File index

1.py: Python code for solving the first problem
1_o.py: Optimized code for solving the first problem
1_n.py: Python code for solving the first problem with the ability to set number of processors and reducers

2.py: Python code for solving the second problem
2_o.py: Optimized code for solving the second problem
2_n.py: Python code for solving the second problem with the ability to set number of processors and reducers

Output paths:
Problem 1 output path: /gpfs/home/aabhilash/results_1.txt
Problem 2 output path: /gpfs/home/aabhilash/results_2.txt

P.s. adding output path instead of the results file since Seawulf cluster is not letting logins or downloads with the connection reset error

Problem 1
1) Implemented in 1.py
2)Implemented in 1_n.py : to take the number of mappers and reducers programmatically as num_mappers and num_reducers in the code.

Since I am running on seawulf with mpi, the total number of mappers or reducers are limited by the total number of cores available*number of threads in each core: for the cpu i am running, it is 48 cores*2threds. So Mapper experimentation cannot go beyond 96.

Analysis : More mappers resulted in faster code. Since there is better distribution. Good results at 90+, but the spill over data chunk causing more time overhead.

10 reducers giving fastest results, no of reducers to be left under # of mappers.

For # of mappers or #reducers in power of 2, faster results since better distribution of data.

3) and 4)

In the original code, in each mapper, we are taking the mapping of customer id and the corresponding amount, making it a list of tuple and sending it to the reducer to do the aggregation. But in the optimized code, we do the stepwise aggregation at the mapper itself, such that for a given customer id, all the amounts for that customer id are already summed and only the final sum is sent. The reducer then sums up the part sums. This will decrease the communication overhead since less items to send. Also this brings down the time complexity since less dictionary usage and streamlined aggregation.The result was a faster output.

Problem 2
1)Implemented in 2.py
2)Impleemnted in 2_n.py: to take the number of mappers and reducers programmatically as num_mappers and num_reducers in the code.
ince I am running on seawulf with mpi, the total number of mappers or reducers are limited by the total number of cores available*number of threads in each core: for the cpu i am running, it is 48 cores*2threds. So Mapper experimentation cannot go beyond 96.
I noticed that increasing the mappers increased the speed a little but increasing the reducers beyond a certain number didn't increase the speed. There being a saturation in the number of unique items with respect to A

3) and 4) Optimization Strategy and Analysis
The new code 2_o.py will have the following betterments over the old code 2.py
a)Efficient Data Distribution
The original code had division into chunks without considering remainders. This could lead to some process having more data causing imbalance
Changes: Lines 99 to 102

```
chunks_per_process = len(shuffled_data) // size
remainder = len(shuffled_data) % size
scattered_chunks = [shuffled_data[i*chunks_per_process + min(i, remainder):(i+1)*chunks_per_process + min(i+1, remainder)] for i in range(size)]
```

The optimized code calculates chunks_per_process and distributes any leftover data (remainder) evenly among the first few processes. This ensures a more balanced workload

b) Communication overhead
The original approach might involve multiple communication steps. The optimized code gathers all mapped data at the root after mapping and only scatters keys once after grouping

```
all_mapped_data = comm.gather(local_mapped_data, root=0)
if rank == 0:
    flat_mapped_data = [item for sublist in all_mapped_data for item in sublist]
    grouped_data = group_by_keys(flat_mapped_data)
```

```
    scattered_keys = distribute_keys(grouped_data, size)
local_keys = comm.scatter(scattered_keys, root=0)
```

c)Optimized Join
The optimized code uses list comprehensions to streamline join process

```
final_joined_results.extend(
    (key,) + b_c + d_e + f_g
    for b_c in r1
    for d_e in r2
    for f_g in r3
)
```

The new code turned out to be significantly faster with time. With scaling of data, its time saving is only going to increase.