Part 3 [34 points]

1. **[17 points]** With your scheduling policy, run the entire workflow **3 separate times**. For each run, measure the execution time of each batch job, as well as the latency outputs of memcached, running with a steady client load of 30K QPS. For each batch application, compute the mean and standard deviation of the execution time [1] across three runs. Also, compute the mean and standard deviation of the total time to complete all jobs - the makespan of all jobs. Fill in the table below. Finally, compute the SLO violation ratio for memcached for the three runs; the number of data points with 95th percentile latency > 1ms, as a fraction of the total number of data points. The SLO violation ratio should be calculated during the time from when the first batch-job-container starts running to when the last batch-job-container stops running.

job name	mean time [s]	std [s]
blackscholes	67	1
canneal	189	4.36
dedup	28.33	0.58
ferret	179.33	0.58
freqmine	174	1
radix	23	0
vips	104	0
total time	189	4.36

Answer: For each run, we define $t_{i,j}$ where $i \in \{\text{blackscholes, canneal, dedup, ferret, freqmine, radix, vips}\}$ is the job name and $j \in \{1, 2, 3\}$ the run index. Here are the formulas for the mean time and standard deviation:

$$\text{mean}_i = \frac{t_{i,1} + t_{i,2} + t_{i,3}}{3}$$

$$\text{std}_i = \sqrt{\frac{(t_{i,1} - \text{mean}_i)^2 + (t_{i,2} - \text{mean}_i)^2 + (t_{i,3} - \text{mean}_i)^2}{2}}$$

To evaluate Memcached latency performance, we define:

- **Start time (T_{start}):** The earliest start time among all batch jobs (excluding memcached).
- **End time (T_{end}):** The latest end time among all batch jobs (excluding memcached).
- N_{total}: total number of latency data points recorded during $[T_{start}, T_{end}]$
- $N_{violations}$: number of data points where p95$^{\text{th}}$ latency > 1ms.

The **SLO Violation Ratio** is computed as:

$$\text{SLO Violation Ratio}_{memcached} = \frac{N_{\text{violations}}}{N_{\text{total}}} = 0 \quad (1)$$

[1] Here, you should only consider the runtime, excluding time spans during which the container is paused.

Create 3 bar plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis), with annotations showing when each batch job started and ended, also indicating the machine each of them is running on. Using the augmented version of mcperf, you get two additional columns in the output: `ts_start` and `ts_end`. Use them to determine the width of each bar in the bar plot, while the height should represent the p95 latency. Align the x axis so that $x = 0$ coincides with the starting time of the first container. Use the colors proposed in this template (you can find them in `main.tex`). For example, use the vips color to annotate when vips started and stopped, the blackscholes color to annotate when blackscholes started and stopped etc.

Plots:

