Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cloud Computing Architecture

Semester project report

Group 11
Benigna Bruggmann - 18-928-200
Alexander Joossens - 21-911-367
Yves Kempter - 16-928-251

Systems Group
Department of Computer Science
ETH Zurich
May 25, 2022

Instructions

- Please do not modify the template, except for putting your solutions, group number, names and legi-NR.

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 PARSEC job, as well as the latency outputs of memcached running with a steady client load of 30K QPS. For each PARSEC application, compute the mean and standard deviation of the execution time across three runs. Also compute the mean and standard deviation of the total time to complete all jobs. Fill in the table below. Finally, compute the SLO violation ratio for memcached for the three runs; the number of datapoints with 95th percentile latency $>$ 2ms, as a fraction of the total number of datapoints. Do three plots (one for each run) of memcached p95 latency (y-axis) over time (x-axis) with annotations showing when each parsec job started.

   For Part 3, you must use the following `mcperf` command:

```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_A_IP -a INTERNAL_AGENT_B_IP  \
           --noload -T 6 -C 4 -D 4 -Q 1000 -c 4 -t 20 \
           --scan 30000:30500:10
```

job name	mean time [s]	std [s]
dedup	142	8.49
blackscholes	265	6.48
ferret	294	0.82
freqmine	195	0.82
canneal	229.33	1.70
fft	246	5.89
total time	294	0.82

   The SLO violation ratio for memcached for all three runs is 0, since we run it on node a, while we run all PARSEC jobs on the other nodes b and c. Thus, there is no interference between the jobs and memcached and the p95 latency is constantly around 0.4 to 0.42ms.

2. **[17 points]** Describe and justify the "optimal" scheduling policy you have designed. This is an open question, but you should at minimum answer the following questions:

   - Which node does memcached run on?
   - Which node does each of the 6 PARSEC apps run on?
   - Which jobs run concurrently / are collocated?
   - In which order did you run 6 PARSEC apps?
   - How many threads you used for each of the 6 PARSEC apps?

   Describe how you implemented your scheduling policy. Which files did you modify or add and in what way? Which Kubernetes features did you use? Please attach your modified/added YAML files, run scripts and report as a zip file. **Important: The search space of all the possible policies is exponential and you do not have enough credit 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**

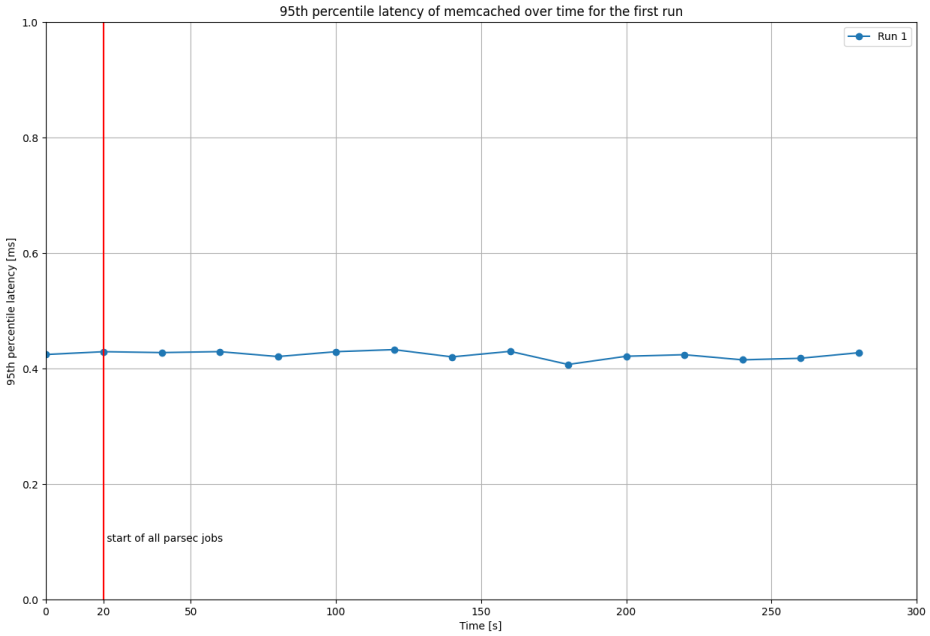

Figure 1: Memcached p95 latency over time for the first run

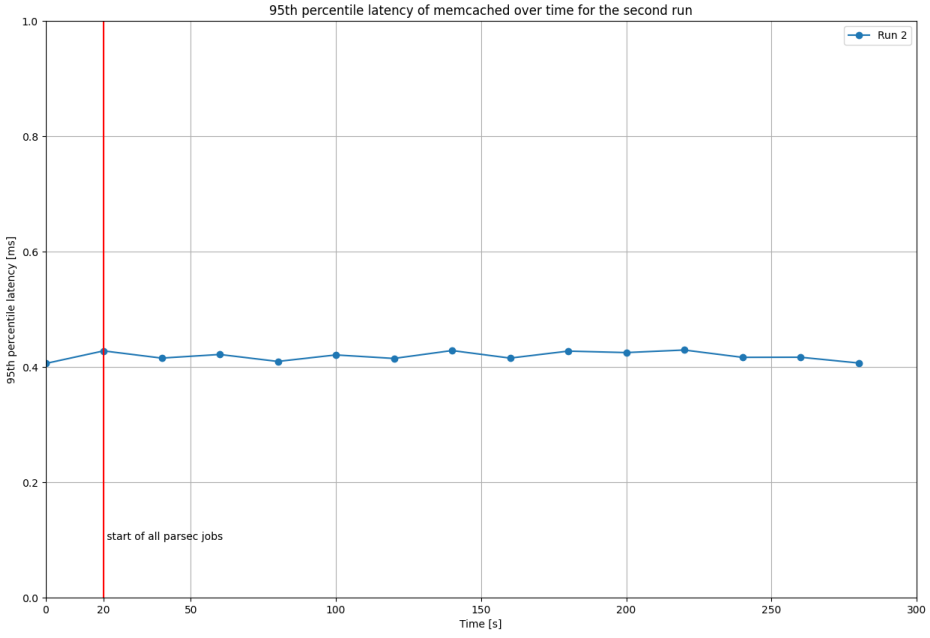

Figure 2: Memcached p95 latency over time for the second run

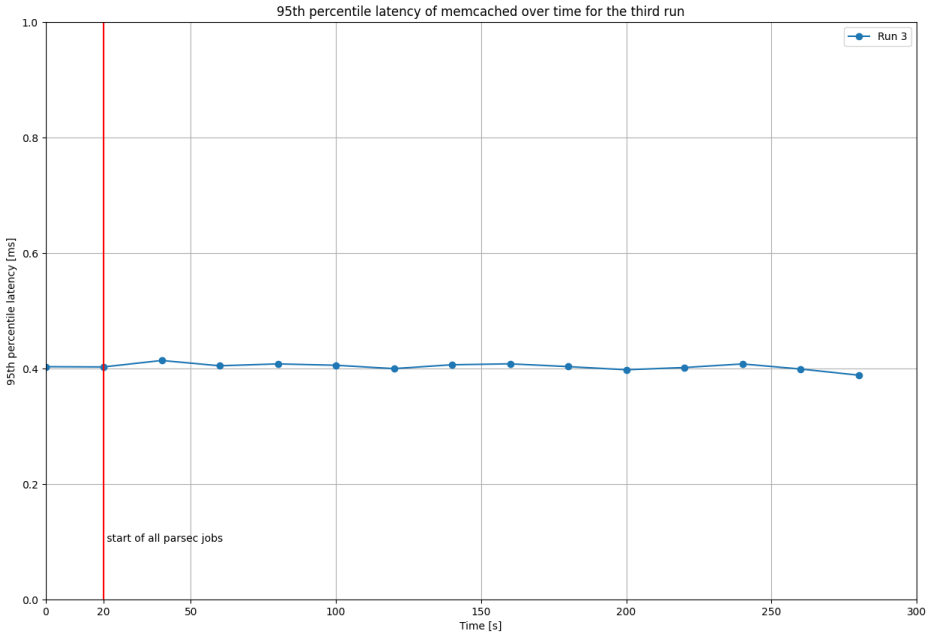

Figure 3: Memcached p95 latency over time for the third run

violate the SLO and takes into account the characteristics of the first two parts of the project.

Memchached was run on node node-a-2core of nodetype n2d-highcpu-2. This node is especially well suited to memcached's requirements, since we know from Part 1 that memchached is heavy on CPU usage. Additionally we know from Part 1 that memchached achieves the required performance on a two core machine. Memchached was run using one thread.

We choose not to run any PARSEC apps on node node-a-2core. This avoids latency problems with the memcached application since memchached has a machine for itself without any interference. On node node-b-4core we only ran the PARSEC app ferret. Part 2 showed that ferret requires more memory than other PARSEC jobs, therefore the machine type n2d-highmem-4 with especially much memory resources is a suitable choice. The other 5 PARSEC apps are all run on the node node-c-8core.

All PARSEC apps were started in consecutive order without any delay in between them using a bash script with individual kubectl commands. The exact order can be seen in the script provided. Using this method, all pods are created within approximately 6 seconds. This means ferret runs alone on node-b-4core, while all other jobs start out collocated on node-c-8core and finish gradually. The first PARSEC job to finish is dedup after on average 142 seconds, followed by freqmine after on average 195 seconds, then canneal terminates after on average 229 seconds, fft after on average 246 seconds, and lastly blackscholes after on average 265 seconds. The last job to finish is ferret on node-b-4core after on average 294 seconds. This is probably due to ferret being the heaviest job (when we look at the resource usage table from Part 2) so it takes the most time.

