COL733 Lab1 Analysis

Abhishrut Omprakash Vaidya
2021SIY7558

January 23, 2022

- **Q1. Maximum Speedup over serial.py**
  I have calculated speedup for two cases:
  - a. Maximum speedup when we have lesser samples:

    Average $T_s = 61.250$ secs
    Average $T_p = 0.480$ secs
    so, **Speedup** $= \frac{T_s}{T_p} =$ **127.60**

  - b. Speedup when we use all the data:

    Average $T_s = 61.250$ secs
    Average $T_p = 82.748$ secs
    so, **Speedup** $= \frac{T_s}{T_p} =$ **0.74**

- **Q2. Fixed Input size, Varying efficiency w.r.t. increase in worker threads allocated**
  So, let us consider the fixed input size as **500 files**.
  Time taken by **serial.py** is T_s = 18.872 secs and
  Time taken by **serial execution with single thread of client.py** T_s = 38.922 secs

No. of Workers	threads/Worker	with print(secs)	w/o print(secs)	Efficiency(%)
1	4	24.652	12.495	77.87
1	8	23.680	9.414	51.68
1	12	23.418	9.889	32.79
1	16	23.487	11.160	21.79
2	4	22.324	9.809	49.59
2	8	23.211	9.857	24.67
2	12	22.284	9.504	17.06
2	16	23.396	9.760	12.46
3	4	15.980	7.356	44.09
3	8	15.327	7.484	21.66
3	12	15.516	7.241	14.93
3	16	17.095	6.985	11.60
4	4	16.80	8.657	28.10
4	8	15.978	8.359	14.55
4	12	15.515	8.253	9.82
4	16	16.685	9.214	6.60

In the above table efficiency is calculated using the time which in which program doesn't print the output as it was being big threshold and wasn't utilizing concurrency.
So, the efficiency we get is per worker thread.

$$\text{Speedup} = \frac{T_s}{T_p}$$

$$\textbf{Efficiency} = \frac{Speedup}{number of threads allotted to application}$$

Analysis:
From the above table it is seen that as the number of threads are less the efficiency per thread is more but program is taking more time.
For a give number of workers, the time taken by program in concurrent execution goes down up-to the point and then increases a bit in many cases. Like when no. of workers = 1 or 4, in both cases time take was going down till number of threads 12 then time taken increases when thread count becomes 16. The reason for this could be as efficiency is going down, some toll is being incurred for the threads management also in the background by celery, as more time incurs in context switch of worker than of thread.

- **Q3. Fixed worker thread = 8 allocated to application, observe varying efficiency w.r.t. input size** We need to allocate **8 threads** to application and the recorded time and efficiency reading is shown below table:

No of I/P files	Serial	1 W, 8 T	Eff 1-8	2 W, 4 T	Eff 2-4
10	1.079	3.532	3.81	0.889	15.17
100	6.254	4.477	17.44	3.957	19.75
500	38.922	16.758	29.03	19.246	25.27
1000	82.399	31.150	33.06	37.135	27.73
1500	125.495	54.183	28.95	60.016	26.13
2000	175.945	70.478	31.20	77.535	28.36
3003	271.258	114.880	29.51	135.903	24.94

No of I/P files	Serial	4 W, 2 T	Eff 4-2	8 W, 1 T	Eff 8-1
10	1.079	0.972	13.87	1.098	12.28
100	6.254	3.511	22.26	3.836	20.37
500	38.922	14.745	32.99	14.843	32.77
1000	82.399	27.519	37.42	29.283	35.17
1500	125.495	44.785	35.785	47.431	33.07
2000	175.945	76.735	28.66	76.688	28.67
3003	271.258	126.798	26.74	103.535	32.74

Analysis:
From above table we can see that as the number of input files increases time taken by the application increases. Same goes for both serial and parallel execution. As respect to the efficiency the trend is seen that as the number of input files increases the efficiency per worker also increases up till some point and later on it reduces a bit. Also, one other observation is that with single worker having more number of threads is more efficient in general than more number of workers having lesser number of threads per worker. This may be due to the context switch over head of worker is more than that of a thread.

- **Q4. Is it Scalable?**
  Yes, the designed solution is scalable.
  How? So the design consists of **mapper** function and **reducer** function. Based on the availability of the worker threads, that number of either mapper or reducer tasks can be run simultaneously as per the configuration. As we perform all the mapper function, which consist of extracting 'tweet' text data and creating word-count dictionary in 'json' file. One file per mapper task.
  After all the mapper task completes, reducer task picks up the 'json' files and starts merging them in exclusive manner. And Hence, even if the number of input files increases the implemented design will work efficiently.

- **Q5. Is it fault-tolerant?**
  Yes, the designed solution is Fault-tolerant.
  How? As all the mapper and reducer tasks are executed by the 'Chord' method of the Celery, even if some worker goes down, other one can pick up the task from where one has left it off. Even tested on the VM.