COL733 | Cloud Computing Technology Fundamentals
LAB1 | Batch Processing

Dipen Kumar
2018CS50098

Readings (time is measured in seconds)

Serial Program:

Run1	32.231
Run2	27.894
Run3	30.147
Run4	38.452
Run5	28.547
Average Run Time	31.4542

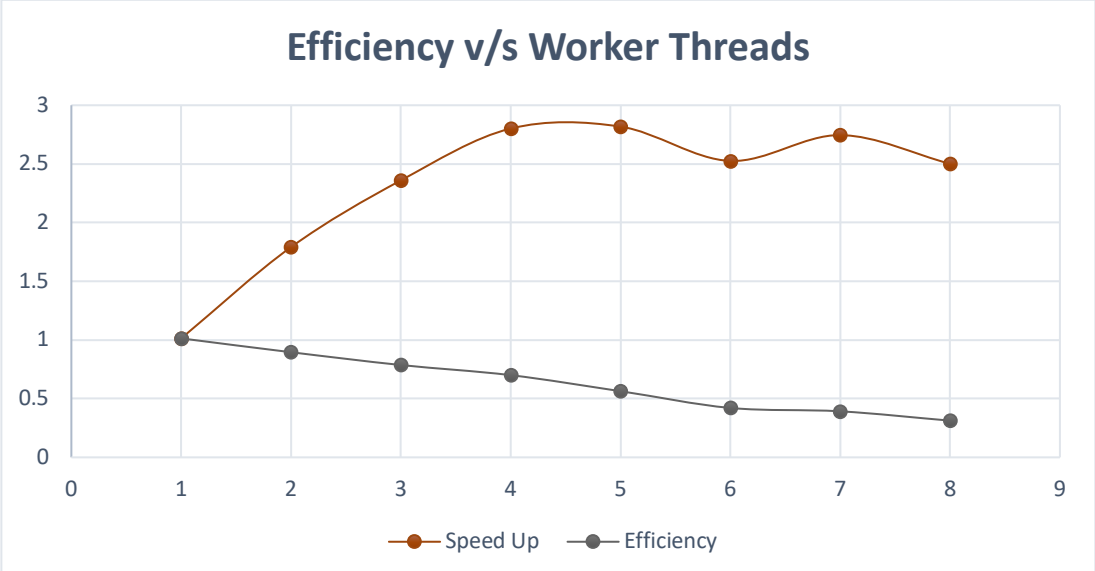
Parallel Program: (size is number of processed in dataset)

Threads	Size	Run1	Run2	Run3	Run4	Run5	Avg Run	Speed Up	Efficiency
1	3000	31.572	31.11	30.827	30.289	31.348	31.0292	1.01369678	1.01369678
2	3000	17.34	17.301	17.598	18.096	17.457	17.5584	1.79140468	0.89570234
3	3000	13.9	13.856	12.516	13.43	12.937	13.3278	2.36004442	0.78668147
4	3000	12.377	11.256	10.934	10.952	10.618	11.2274	2.80155691	0.70038923
5	3000	11.127	10.704	11.526	11.444	11.01	11.1622	2.8179212	0.56358424
6	3000	11.842	12.647	12.737	13.271	11.773	12.454	2.52563032	0.42093839
7	3000	12.384	10.56	10.951	10.859	12.527	11.4562	2.745605	0.39222929
8	3000	13.312	11.32	13.365	13.097	11.779	12.5746	2.5014076	0.31267595

1. What is the best speedup achieved over serial.py?

From the readings, the best speedup achieved was 2.82 with 5 workers (threads).
Theoretically, the speedup should increase with increase in workers and then saturates and may be increase due to increase in overhead from synchronization and communication between workers.
Due to large machine errors, above data don't show increase of speedup with increase in workers.
Runtime of the job is in order of few seconds and so is the machine error due to multiple processes and dynamic load.

2. 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.

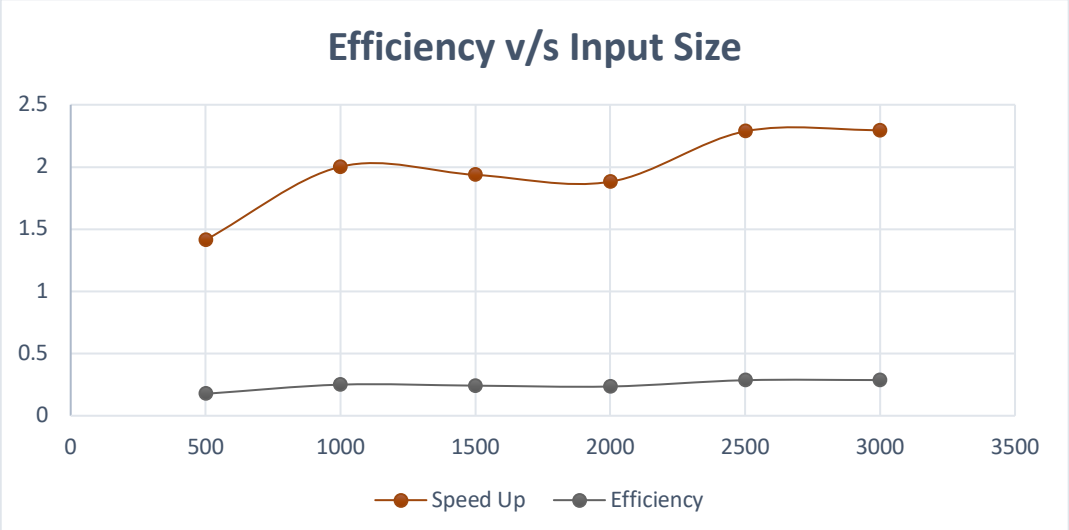


Efficiency decreases with increase in worker threads for fixed input size.