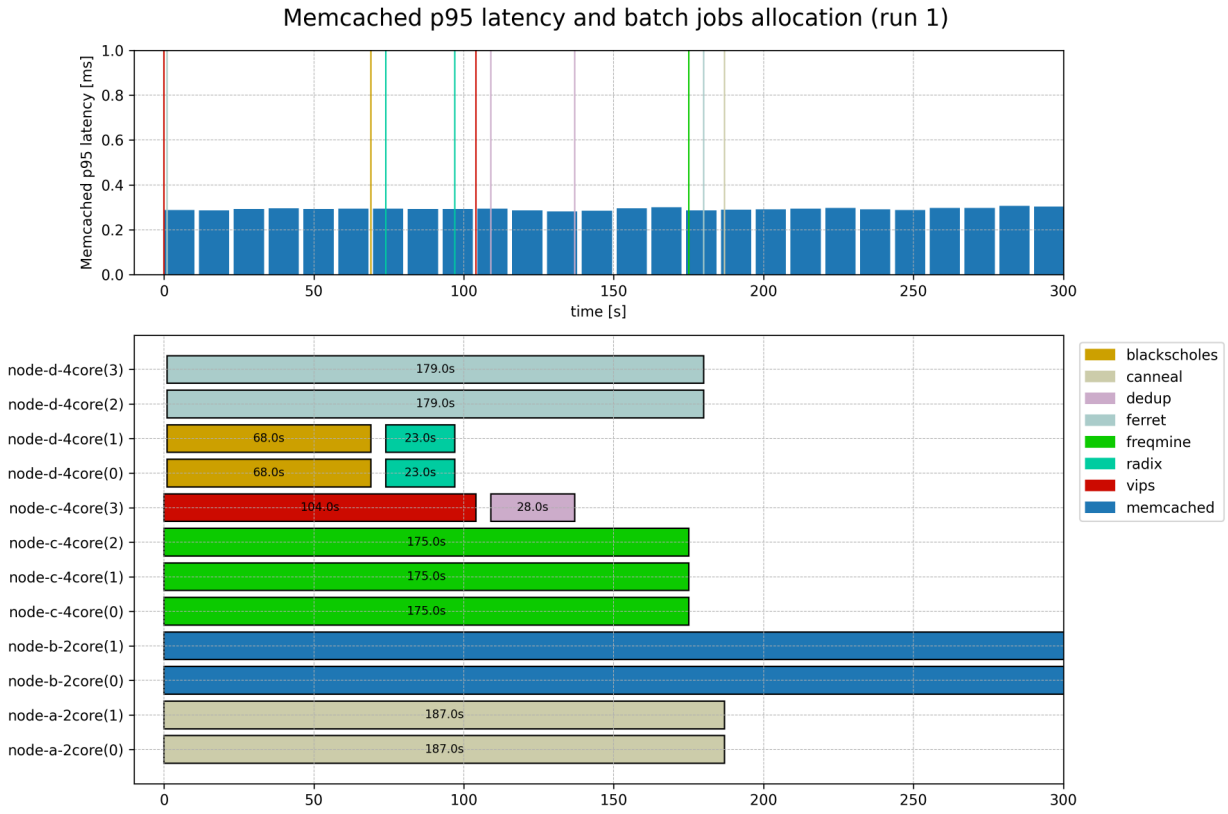

Figure 3: This figure shows the measurements of the first run. **The upper diagram** displays the p95 latency of `memcached` in each measurement interval using the augmented version of `mcperf`. Each bar represents the p95 latency at a specific time slice. The vertical colored lines indicate the exact start and end times of each batch job. **The lower diagram** visualizes the mapping of batch jobs to physical cores across nodes during execution. Each horizontal bar shows where and for how long a specific batch job ran, with the job name represented by both color and bar label. The y-axis labels show the specific cores across nodes (e.g., `node-d-4core(1)`). For example, `freqmine` was scheduled on three cores of `node-c`, running concurrently, while `ferret` occupied two cores on `node-d`. Jobs such as `radix` and `blackscholes` were scheduled sequentially on the same core, one after the other. **Note** that some jobs, such as `freqmine` and `vips`, were executed simultaneously, making their corresponding vertical lines in the upper plot difficult to distinguish due to overlapping.

