COL733- LAB1
Vatsal Agrawal - 2021 MCS 2157

Analysis

Q1 What is the best speed up achieved over serial.py?

Speed up achieved is 0.97 using default config and 1.09 with 2nd config. Serial.py takes on average 36 seconds in real-time (max goes to around 45 seconds). My client.py takes 39 seconds real-time with default parameters with a single worker and 37 seconds real-time with multi worker instances. For the 2nd param, 33 seconds is with a single worker. Config. are described in last. All the results are without printing the whole dictionary. Printing in the file takes an extra 4 sec and in terminal depends upon which terminal is used. 2nd config may give Redis fault due to large flow of data or socket timeout due to memory overflow. Default config works in all cases due to proper further division of data in parts.

Reason for this is because serial.py is written in a way that keywords get directly updated in a dictionary when they are found in a file, and dictionary works on hashing, while in my programme, we read some file set in parallel and create dictionary and then combine those dictionary, this combining of dictionary takes extra time. Thus although works are done in parallel but work is also increased as compared to serial.py because we have to combine dictionary also. Also, we can increase performance by lessening no. of dictionary size, but this will create memory overflow in Redis (may result in socket timeout in exchange of large files) or slow down the process due to data exchange. Same reason can be justified by setting concurrency 1 on client.py (serial version of our programme). Clearly, it will take more time than concurrency 8 (parallel version).

Q2 Given a fixed input size, measure how the efficiency of the word-count application varies with an increase in worker threads allocated to the application. Justify

Increasing the number of workers threads, decreases the time for fixed input size. Also, decreasing the number of worker threads, increases time. From the graph given below, the mathematical relation between worker and time can be found to be of type

$$T(x) = \frac{k1}{x + k2} + k3$$

Here x is the number of workers threads, k1,k2,k3 are constants and T(x) is Time

Reason for this is because as we increase the number of workers threads, more no. of cores of CPU or available resources can be allocated to our programme, and more work can be done in parallel. For example, if there is 2 worker, then 2 tasks can be run in parallel using group command, and if three workers are there, thus three tasks can be run in parallel. This

parallelization of tasks increases speed and decreases time. I have used the group function of celery for parallelization, and divided the input into several tasks which can run in parallel.

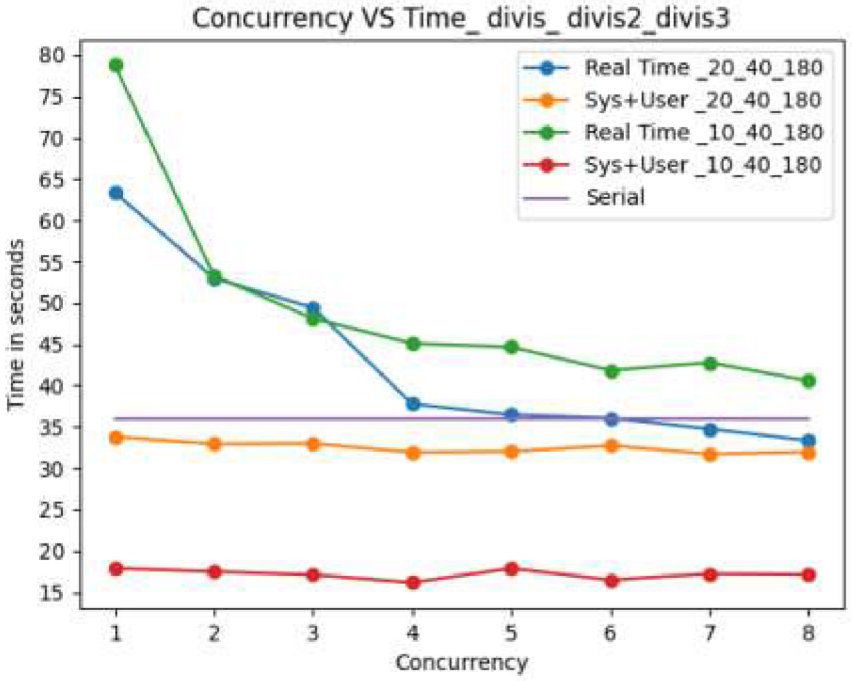

Fig 1 Concurrency Vs Time

In legend divis_divis2_divis3 are configuration parameters.
The default configuration is 10_40_180 (works in all cases).
The table is in last