Our scheduling policy relies on resource requests in order to guaranty that PARSEC apps receive enough resources to finish in reasonable time. Jobs that have shown in preliminary tests to take more resources request these explicitly. Additionally they are given more threads in order to dynamically take over freeing up cores via the machines scheduling policy. Ferret was run with 4 threads, since it occupies a machine with 4 cores alone, i.e. without any collocated job. This is the number of threads that is expected to achieve maximal speed on a n2d-highmem-4 machine. Any additional threads would lead to overhead of context switching and would increase the execution time of ferret. Freqmine was run with 4 threads, however we used the request resources function of kubectl to make sure that at any time it gets at least 2 cores. Canneal was run with 3 threads and a resource request of 2 cores (This is a request, so the scheduler could give it even more CPU cores if it is possible). This was done since freqmine and canneal proved to be the longer running jobs during preliminary tests. Giving them less requested resources than threads gives the other jobs space to execute in parallel and once they finish, the scheduling machine will assign the additional threads of canneal and freqmine to the freed cores. Both fft and dedup were run with one thread and without any resource request.

To implement our policy we used the nodeselector field in the yaml files to specify on which node a process should run. Additionally, we specified in the yaml files how many threads each process should have and used resource requests in the yaml files to make sure certain jobs get reliable resources. Finally, we changed the used dataset by specifying the native dataset.

Part 4 [76 points]

1. **[18 points]**

   For this question, use the following `mcperf` command to vary QPS from 5K to 120K:

   ```
   $ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
   $ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP  \
             --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 \
             --scan 5000:120000:5000
   ```

   a) How does memcached performance vary with the number of threads (T) and number of cores (C) allocated to the job? In a single graph, plot the 95th percentile latency (y-axis) vs. QPS (x-axis) of memcached (running alone, with no other jobs collocated on the server) for the following configurations (one line each):

   - Memcached with T=1 thread, C=1 core
   - Memcached with T=1 thread, C=2 cores
   - Memcached with T=2 threads, C=1 core
   - Memcached with T=2 threads, C=2 cores

   Label the axes in your plot. State how many runs you averaged across (we recommend three runs) and include error bars. The readability of your plot will be part of your grade.

   From the plot we can see that with 1 thread and 1 core, we can reach a QPS rate of up to about 110K. The 95th percentile latency stays around 0.75ms until 90K QPS. Afterwards, it increases up to 1ms. The reason for the QPS limit is the limited service rate of the thread. The thread can serve a certain number of queries per second, and as soon as this limit is reached, the QPS rate will not increase anymore but instead the latency will start to increase.

   If we have 1 thread and 2 cores, we reach about the same performance as with 1 thread and 1 core. The reason is that we still only have one thread and this cannot run on multiple cores. Thus, it will result in one thread running on one of the two available cores.

   If we have 2 threads on 1 core, the 95th percentile latency is a little higher than with only 1 thread, until about 90K QPS, after which point it will start increasing exponentially to nearly 2ms. The reason is that with 2 threads on one core, the scheduling is more involved and thus adds an overhead to the latency. This is especially high if we reach the maximum possible QPS, i.e. if the core is under full use.

   Finally, for 2 threads and 2 cores, we can see that the latency is lower than any other combination. It is about 0.5 to 0.6ms while the QPS rate increases up to nearly 120K. So the overall latency is quite constant for different loads between 5K and 120K QPS. This is due to the two threads which can run on two different cores. Thus, the two threads do not interfere on the core and can run optimally until their saturation point which is not yet reached at 120K QPS.

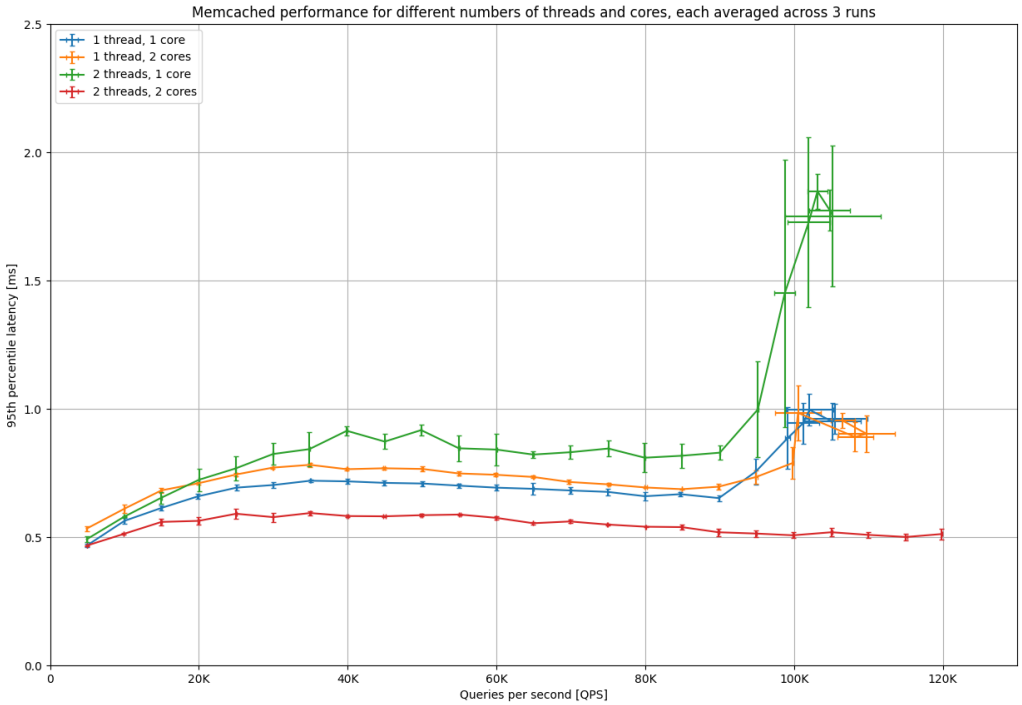

Figure 4: Memcached performance for different numbers of threads and cores, each averaged across 3 runs

b) To support the highest load in the trace (120K QPS) without violating the 1.5ms latency SLO, how many memcached threads (T) and CPU cores (C) will you need? i.e., what value of T and C would you select?

As we can see in the plot from a), if we use 2 threads and 2 cores, we could reach the 120K QPS with about 0.5ms 95th percentile latency. Thus, we would not violate the SLO. In our measurements, we did not reach exactly 120K QPS, but about 119K, and if we would increase the QPS load, we would reach the goal.

c) Assume you can change the number of cores allocated to memcached dynamically as the QPS varies from 5K to 120K, but the number of threads is fixed when you launch the memcached job. How many memcached threads (T) do you propose to use to guarantee the 1.5ms 95th percentile latency SLO while the load varies between 5K to 120K QPS? i.e., what values of T would you select?

As we have seen in b) and in the plot, we need at least 2 threads to reach the 120K QPS. We will satisfy the SLO with 2 threads if we have 2 cores or more. If we have more cores, this will not have any impact on the latency, as the rest of the cores would run idle. If we have only one core, we can see in the plot that we will not satisfy the SLO. If we use more threads, we will violate the latency SLO even more than with 2 threads. Thus, the best value for T is

2.
d) Run memcached with the number of threads T that you proposed in c) above and measure performance with $C = 1$ and $C = 2$.

Measure the CPU utilization on the memcached server at each 5-second load time step.

Plot the performance of memcached using 1-core ($C = 1$) and using 2 cores ($C = 2$) in **two separate graphs**, for $C = 1$ and $C = 2$, respectively. In each graph, plot QPS on the x-axis, ranging from 5K to 120K. In each graph, use two y-axes. Plot the 95th percentile latency on the left y-axis. Draw a dotted horizontal line at the 1.5ms latency SLO. Plot the CPU utilization (ranging from 0% to 100% for $C = 1$ or 200% for $C = 2$) on the right y-axis. For simplicity, we do not require error bars for these plots.

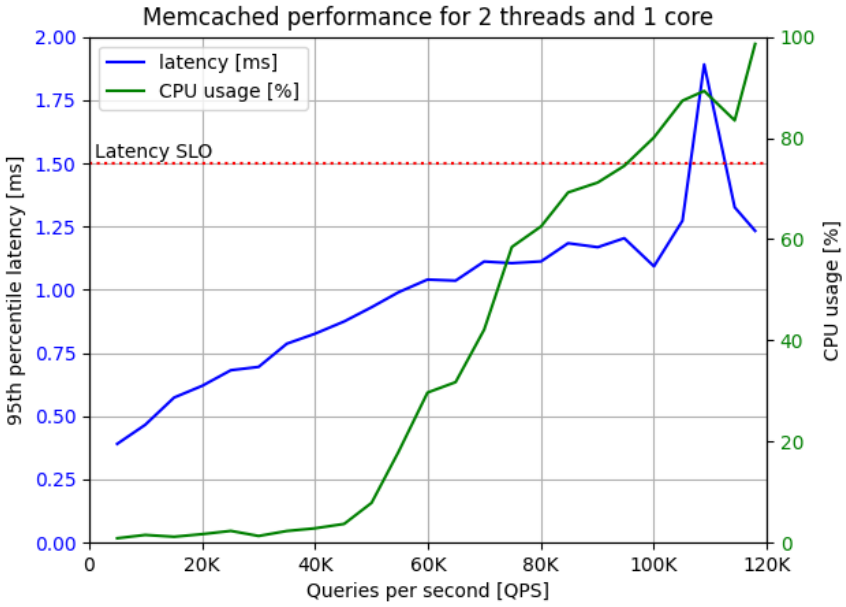

Figure 5: Memcached performance for 2 threads and 1 core

As we can see from the plots, for two threads on one core, we do not satisfy the SLO for a QPS load of about 100K and higher. This is in correlation with the CPU usage which increases steadily with the load and is especially high at 100K QPS and above. It even reaches 100% CPU usage, at which point not all queries can be served in the requested time and thus the SLO gets violated.