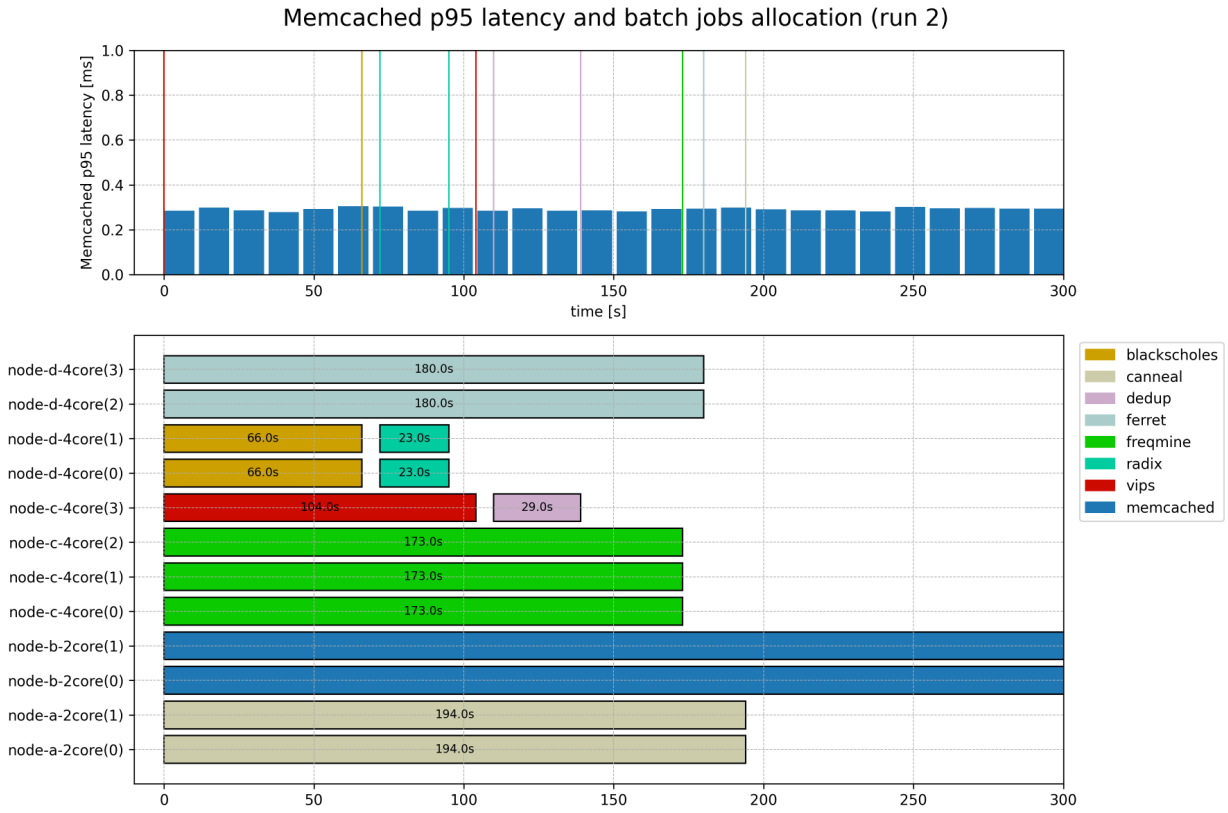

Figure 4: This figure shows the measurements of the second run. **The upper diagram** displays the p95 latency of `memcached` in each measurement interval using the augmented version of `mcperf`. Each bar represents the p95 latency at a specific time slice. The vertical colored lines indicate the exact start and end times of each batch job. **The lower diagram** visualizes the mapping of batch jobs to physical cores across nodes during execution. Each horizontal bar shows where and for how long a specific batch job ran, with the job name represented by both color and bar label. The y-axis labels show the specific cores across nodes (e.g., `node-d-4core(1)`). For example, `freqmine` was scheduled on three cores of `node-c`, running concurrently, while `ferret` occupied two cores on `node-d`. Jobs such as `radix` and `blackscholes` were scheduled sequentially on the same core, one after the other. **Note** that some jobs, such as `freqmine` and `vips`, were executed simultaneously, making their corresponding vertical lines in the upper plot difficult to distinguish due to overlapping.