Q3 Given a fixed worker thread (=8) allocated to the application, measure how the efficiency of the word-count application varies with input size. Justify

Increasing total input size or the number of files, increases time for a fixed worker thread size. Also, decreasing the number of files or total file size, increases time. From the two graphs given (one for input size vs time and another for number of input files vs time) below mathematical relation between input and time can be found to be of linear type

$$T(x) = \mathrm{k1} * \mathrm{x} + k2$$

Here x is the number of files/file size, k1,k2 is constants and T(x) is Time

Reason for this is because as we increase input size, more no of files/data has to be read and more number of computation need to be done in both preparing dictionaries and combining dictionaries. Thus increasing no of tasks will also increase the time as the number of threads is fixed; thus, there can be only eight tasks run in parallel, and additional tasks need to be run in the next parallel time and so on. I have divided my input file size into several pieces to call tasks (in both header and callback) so that they can be parallelized among different workers; so that no workers do single large job, which can increase the time. Also, I have turned prefetching off for to help with the straggler.

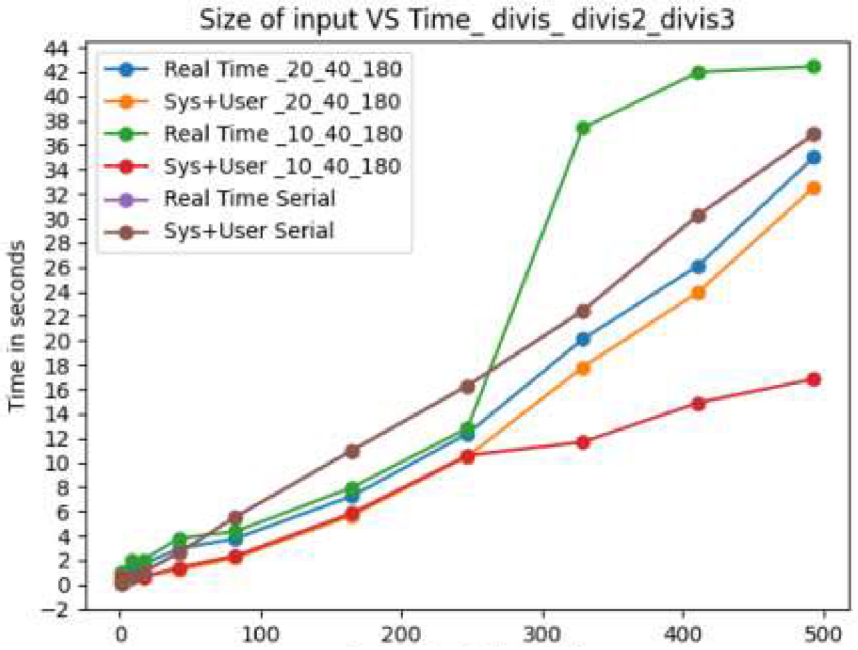

Fig 2 Size of input in MB Vs Time

In legend divis_divis2_divis3 are configuration parameters.
The default configuration is 10_40_180 (works in all cases)
The table is in last