For two threads on two cores, the SLO is not violated. The latency is constantly around 0.5ms. Nevertheless, the CPU usage starts to increase at around 45K QPS. We can see that we reach a CPU usage of around 140% at the highest point. Thus, the cores are not under full load and can therefore serve the request in a reasonable time, satisfying the SLO. Moreover we can see that the CPU usage is over 100% and hence, if we would use only one core, we would reach full CPU usage and violate the SLO with high probability. This confirms what we have seen in the first plot.

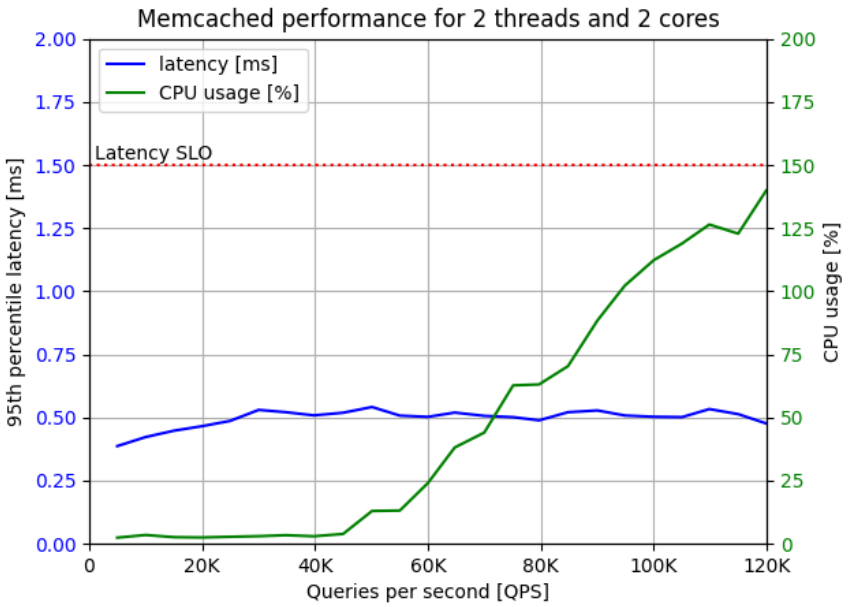

Figure 6: Memcached performance for 2 threads and 2 cores

2. **[15 points]** You are now given a dynamic load trace for memcached, which varies QPS randomly between 5K and 100K in 10 second time intervals. Use the following command to run this trace:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
          --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
          --qps_interval 10 --qps_min 5000 --qps_max 100000
```

Note that you can also specify a random seed in this command using the `--qps_seed` flag.

Design and implement a controller to schedule memcached and the PARSEC benchmarks on the 4-core VM. The goal of your scheduling policy is to successfully complete all PARSEC jobs as soon as possible without violating the 1.5ms 95th percentile latency for memcached. **Your controller should not assume prior knowledge of the dynamic load trace. You should design your policy to work well regardless of the random seed.** Also make sure to check that all the PARSEC jobs complete successfully and do not crash. Note that PARSEC jobs may fail if given insufficient resources.

Describe how you designed and implemented your scheduling policy. Include the source code of your controller in the zip file you submit. To describe your scheduling policy, you should at minimum answer the following questions. For each, also **explain why**:

- How do you decide how many cores to dynamically assign to memcached?
- How do you decide how many cores to assign each PARSEC job?
- How many threads do you use for each of the PARSEC apps?

- Which jobs run concurrently / are collocated and on which cores?
- In which order did you run the PARSEC apps?
- How does your policy differ from the policy in Part 3?
- How did you implement your policy? e.g., docker cpu-set updates, taskset updates for memcached, pausing/unpausing containers, etc.

Memchached is run using two threads and pinned to core 0 using the `taskset` command at the start of the execution. In order to determine whether or not memchached needs to run on more than one core, we observe the CPU usage on core 0. If the usage is above 60%, we change memchached's CPU affinity to CPU 0 and 1 using the `taskset` command. Additionally, we reduce the CPU shares of the PARSEC job currently running on CPU 1 by half to 512 using the `docker update --cpu-shares` command to run the job and memcached concurrently on one core. Furthermore, if the CPU usage of CPU 0 increases to over 90%, the PARSEC job collocated with memchached on CPU 1 is paused using the `docker pause` command, to give more resources to memcached. The job is unpaused using `docker unpause` when the CPU usage on core 0 and 1 combined falls below 75%. If the CPU usage of CPU 0 increases to over 90% at a time where all `lowprio_tasks` have finished, we instead change the CPU affinity of the `highprio_task` to cores 2 and 3 instead of pausing. To reverse this once CPU usage has fallen low enough again we set the `highprio_task` CPU affinity to cores 1, 2 and 3 again. Memchached's CPU affinity is reset to CPU 0 in case the CPU usage of core 0 falls below 30 percent. We choose such a low value to prevent the controller from switching back and forth between the two states.

We put each PARSEC job to one of two groups. Group one contains fft, dedup and canneal we call these `lowprio_tasks`. They are started using `docker run --cpuset-cpus` with one thread and pinned to CPU 1, since they take much less time to execute than the others, which we learned from part 3. The other group is called the `highprio_tasks` and consists of ferret (started with 3 threads), freqmine (started with 3 threads) and blackscholes (started with 2 threads). The order of execution is only determined within the group, `lowprio_tasks` are executed starting with fft, followed by canneal and finished with dedup. The first `highprio_task` that is started is blackscholes, followed by ferret and the last one is freqmine.

Initially, `highprio_tasks` are pinned to core 2 and 3 until all `lowprio_tasks` have been finished. At that point, nothing is collocated on CPU 1 together with memchached and thus the remaining `highprio_tasks` are pinned to cores 1, 2 and 3 using the `taskset` command. This is the reason why ferret and freqmine are started using 3 threads. The `lowprio_tasks` consistently finished several minutes earlier than the `highprio_tasks`. Therefore, the reverse case of `lowprio_tasks` taking up more CPU resources than 1 was neglected.

The biggest difference to part 3 is that this time we not only collocate PARSEC jobs on the same machine as memchached, but we even collocate the PARSEC jobs on the same CPU at time. Additionally, our implementation in part 3 did not use either the docker pause functionality nor did it update container resources while a container was running. All jobs were started at the same time in part 3, whereas now we waited for others to finish to minimize congestion.

We implemented our policy using a mix of the Docker SDK for Python and Docker commands itself, since we ran into some problems if we used solely the Docker SDK for Python. In order to measure the CPU utilization we used the psutil package in python. For this we used `psutil.cpu_percent` function with a 0.1 second measuring interval. This call is embedded at the start of the `while:True` control loop. Since this is a blocking call it blocks the loop execution for 0.1 seconds, therefore our controller logic is executed once ever 0.1 seconds. To get the state of containers we used `docker inspect -f '.State.Status'`. We then iterate through the possible states of each task group and take actions based on the current state. Mainly, once a task has exited, time stamps are given out and a new task of the same group is started if one is left. The starting procedure consists of a predefined function that starts the task using `docker run --cpuset-cpus`. These functions are created and stored in an list at the start of the controller. Additionally, we take into account the current state of the execution to use `docker update` to assign the correct amount of cores to the task. At the end of the control loop we execute the logic implementing our scheduling policy regarding CPU assignment for memchached as well as pausing and unpausing using `docker pause` and `docker unpause` for the relevant PARSEC job.

3. **[23 points]** Run the following `mcperf` memcached dynamic load trace:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
          --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
          --qps_interval 10 --qps_min 5000 --qps_max 100000 \
          --qps_seed 42
```

Measure memcached and PARSEC performance when using your scheduling policy to launch workloads and dynamically adjust container resource allocations. Run this workflow 3 separate times. For each run, measure the execution time of each PARSEC job, as well as the latency outputs of memcached. For each PARSEC application, compute the mean and standard deviation of the execution time across three runs. Also compute the mean and standard deviation of the total time to complete all jobs. Fill in the table below. Also compute the SLO violation ratio for memcached for each of the three runs; the number of datapoints with 95th percentile latency $>$ 1.5ms, as a fraction of the total number of datapoints.

job name	mean time [s]	std [s]
dedup	125.27	6.12
blackscholes	236.92	1.74
ferret	875.89	4.06
freqmine	351.37	2.73
canneal	730.17	4.75
fft	234.48	3.01
total time	1465.02	5.69

Include six plots – two plots for each of the three runs – with the following information. Label the plots as 1A, 1B, 2A, 2B, 3A, and 3B where the number indicates the run and the letter indicates the type of plot (A or B), which we describe below. In all plots, time will be on the x-axis and you should annotate the x-axis to indicate which PARSEC benchmark starts executing at which time. If you pause/unpause any workloads as part of your policy, you should also indicate the timestamps at which jobs are paused and unpaused. All the plots will have have two y-axes. The right y-axis will be QPS. For Plots A, the left y-axis will be the 95th percentile latency. For Plots B, the left y-axis will be the number of CPU cores that your controller allocates to memcached.

For all three runs we achieve a violation rate of 0.0%. We can check this by looking at the plots A. We can see that the latency never reaches the SLO limit and thus we have no violations. In addition, we can see that we quite frequently pause and unpause the `lowprio_tasks`. Nevertheless, they still finish before the `highprio_tasks`. From the plots B we learn that the number of cores assigned to memcached change rather frequently as well. The reason for these frequent changes are the volatile CPU usages for memcached. Thus, our controller tries to adapt to the new measurements by changing the resource assignment accordingly.