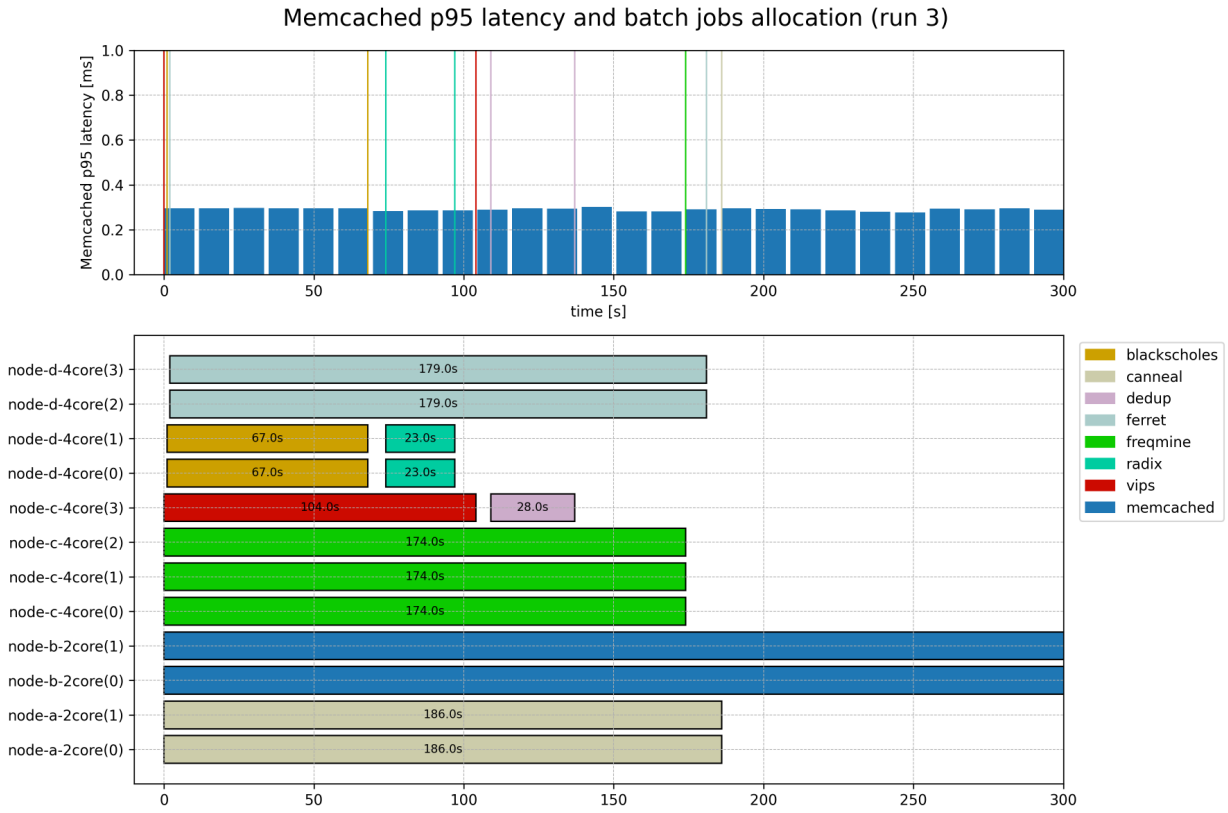

Figure 5: This figure shows the measurements of the third run. **The upper diagram** displays the p95 latency of `memcached` in each measurement interval using the augmented version of `mcperf`. Each bar represents the p95 latency at a specific time slice. The vertical colored lines indicate the exact start and end times of each batch job. **The lower diagram** visualizes the mapping of batch jobs to physical cores across nodes during execution. Each horizontal bar shows where and for how long a specific batch job ran, with the job name represented by both color and bar label. The y-axis labels show the specific cores across nodes (e.g., `node-d-4core(1)`). For example, `freqmine` was scheduled on three cores of `node-c`, running concurrently, while `ferret` occupied two cores on `node-d`. Jobs such as `radix` and `blackscholes` were scheduled sequentially on the same core, one after the other. **Note** that some jobs, such as `freqmine` and `vips`, were executed simultaneously, making their corresponding vertical lines in the upper plot difficult to distinguish due to overlapping.

2. **[17 points]** Describe and justify the "optimal" scheduling policy you have designed.

   - Which node does memcached run on? Why?

     Answer: memcached runs on the node-b. The reason is that even though it doesn't provide the best cpu resources nor a lot of memory, we concluded from Task 1 that it is largely capable of meeting the SLO requirements unless there is a high CPU or l1i interference, which we avoid. This enables us to keep the other better performing cores and higher memory nodes for other more resource hungry tasks.

   - Which node does each of the 7 batch jobs run on? Why?

Answer:

- blackscholes: Blacksholes runs on node-d. Since it neither suffers from nor causes significant interference, it is ideal for collocating with other jobs. We considered both node-c and node-d for its deployment, and chose node-d because Blacksholes isn't CPU-intensive and wouldn't benefit from the more powerful CPU available in node-c.
- canneal: canneal runs on node-a. canneal is both memory and cache dependent, but also sensitive to l1i interference. So, we decided to run it on it's own on a high-memory node, using the fact that it scales decently with parallelism. It doesn't need the best performing CPU, so node-a was the right fit.
- dedup: dedup runs on node-c. We mainly put it here because it's a short running job that only suffers from sharing a core. So we could just run it after vips finishes.
- ferret: ferret runs on node-d. ferret is a long running job that benifits a lot from parallelism (until 4 threads), and also is at the same time CPU and memory hungry, so node-d presented the perfect balance for all of this.
- freqmine: freqmine runs on node-c. freqmine is a long running job that benifits from parallelism, and so we put it on 3 cores. Also it is CPU hungry, and node-c provides the best performing CPUs.
- radix: radix runs on node-d. radix is a small job, that doesn't require much resources and doesn't interfere with other jobs, so we could just run it after blackscholes finishes running.
- vips: vips runs on node-c. vips needs a better CPU, and it experimentally did fine when running on the same node as freqmine, in the sense that the slowdown wasn't that important.