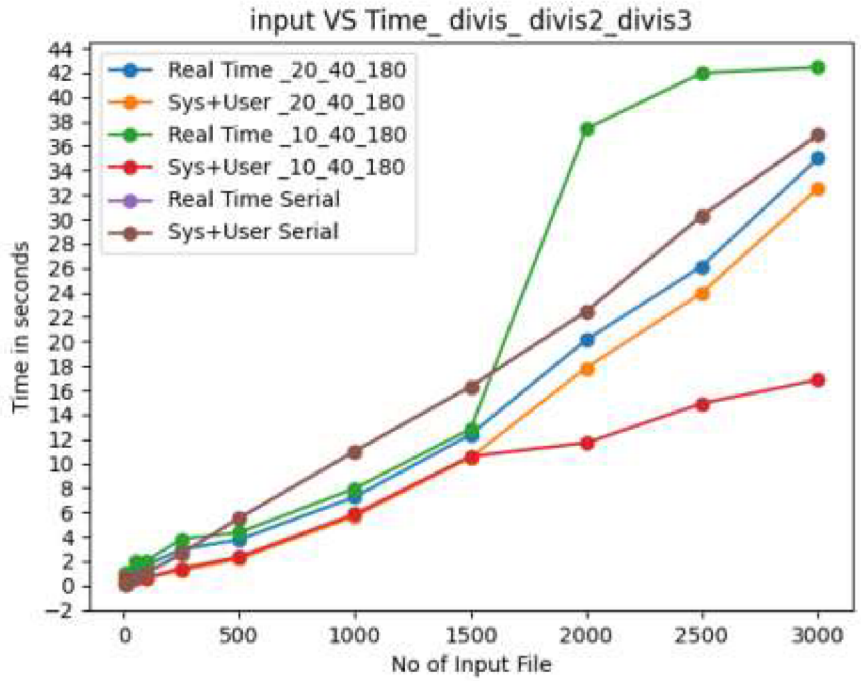

Fig 3 Number of file Vs Time

In legend divis_divis2_divis3 are configuration parameters.
The default configuration is 10_40_180 (works in all cases)
The table is in last

Q4 The designed solution is scalable. Justify.

For Scalable

1) Utilize available resources – Yes, my system utilizes available resources and reduce runtime by dividing the input into smaller subtasks (for both header and callback) which can be run parallel and different resources can use them without hindering others work. I have divided the file list into a smaller set of the file list and then divided these sets of lists into different groups and getting a result, then Passing the result to the callback, which divides the result into a different list and combines them by calling group.

2) Works with large Input – My programme works correctly as we increase inputs size because it divides the input into smaller parts as described above so that memory does not

overflow or timeout does not occur while transferring files to tasks or vice versa. It ensures no large file is handled at all once by a single worker by this division. This also reduces time.

3) Reduce time by increasing resources – Yes, it can be seen clearly from the concurrency vs time graph that when we increase workers, thread time is reduced because of the reason explained in question2.

I have also turned off prefetching to ensure no straggler takes extra work and the task is distributed according to the fastness of the programme

Q5 The designed solution is fault-tolerant. Justify

1) Late Ack – I have ensured that my programme is fault-tolerant by enabling a late ack flag which ensures that task is only removed from the queue when the task is finished. It enables the celery to ensure that it does not send pre ack of receiving task, and if a worker dies, then Rabbitmq can handle the same and reassign the task to the new worker.

2) Soft time limit with retry – It ensures that if some mishap happens with some worker and tasks is not completed on time, then it is retried after the soft limit. It is by enabling a flag of the soft time limit and auto-retry flag.

I have also turned off prefetching to ensure no straggler takes extra work and task is distributed according to the fastness of the programme.

I have also set a hard time limit to ensure some mishap is detected on time.

This fault tolerant is seen in config1.

Tables –

Table 1 - Concurrecy Vs Time 10_40_180

n_thread	Real	user	sys
8	0m39.588s	0m14.914s	0m2.225s
7	0m42.787s	0m15.197s	0m2.041s
6	0m41.844s	0m14.725s	0m1.742s
5	0m44.665s	0m15.732s	0m2.157s
4	0m45.100s	0m14.378s	0m1.778s
3	0m48.112s	0m15.159s	0m1.947s
2	0m53.309s	0m15.546s	0m1.962s
1	1m18.797s	0m15.394s	0m2.500s

Table 2 - Concurrecy Vs Time 20_40_180

n_thread	Real	user	sys
8	0m33.344s	0m29.829s	0m2.136s
7	0m34.755s	0m29.747s	0m1.975s
6	0m36.071s	0m30.765s	0m1.991s
5	0m36.478s	0m30.141s	0m1.888s
4	0m37.769s	0m30.050s	0m1.864s
3	0m49.428s	0m30.608s	0m2.385s
2	0m53.013s	0m30.968s	0m1.971s
1	1m3.383s	0m31.766s	0m2.037s

