted sublists. Sorting sublists of size n typically exhibits a time complexity of O(nlogn), while merging two sorted lists of size each has a time complexity of O(n).

In an ideal parallel scenario, the sorting of sublists can be distributed across multiple workers, potentially reducing the overall sorting time to O(n/logn) with perfect parallelism. Similarly, merging operations can be parallelized to a certain extent, albeit with dependencies between merging tasks that can limit the achievable speedup.

The theoretical maximum speedup S is derived by considering the total workload W required for sorting a list of size n. In a sequential implementation, the time complexity is Tseq = O(W). In contrast, in an ideal parallel implementation with p workers, the time complexity becomes Tpar = O(W/p). Therefore, the theoretical maximum speedup can be expressed as:
S = Tpar/Tseq = W/(W/p) = p

This relationship illustrates that the theoretical maximum speedup is linearly proportional to the number of workers employed.

Achieving the theoretical maximum speedup is difficult due to factors such as:

- Communication overhead: There is overhead associated with distributing tasks among workers and collecting results.
- Synchronization: Workers may need to synchronize their operations, which can introduce delays.
- Resource contention: If resources such as CPU cores or memory are limited, adding more workers may not lead to proportional speedup due to contention.
- Some parts of the parallel code can only run serially, i.e. they can’t be parallelized, so that doesn't allow ideal speed-up.