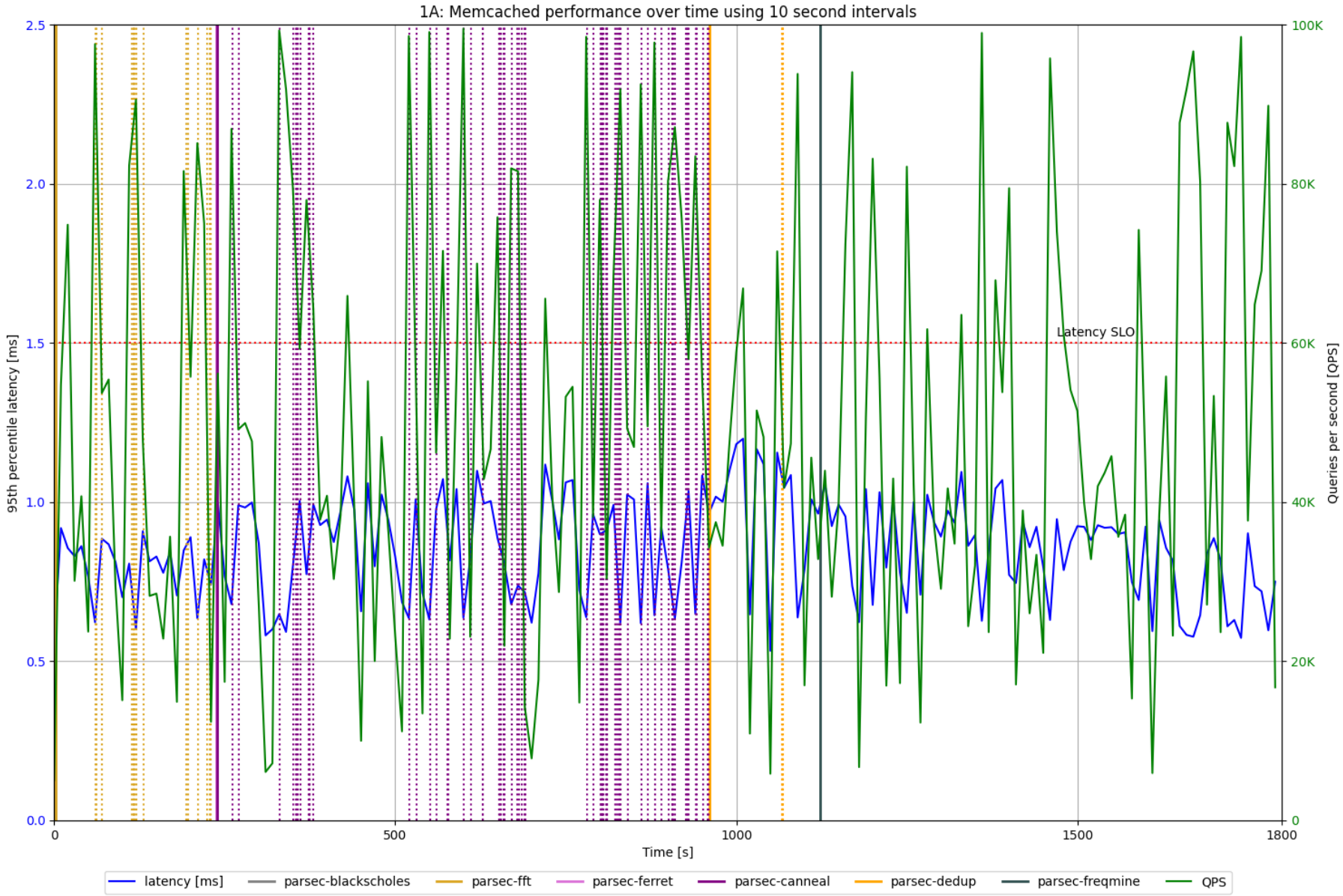

Figure 7: 1A: Memcached performance using 10 second intervals

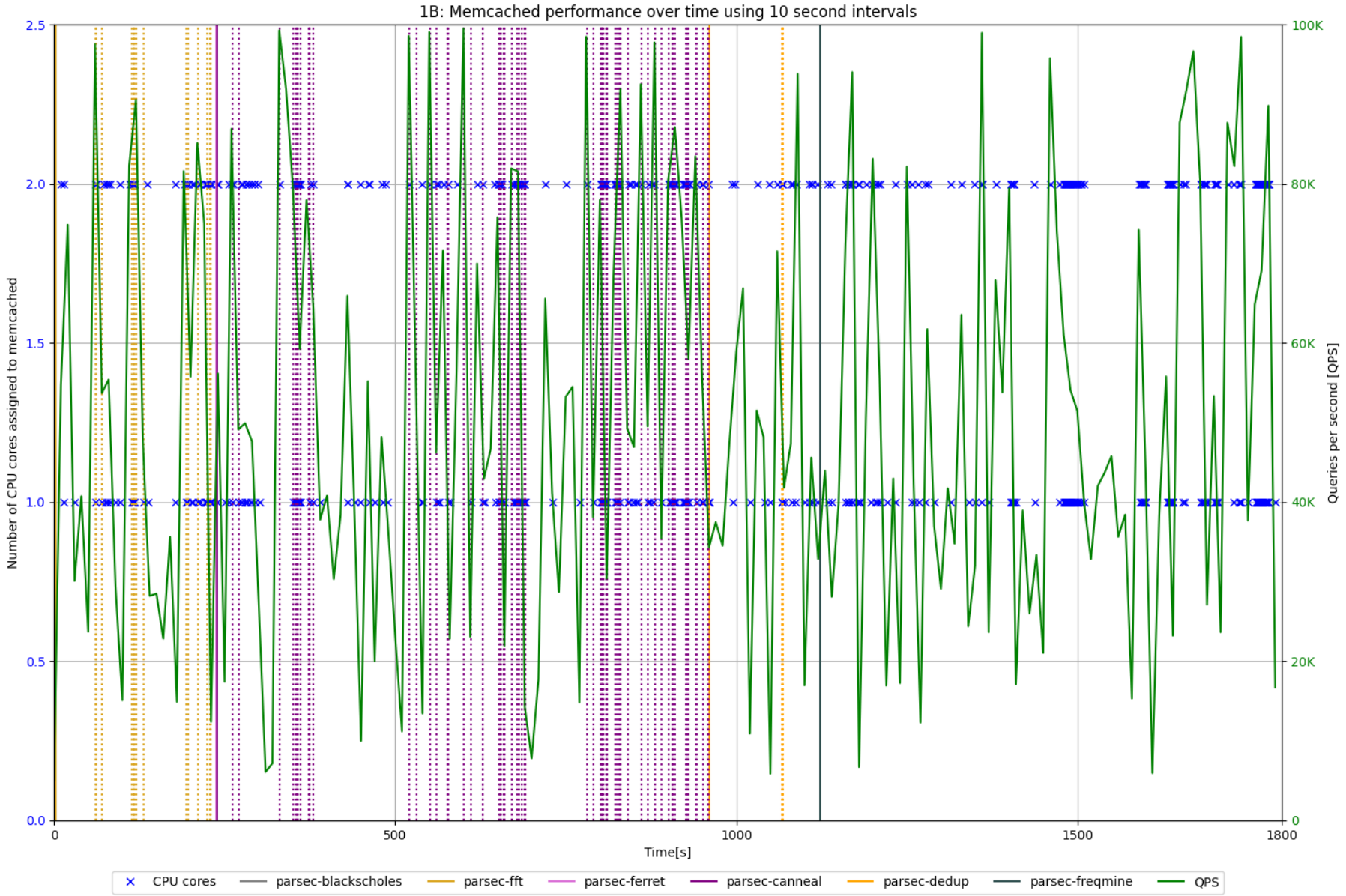

Figure 8: 1B: Memcached performance using 10 second intervals

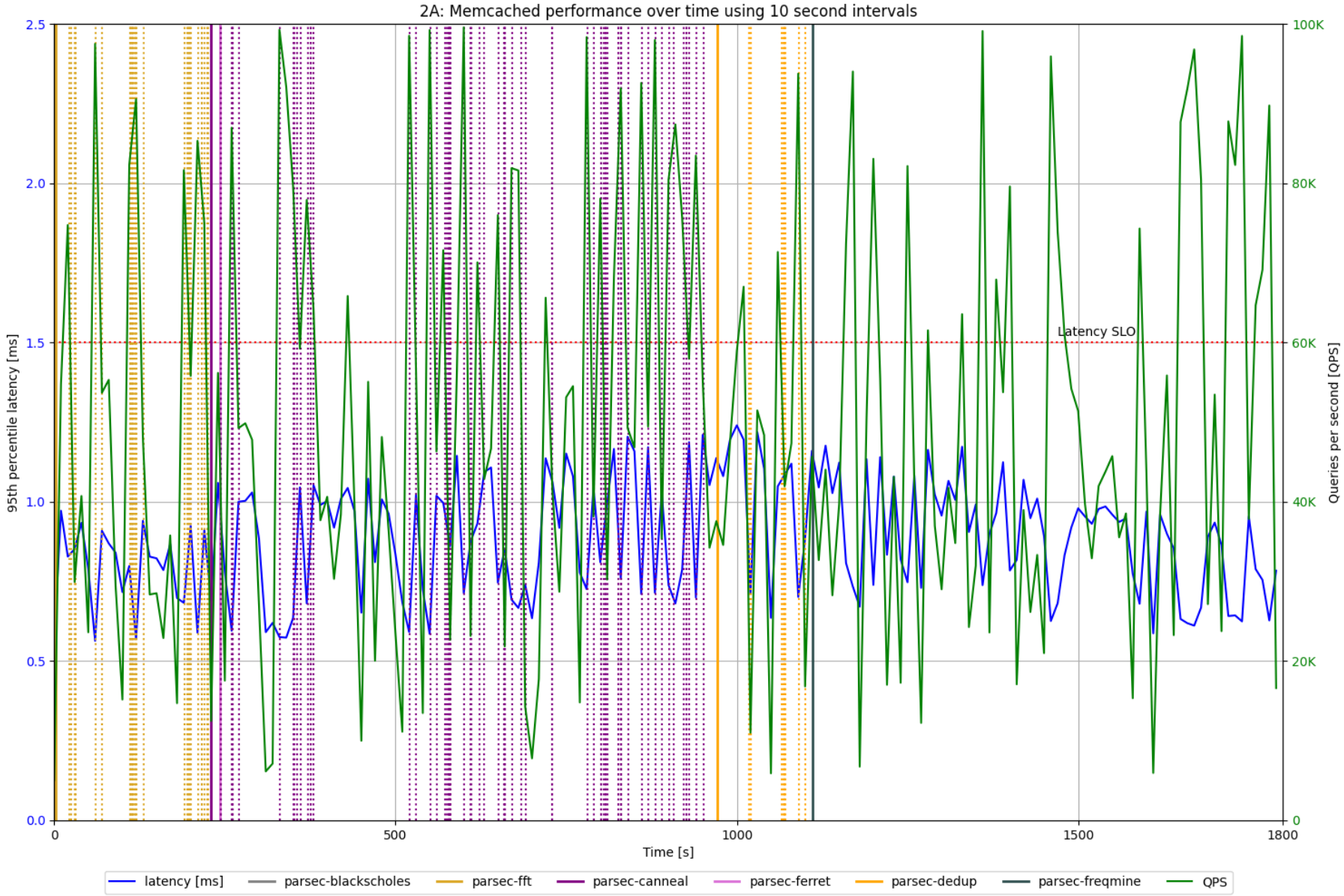

Figure 9: 2A: Memcached performance using 10 second intervals

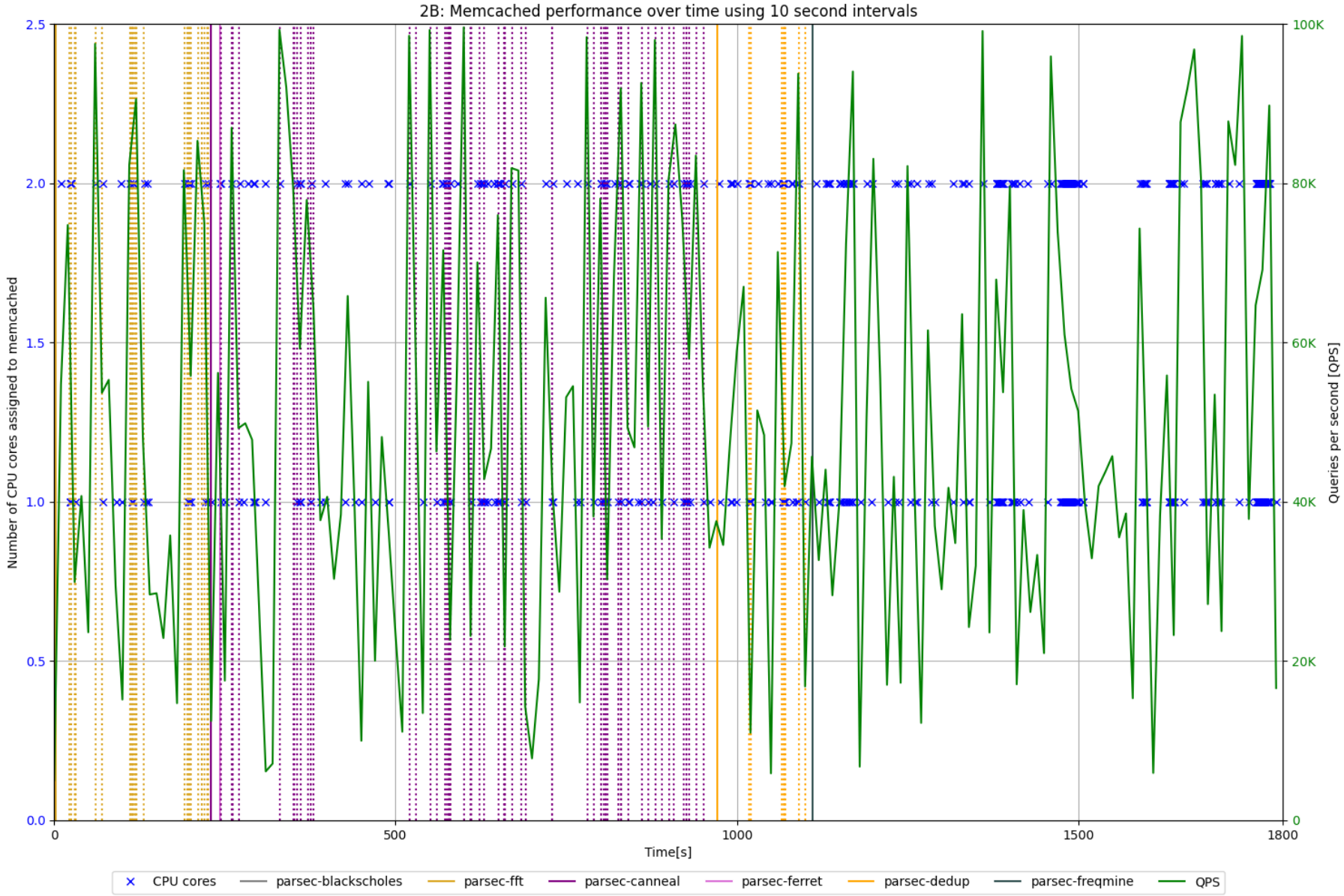

Figure 10: 2B: Memcached performance using 10 second intervals

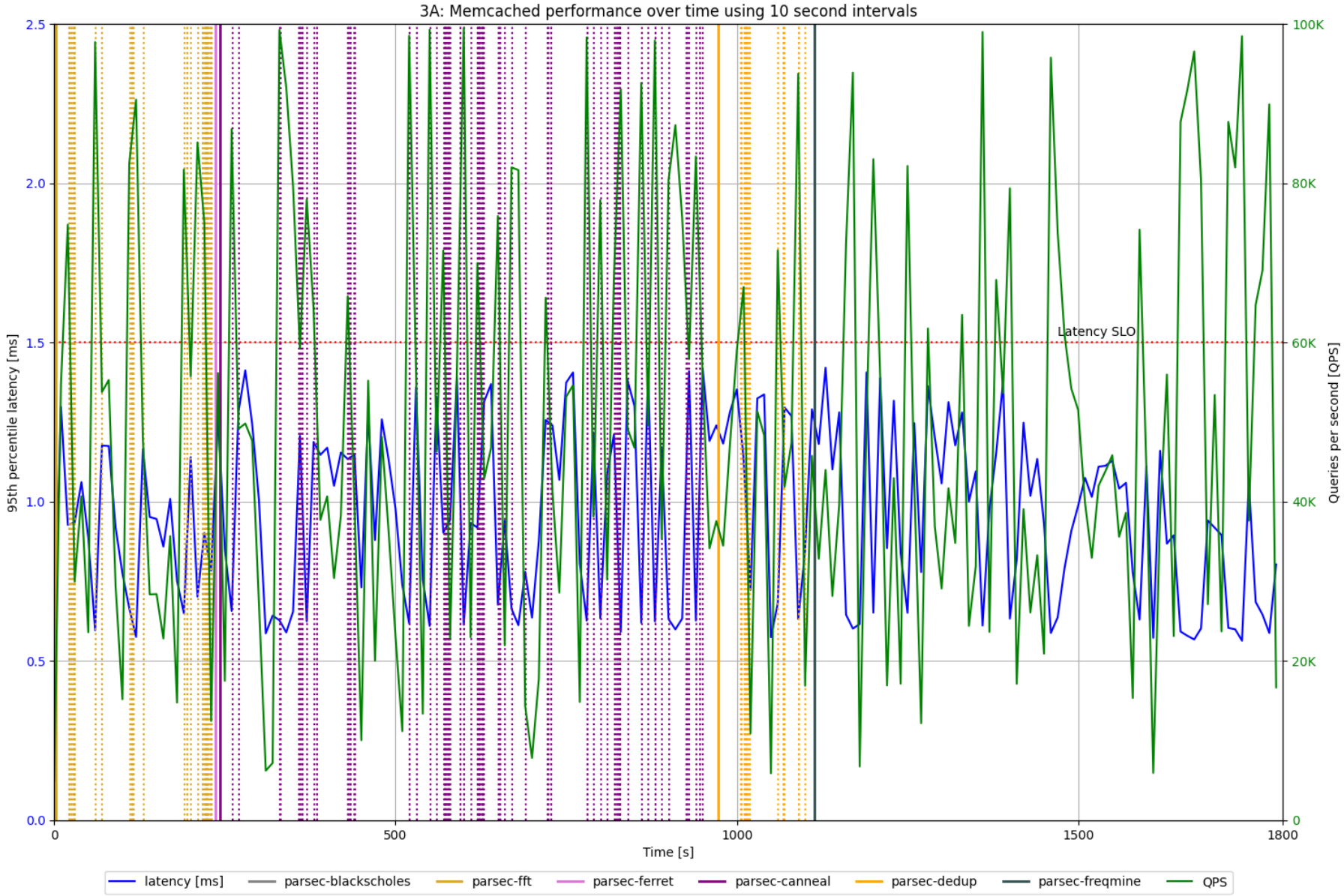

Figure 11: 3A: Memcached performance using 10 second intervals

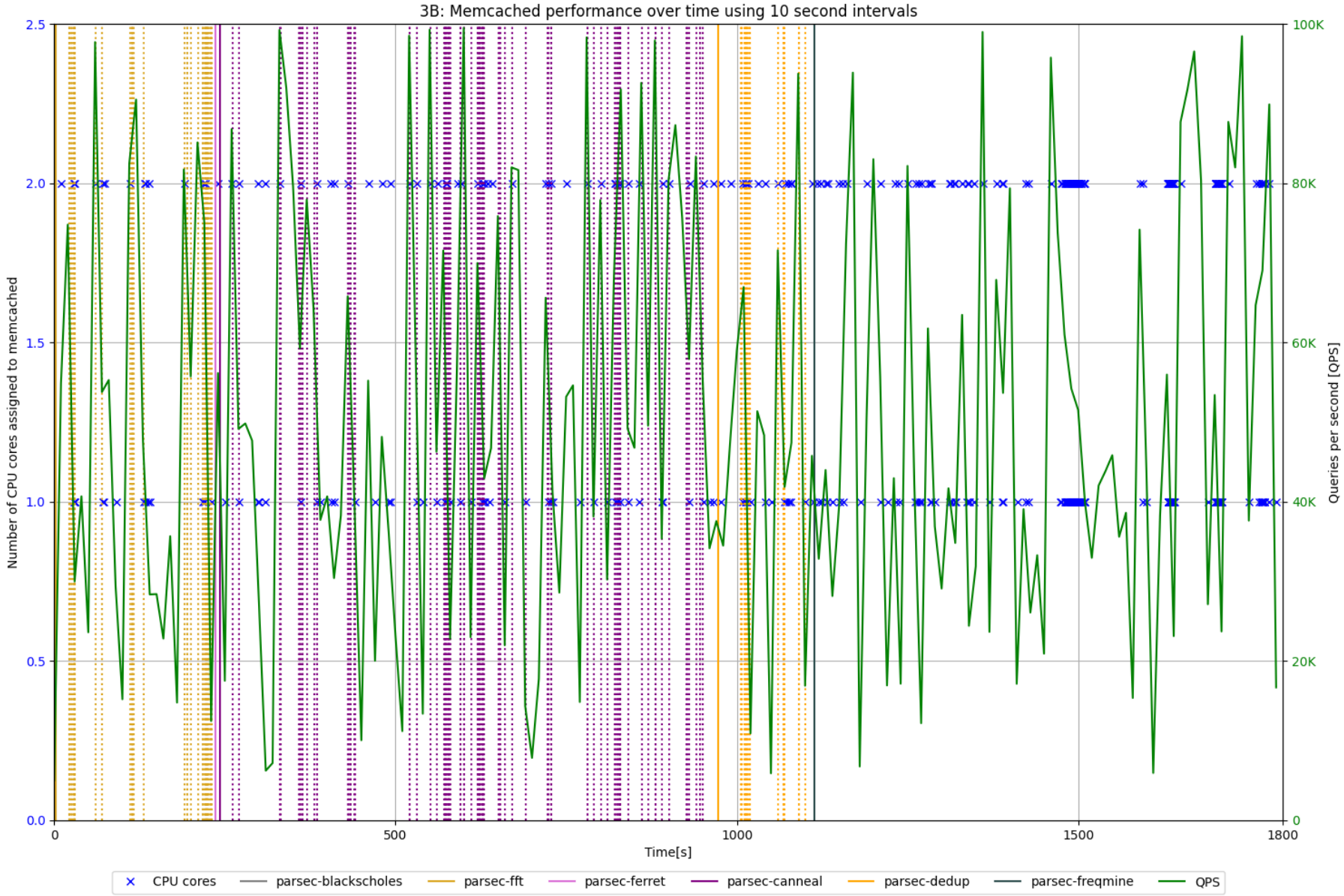

Figure 12: 3B: Memcached performance using 10 second intervals

4. **[20 points]** Repeat Part 4 Question 4 with a modified `mcperf` dynamic load trace with a 5 second time interval (`qps_interval`) instead of 10 second time interval. Use the following command:

```
$ ./mcperf -s INTERNAL_MEMCACHED_IP --loadonly
$ ./mcperf -s INTERNAL_MEMCACHED_IP -a INTERNAL_AGENT_IP \
          --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 1800 \
          --qps_interval 5 --qps_min 5000 --qps_max 100000 \