My application can be dived in two parts- parallel part and serial part.
The work is dived among workers and are being parallelly processed. At end I have some results from previous parallel execution which is then serially merged to form final result.
Roughly speaking, size of the work can be approximated with number of files. These files are bundled to form a job, in my case 100 files. Each job is independent and processed by any idle work at given time parallelly with other jobs being processed by other worker.
In serial scenario execution time would be of order number of files.
Here in parallel execution, running time = (time serial) / (# of workers) + merge time
merge time = O(|jobs|) = O(number of files) very toughly O(time serial)
let merge time = f x (time serial) where f is very small
Now, calculating speedup = Ts/Tp = Ts/(Ts/p + fTs) = p/(1+fp)
Hence, we can say speedup should increase with increase till it is not giving high overhead and is still parallelable.
Finally, efficiency = speedup/p = 1/(1+fp) which clearly says efficiency should decrease with increase in workers.
Another way to see the same thing is that during serial part of program other than master thread all workers are idle and which is constant time for given fixed input but with increase in worker parallel part time is reduced that means workers are working for lesser time now while resting for the same time which decreases the efficiency.

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

Threads	Size	Run1	Run2	Run3	Run4	Run5	Avg Run	Speed Up	Efficiency
1	500	4.566	4.902	4.72	4.964	4.621	4.7546	1	1
8	500	3.456	3.245	3.618	3.288	3.207	3.3628	1.41388129	0.17673516
1	1000	9.226	8.979	12.112	9.062	9.778	9.8314	1	1
8	1000	4.99	4.913	5.388	4.592	4.673	4.9112	2.00183255	0.25022907
1	1500	13.757	13.766	14.961	13.855	13.824	14.0326	1	1
8	1500	7.57	6.939	6.878	6.769	8.076	7.2464	1.9364926	0.24206158
1	2000	18.92	17.951	18.291	18.393	17.993	18.3096	1	1
8	2000	10.534	10.045	8.658	10.355	9.07	9.7324	1.88130369	0.23516296
1	2500	23.588	23.065	22.164	24.826	24.363	23.6012	1	1
8	2500	10.958	10.091	9.881	10.597	10.086	10.3226	2.28636196	0.28579525
1	3000	28.868	31.85	27.232	26.677	27.662	28.4578	1	1
8	3000	13.452	11.66	13.138	12.575	11.152	12.3954	2.29583555	0.28697944



From above reading we can say efficiency increases with increase in input size for fixed number of worker threads.
Justification to this follows from the justification of previous part. There since we didn't had to comment on input size I approximated it very roughly but let's see here with little more precision.
I said that program is in two parts- parallel and serial parts
Tp = Ts/p + (merging time)
Note, previously I approximated merging time very roughly saying merging time = f x Ts where f is very small. For sure merging time increases with increase in input size but it doesn't increase at the same rate. that is merging time = $O((\text{input size})^p)$ where $p < 1$. In fact it is very small and can be ignore and approximated to be constant for sufficiently large input size.
Now, let's calculate speedup again.
Speedup = Ts/Tp = Ts/ (Ts/p + c) = p/(1+pc/Ts)
we can see with increase in input size implies increase in Ts will increase speedup.
Efficiency = speedup/p = 1/(1+pc/Ts)
Hence efficiency increases too with increase in input size.
Another way to look at the same thing is that with increase in input size parallel part time increases and hence workers are now working for longer time while resting for the same time which implies high efficiency.

4. The designed solution is scalable. Justify.

I would say the solution is scalable. To justify my answer I will calculate its scalability.
In previous part I said merging time = $O((\text{input size})^p)$ where $p < 1$ and doing the maths out we calculated efficiency = 1/(1+pc/Ts). Let's put the value of $c = k'(Ts)^p$
simplifying, efficiency = $1/(1+kp/Ts^{1-p})$
The above efficiency say- on increasing p (processor/worker) we can increase input size (Ts) to keep the efficiency constant. We saw this from above mentioned two graphs that says efficiency decreases with increase in workers and increase with increase in input size.
scalability = $O(n^{1/(1-p)})$ where n is input size and $0 < p < 1$

5. The designed solution is fault-tolerant. Justify

The design is fault tolerant because I have used acks_late = True which says worker will acknowledge the message to master after the task returns. This ways master is keeping track of the tasks whose acknowledgment it don't receive for a certain long period of time and knows there is some fault and hence the master reschedule the task and is taken by another worker and process it and returns and send back the acknowledgment. Even if the pervious worker whose connection we lost due to fault also does the task we don't need to worry and correctness is ensured since my tasks are idempotent. multiple execution of same tasks will not create any inconsistency in my data since task is only reading some files. multiple execution of same tasks will just result in performance wastage and no correction worries.
Hence the designed solution is fault-tolerant.