- Which jobs run concurrently / are colocated? Why?

  Answer: ferret runs colocated with blackscholes and then afterwards with radix. ferret is the most sensitive job from all the proposed ones, and so we made sure to run it with the 2 least invasive jobs that are blacksholes and radix. On the other hand, freqmine runs colocated with vips and then dedup, the reason is that even though they might contend for llc cache, vips and dedup are relatively small jobs, and they are the most cpu hungry jobs so we wanted them to run on the best CPU cores of node-c. In other configurations we tried, we saw that canneal, which currently constitutes our bottlneck, could be run much faster (2x) by using the 4 cores on node-c, but we get a comparable makespan at the end because of a new bottleneck.

- In which order did you run the 7 batch jobs? Why?

  Order (to be sorted): [blackscholes, ferret, freqmine, vips, canneal] ; [dedup, radix]

  Why: We run all the longer jobs from the beginning, and as dedup and radix are comparatively very short, we run them just after the medium running jobs blacksholes and vips. We also explored running radix and dedup from the beginning in colocation with memcache, but it failed due to OOM, and also the node-d wasn't the bottleneck anymore after some changes so radix run just fine in colocation.

- How many threads have you used for each of the 7 batch jobs? Why?

Answer: Experiments showed us that using more threads than the number of allocated cores for a job doesn't result in any performance gain, perhaps because of the overhead of context switching. Hence, we allocate each time the same number of threads as the number of allocated cores.

- blackscholes: 2
- canneal: 2
- dedup: 1
- ferret: 2
- freqmine: 3
- radix: 2
- vips: 1

- Which files did you modify or add and in what way? Which Kubernetes features did you use?

  Answer: We make use of a script that adapts each time it is run all the YAML files according to the new allocation we specified. More specifically, we populate in our script, for each job, the node we want it to be run on, the cores it should run on using taskset, and the number of threads. We didn't make use of any additional Kubernetes features.

- Describe the design choices, ideas and trade-offs you took into account while creating your scheduler (if not already mentioned above). Describe how your policy (and its performance) compares to another policy that you experimented with (in particular to the second-best policy that you designed).

  Answer: We allocated the jobs on nodes based on the insights gained from task 1 and task 2, and then we experimented with and without concurrency for some setups. More Specifically we saw in the beginning that Radix was a rather low-weight job and decided to run it in colocation with memcached but this has generated an OOM error so we transfered it to another node that we saw had some idle period. Then seeing that canneal is very hard to colocate with any other job because of how sensitive it is to most types of interferences, we decided to run it on it's own. Canneal constitutes the botlleneck of our scheduling, so in one of our schedulings we experimented running it on node-c on the 4 cores, which led to a huge speed up, but due to the other changes to the scheduling we had to make, we ended up at roughly the same makespan". We previously already explained how we ended up choosing the allocation for ferret along with blacksholes. As for freqmine, colocating it with vips and dedup led to a slowdown of approximately 25 seconds compared to when we run it alone on 4 cores in our other scheduling policy, and we thought that this is ok, on top of the fact that it's not the botlneck. Compared to our Second best run where we run everything concurrently, there isn't much difference in the makespan. However, what was interesting is seeing that our hypothesis of canneal and ferret running well together concurrently on node-c did actually yield to good results, as they were far from being the bottleneck. Our rationale was that since both jobs are frequently accessing memory, they often pause processing while waiting for data. Concurrency could potentially allow one job to utilize the CPU while the other waits for memory, improving the overall throughput. In a slightly different version of our second best run where we tried to improve it, we tried exploiting 1 of the cores in node-b (as we know from task1 that we only need 1 core to satisfy the SLO for

memcached), and used it to ran vips, but it ended up being the botteleneck and had approximately the same makespan.

Please attach your modified/added YAML files, run scripts, experiment outputs and the report as a zip file. You can find more detailed instructions about the submission in the project description file.

Important: The search space of all possible policies is exponential and you do not have enough credits to run all of them. We do not ask you to find the policy that minimizes the total running time, but rather to design a policy that has a reasonable running time, does not violate the SLO, and takes into account the characteristics of the first two parts of the project.