          --qps_seed 42
```

You do not need to include the plots or table from Question 4 for the 5-second interval. Instead, summarize in 2-3 sentences how your policy performs with the smaller time interval (i.e., higher load variability) compared to the original load trace in Question 4. What is the SLO violation ratio for memcached (i.e., the number of datapoints with 95th percentile latency $>$ 1.5ms, as a fraction of the total number of datapoints) with the 5-second time interval trace?

What is the smallest `qps_interval` you can use in the load trace that allows your controller to respond fast enough to keep the memcached SLO violation ratio under 3%? Use this `qps_interval` in the command above and collect results for three runs. Include the same types of plots (1A, 1B, 2A, 2B, 3A, 3B) and table as in Question 3.

job name	mean time [s]	std [s]
dedup	160.81	1.54
blackscholes	239.61	4.99
ferret	860.19	7.71
freqmine	393.26	6.86
canneal	774.56	13.28
fft	250.33	5.37
total time	1493.89	3.21

For the 5 second interval we still have an SLO violation rate of 0.0%. Nevertheless, the execution got more volatile than with the 10 second interval. This means that we change the number of cores for memcached more often and we pause and unpause containers really frequently.

Via trial and error we find a 2 second interval for the smallest `qps_interval` to allow the controller to respond fast enough to keep the memcached SLO violatoin ratio under 3%. For the first run, we achieve a violation rate of 1.55%. For the second run, we got 1.0% of SLO violations and for the third run, we are at 1.22% violations. The plots are barely readable since the QPS range and thus also everything else changes very frequently. The short interval times combined with the many measurements displayed in the graphs make it hard to see the spikes above 1.5 ms in SLO measurement at the graphs resolution. We pause and unpause even more often as well as we change the CPU cores assigned to memcached more. The frequent changes from the QPS lead to a more volatile reaction of our controller and thus to more overhead. Nevertheless, we still got only a small violation rate.