Table 3 – Input file Vs Time 10_40_180

n	real	user	sys
3003	0m38.442s	0m14.827s	0m2.029s
2500	0m36.987s	0m12.715s	0m2.203s
2000	0m35.411s	0m10.434s	0m1.260s
1500	0m12.820s	0m9.771s	0m0.821s
1000	0m7.964s	0m5.427s	0m0.440s
500	0m4.348s	0m2.038s	0m0.314s
250	0m3.807s	0m1.188s	0m0.227s
100	0m2.044s	0m0.560s	0m0.080s
50	0m2.055s	0m0.510s	0m0.083s
10	0m0.972s	0m0.400s	0m0.064s
2	0m0.955s	0m0.377s	0m0.072s

Table 4 – Input file Vs Time 20_40_180

n	real	user	sys
3003	0m35.007s	0m30.418s	0m2.127s
2500	0m26.153s	0m22.256s	0m1.723s
2000	0m20.128s	0m16.298s	0m1.537s
1500	0m12.379s	0m9.783s	0m0.714s
1000	0m7.259s	0m5.269s	0m0.402s
500	0m3.769s	0m1.998s	0m0.197s
250	0m2.949s	0m1.081s	0m0.149s
100	0m1.703s	0m0.563s	0m0.104s
50	0m1.132s	0m0.511s	0m0.073s
10	0m1.015s	0m0.421s	0m0.083s
2	0m0.960s	0m0.392s	0m0.060s

Table 5 – Input file Vs Time Serial

n	real	user	sys
3003	0m36.930s	0m32.334s	0m4.550s
2500	0m30.297s	0m26.370s	0m3.909s
2000	0m22.453s	0m20.699s	0m1.749s
1500	0m16.308s	0m15.263s	0m1.041s
1000	0m11.015s	0m9.563s	0m1.438s
500	0m5.537s	0m4.791s	0m0.745s
250	0m2.605s	0m2.295s	0m0.309s
100	0m1.049s	0m0.941s	0m0.108s
50	0m0.615s	0m0.559s	0m0.057s
10	0m0.148s	0m0.136s	0m0.013s
2	0m0.057s	0m0.052s	0m0.005s

Table 6 – Input file Vs Size

n	File-size
3003	493M
2500	411M
2000	329M
1500	247M
1000	165M
500	082M
250	042M
100	017M
50	8.3M
10	1.7M
2	0.327M

Table 7 – Workers Instance Vs Time

Worker	real	user	sys
	sys		
1	0m39.926s	0m15.307s	0m2.231s
2	0m37.296s	0m14.113s	0m1.991s
3	0m37.669s	0m15.181s	0m1.852s
4	0m37.389s	0m14.495s	0m1.953s
5	0m38.811s	0m14.397s	0m1.755s
6	0m40.307s	0m14.832s	0m2.053s