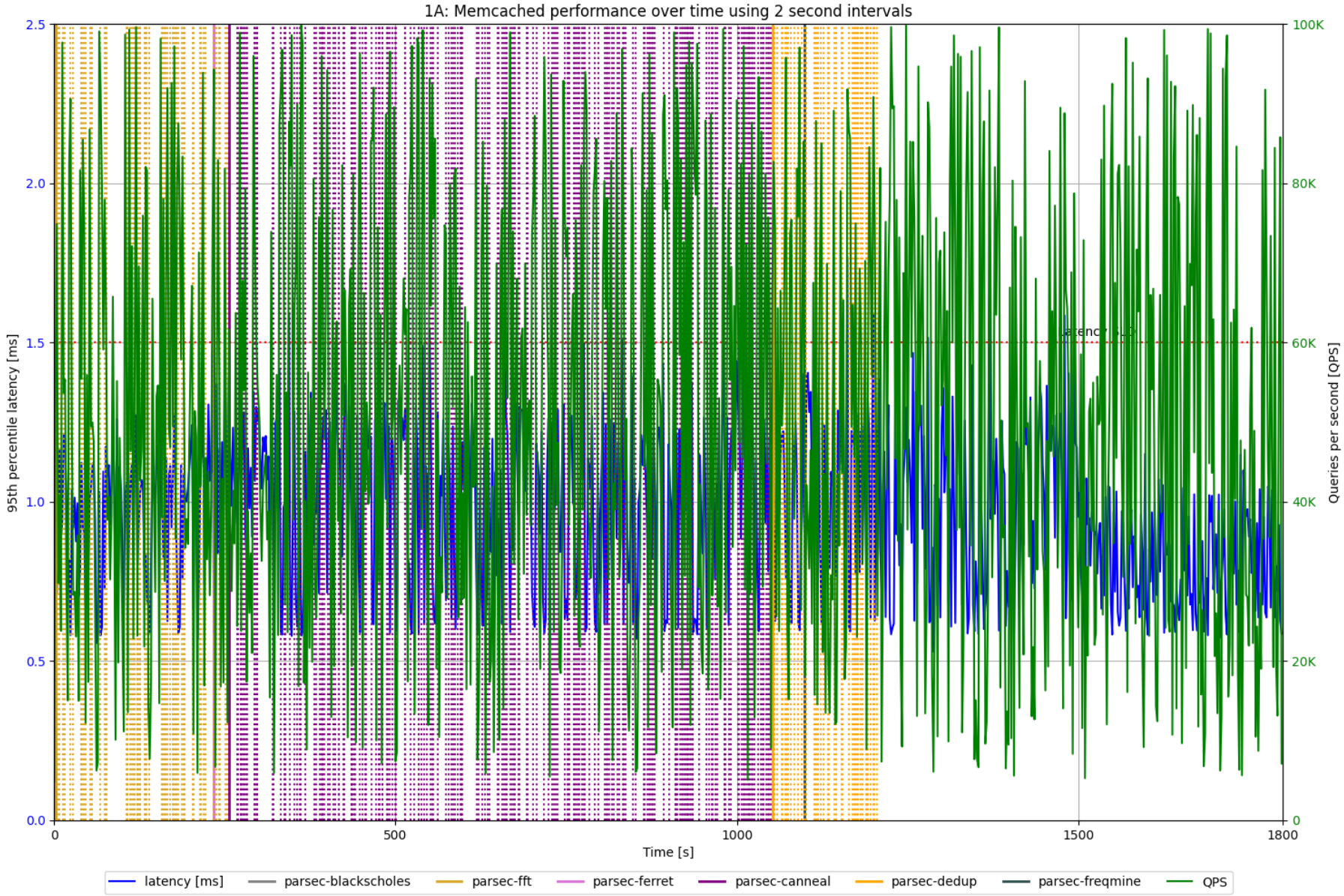

Figure 13: 1A: Memcached performance using 2 second intervals

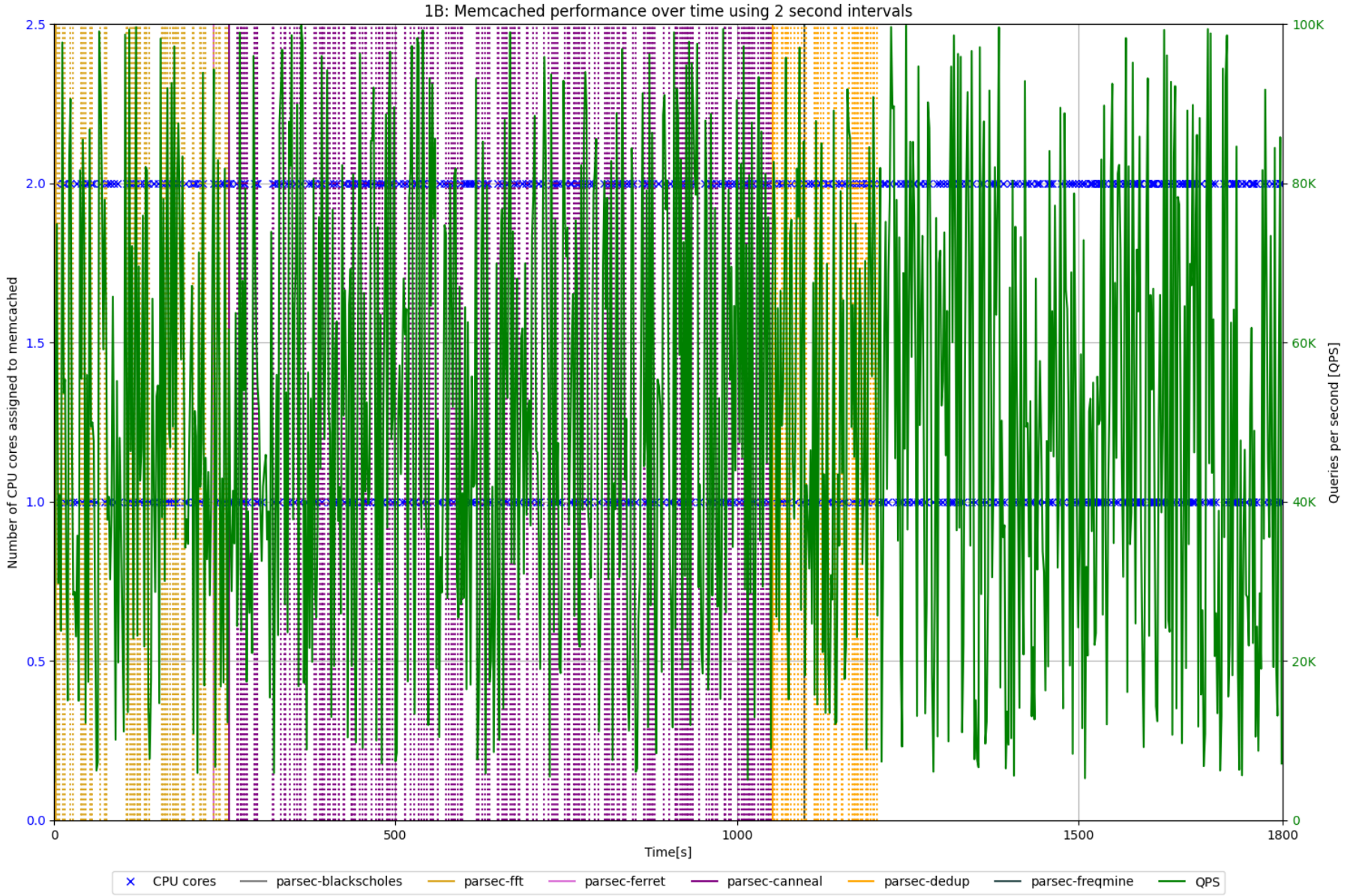

Figure 14: 1B: Memcached performance using 2 second intervals

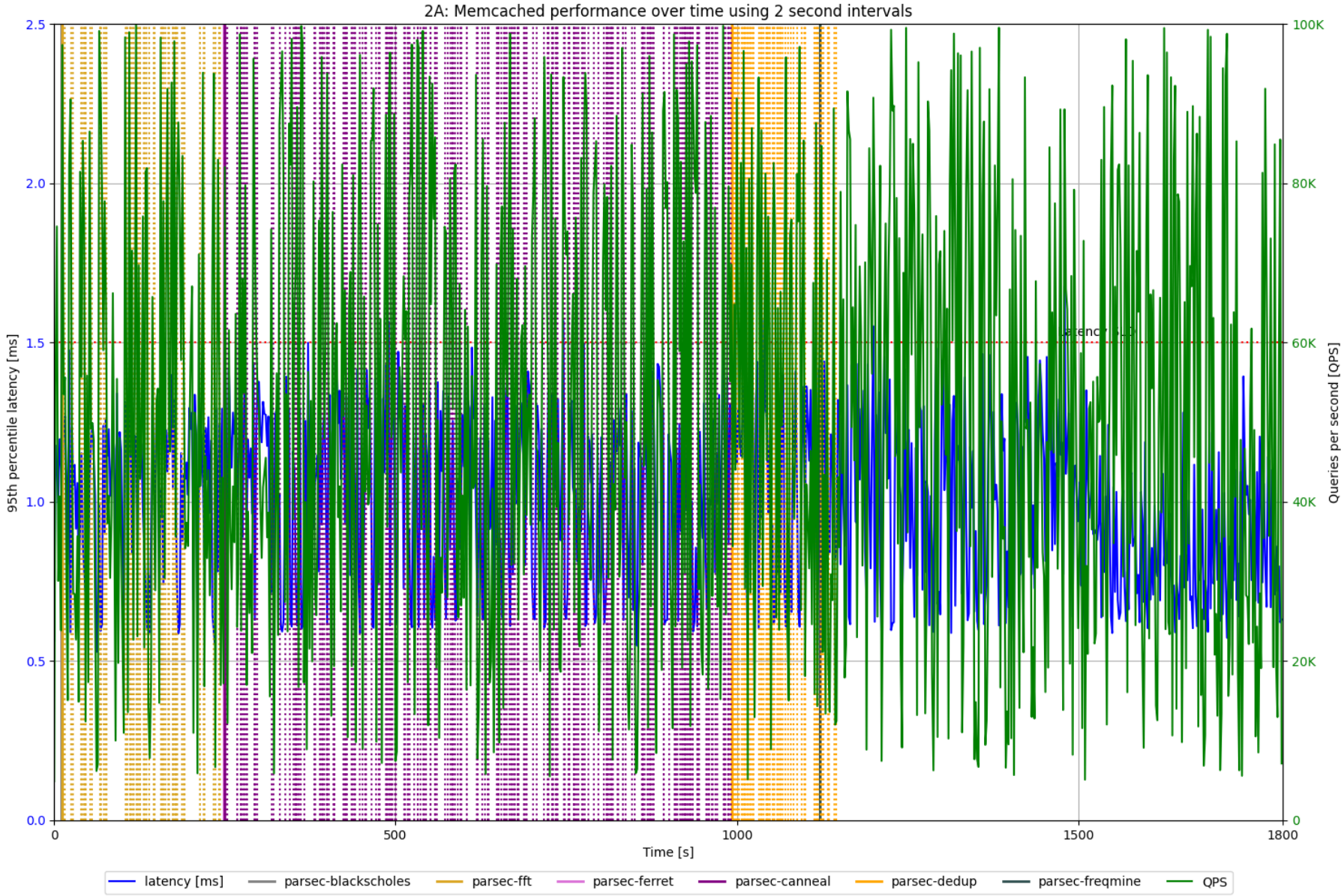

Figure 15: 2A: Memcached performance using 2 second intervals

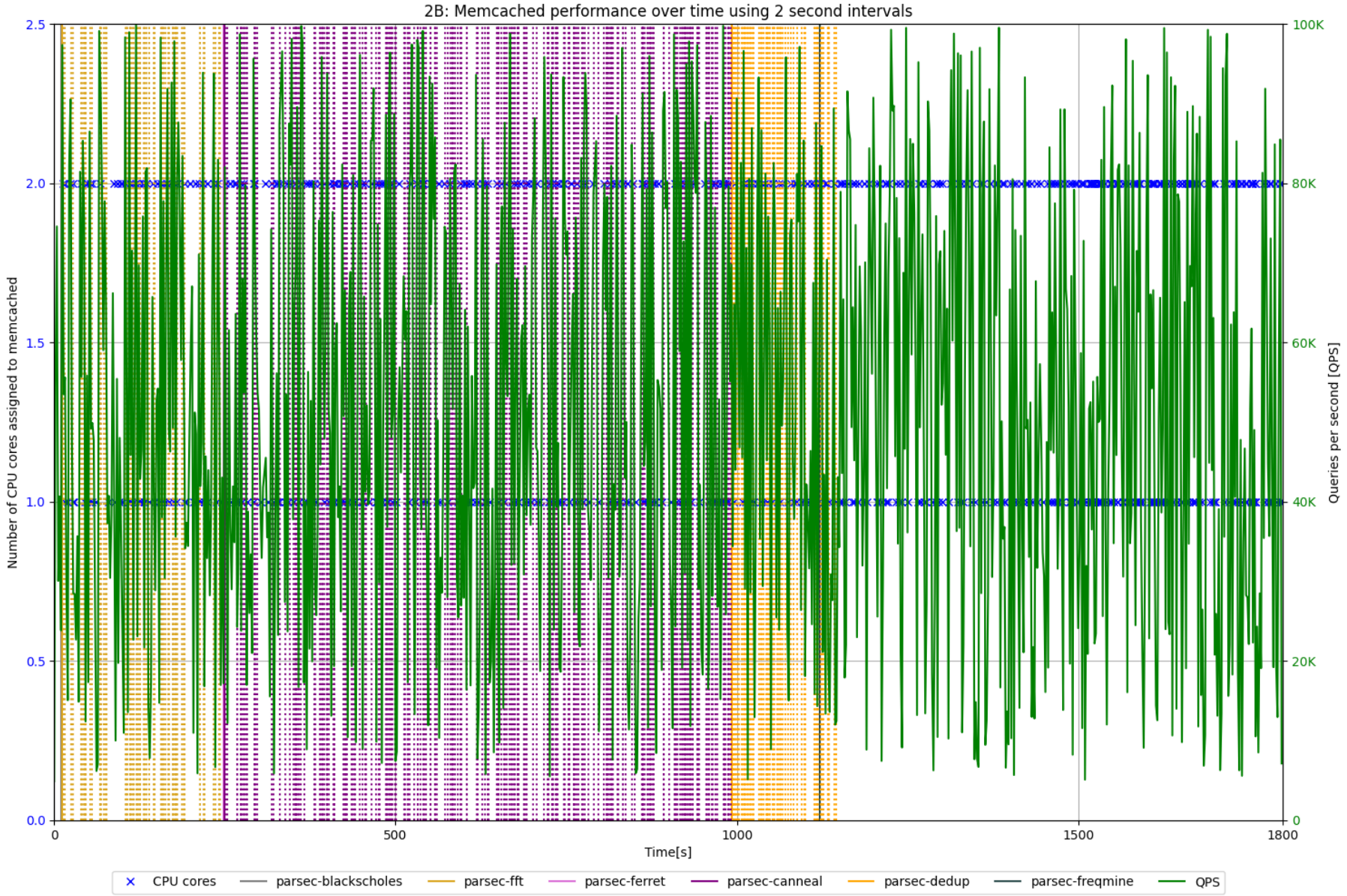

Figure 16: 2B: Memcached performance using 2 second intervals

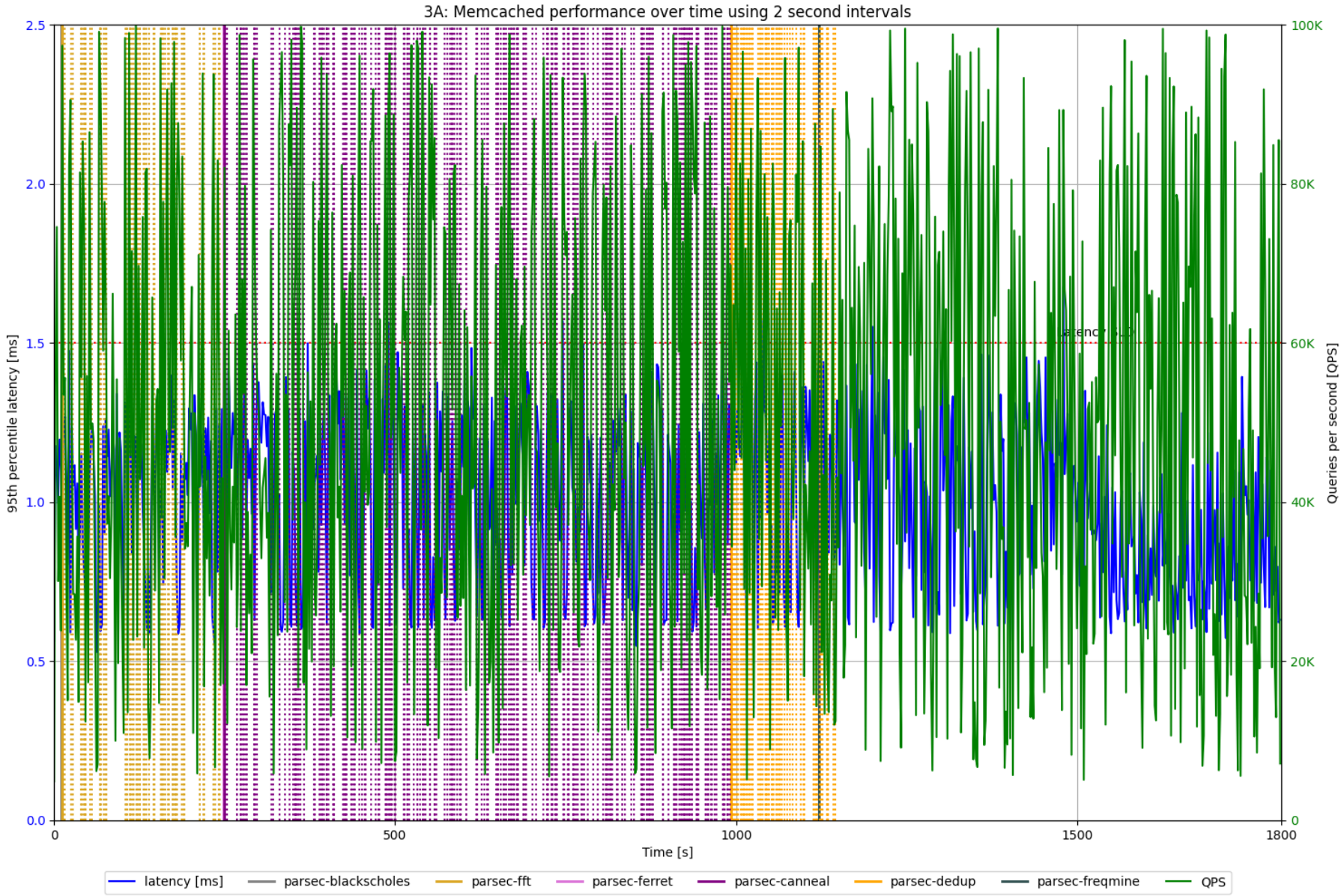

Figure 17: 3A: Memcached performance using 2 second intervals

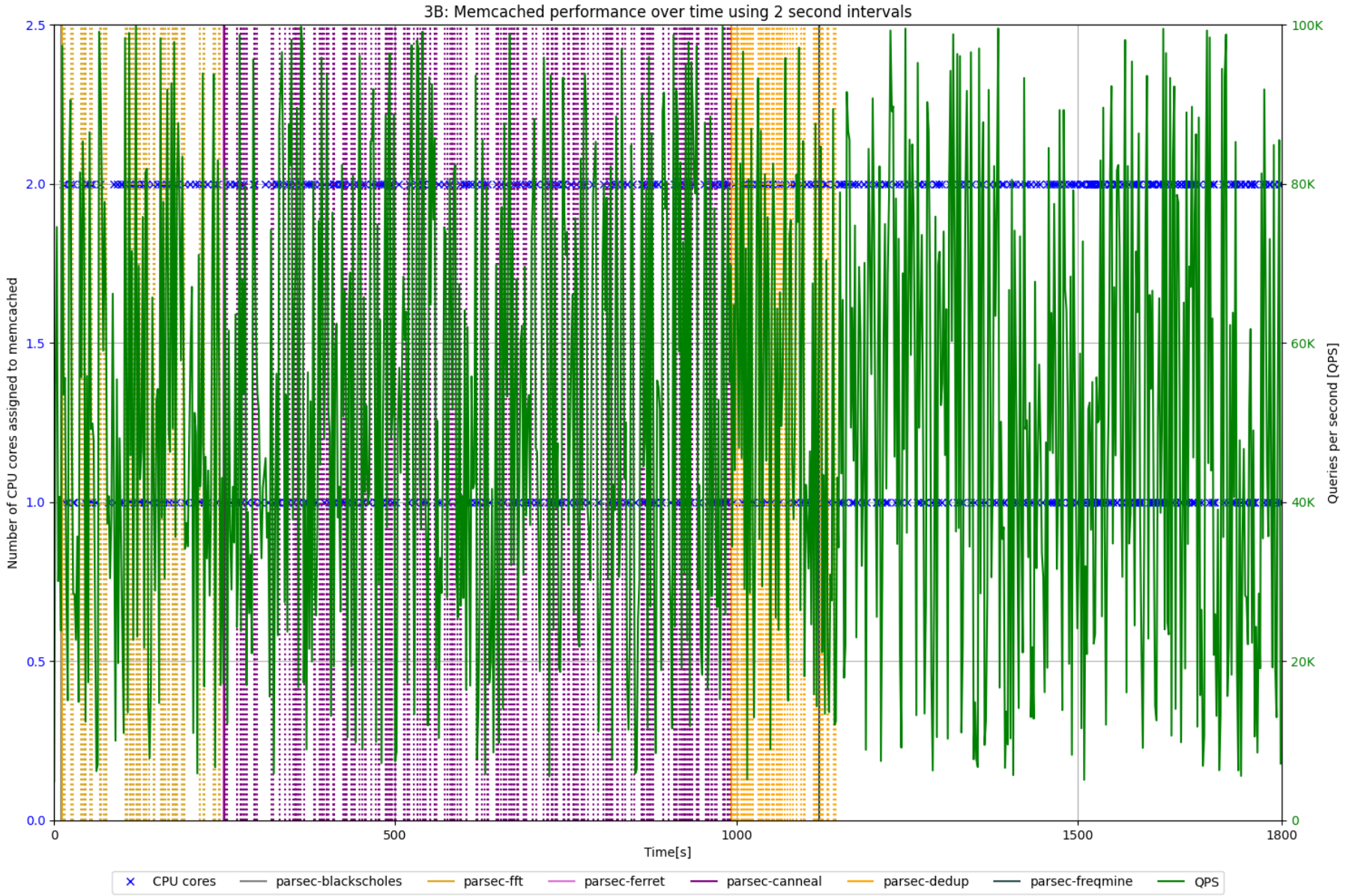

Figure 18: 3B: Memcached performance using 2 second intervals