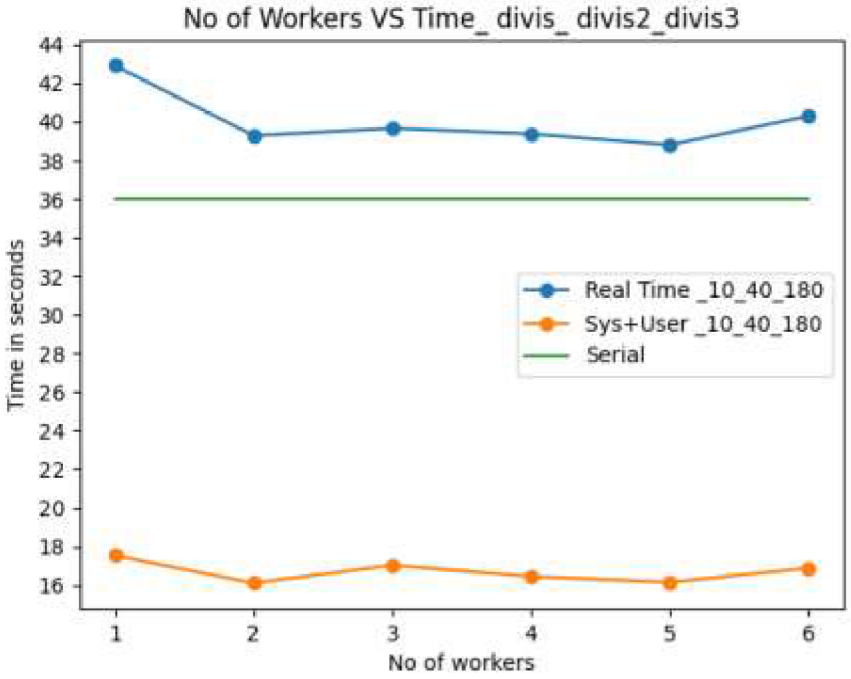

Fig 4 Number of Workers instance of concurrency 8 Vs Time

In legend divis_divis2_divis3 are configuration parameters.
The default configuration is 10_40_180 (works in all cases)

Output-

```
student@baadalvm:~/2021MCS2157_Vatsal2$ time python3 client.py ~/data
2722459

real    0m39.735s
user    0m14.622s
sys     0m1.870s
student@baadalvm:~/2021MCS2157_Vatsal2$
```

Fig 5 Default Config 10_40_180 Single Worker

```
student@baadalvm:~/2021MCS2157_Vatsal2$ time python3 client.py ~/data
2722459

real    0m37.634s
user    0m14.588s
sys     0m1.828s
```

Fig 6 Default Config 10_40_180 Multiple Worker

```
student@baadalvm:~/2021MCS2157_Vatsal2$ time python3 client.py ~/data
2722459

real    0m34.299s
user    0m29.839s
sys     0m1.985s
student@baadalvm:~/2021MCS2157_Vatsal2$
```

Fig 7 Config 20_40_180 Single Worker

Declaration:-

I, Vatsal Agrawal, 2021MCS2157, has made this programme independently without taking help, on baadalvm provided.
I have not used any external library except provided with VM.
serial.py is the same as that provided, I have used some part of it in my client.py and tasks.py
I have verified my result to be True using test.py for the entire dataset provided, containing 3003 CSV files.
The programme is working for single and multi-worker instances with different concurrency on **configuration 1 with an average time of 37 seconds on multi worker 39 seconds with a single worker, and 33second with a single worker with 2nd config**. The lowest time observed on a single worker with different config is 30 sec
My programme is also fault-tolerant and scalable, as tested and described.
All the time and data are without printing the whole final result. It takes 10 minutes extra approx. to print all results in baadalvm terminal webpage.
1st config (default submitted) works in all cases and 2nd config with single worker only (any concurrency) (due to memory overflow in Redis)
Printing output times vary a lot, so I have not used it.
Please read readme with submission code for further working and understanding parameters

Note (Approach) :-

1) 2 of the best configuration is (10,40,180)=(divis, divis2, divis3) and (20,40,180)=(divis, divis2, divis3)
2) Default Configuration is (10,40,180)=(divis, divis2, divis3)
3) Configuration can be changed in client.py
4) divis3 divides no. of total CSV files in the individual list containing divis3 number of files each. This will be given to each task.
5) divis divides no. of the individual list further on which group can be called. The total maximum of the file given to the group without using get is divis * divis3
6) divis2 is the no. of dictionary list that can be executed in one callback function without using get, using group.
7) Average Serial Time observed is 36 seconds.
8) (10,40,180)=(divis, divis2, divis3) configuration works best with multi workers instances, and its average time is 37 seconds real-time with multi worker and 39 seconds with a single worker with concurrency 8.
9) (20,40,180)=(divis, divis2, divis3) configuration works best with single workers instance, and its average time is 33 seconds real-time with a single worker with concurrency 8. **Multiple workers fail on Redis part due to memory overflow in sending and dividing large files and then getting those large files and combining them.**
10) speed up acheived is 1.09 with (20,40,180)=(divis, divis2, divis3) in real time and 0.97 and 0.92 with (10,40,180)=(divis, divis2, divis3) in multi and single worker respectively.
11) speed up depends on the serial time taken to be 36. It can range up to 53 seconds.
12) 3rd config (20,40,200)=(divis, divis2, divis3) has real-time of 30 seconds, but it may have Redis problem described above.
13) Sys+User time is 16 second and 32 seconds in (10,40,180)=(divis, divis2, divis3) and (20,40,180)=(divis, divis2, divis3) respectively