Cloud Computing Architecture

Semester project

April 5, 2022

Overview

The semester project consists of four parts, two of which are described in detail in this handout. In this project, you will explore how to schedule latency-sensitive and batch applications in a cloud cluster. You will deploy applications inside containers and gain experience using a popular container orchestration platform, Kubernetes. Containers are a convenient and lightweight mechanism for packaging code and all its dependencies so that applications can run quickly and reliably from one computing environment to another.

You will work in groups of three students and submit a single report per group. **Please submit your report in the format of the project report template**. We will be assigning groups for the project, however you will have a chance to optionally let us know your preferences for teammates. If you know one or two other students in the class that you would like to work with on the project, please submit your group preference by March 11th, 2022. To do so, each student in your preferred group should sign up for the same group number in the Project Group Selection page on Moodle. We will notify you about final group assignments on March 14th and then you may redeem your cloud credits and begin working on the project.

Important Dates

March 11th, 2022: Deadline to submit group preferences (optional, since we will assign groups).
March 14th, 2022: Groups are assigned and announced. Start working on project.
April 12th, 2022 at 13:00: Deadline to submit Part 1 and 2 of the project.
May 27th, 2022 at 13:00: Deadline to submit Part 3 and 4 of the project.

We will release Part 3 and 4 of the project mid-April. Parts 3 and 4 are more open-ended and will require more time to complete than Part 1 and 2. Please plan your time accordingly.

Cloud Environment and Credits

To run experiments for the project, you will use Google Cloud. We will provide you with Google Cloud credits for your project. To redeem your cloud credits, please follow the steps in Part 1 (Section 1.1), when your project group assignment is confirmed. Each group member should create a Google Cloud account at `https://accounts.google.com`. Please use your ETH email address to create the account.

1 Part 1

In Part 1 of this project, you will run a latency-critical application, memcached, inside a container. Memcached is a distributed memory caching system that serves requests from clients over the network. A common performance metric for memcached is the tail latency (e.g., 95th percentile latency) under a desired query rate. You will measure tail latency as a function of queries per second and explore the impact of hardware resource interference. To add different types of hardware resource contention, you will use the iBench microbenchmark suite to apply different sources of interference (e.g., CPU, caches, memory bandwidth).

Follow the setup instructions below to deploy a Google Cloud cluster using the `kops` tool. Your cluster will consist of four virtual machines (VMs). One VM will serve as the Kubernetes cluster master, one VM will be used to run the memcached server application and iBench workloads, and two VMs will be used to run a client program that generates load for the memcached server.

In your project report (see report **template**), answer the questions in Section **??**.

1.1 Setup Instructions

Installing necessary tools

For the setup of the project, you will need to install **kubernetes**, **google-tools** and **kops**. Instructions based on the operating system on your local machine are provided in the links above. Having installed all the tools successfully, the following three commands should return output in your terminal (for the rest of the document the $ symbol is there to declare a bash command and you shouldn't type it explicitly):

1. `$ kubectl --help`
2. `$ kops --help`
3. `$ ./google-cloud-sdk/bin/gcloud --help`

Note that the final command is relative to where you have downloaded the google cloud tools. If you have installed via a package manager or have added the gcloud tools to your `$PATH` you don't need the prefix and you can just type `gcloud`. Note that you have to open a new terminal or refresh your shell using `source` for your `$PATH` to be updated.

All the scripts that you will need for both parts of the project are available **here**:

```
git clone https://github.com/eth-easl/cloud-comp-arch-project.git
```

Redeeming cloud credits and creating Google Cloud project

Each group member should create a Google Cloud account at `https://accounts.google.com`. Use your ETH email address to create the account. Each group will receive a $50 Google Cloud coupon code. Select **one** group member to enter their name and ETH email address at **this link**. This student must use their **@student.ethz.ch** or **@ethz.ch** email address for the form. A confirmation email will be sent to the student with a coupon code. You can redeem the coupon using the following link, **using your ETH email address**.

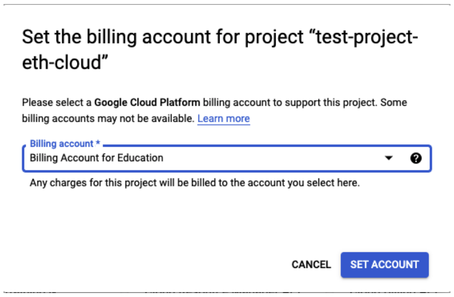
After installing kubernetes tools, connect your local client to your google cloud account using:

```
gcloud init
```

A browser window will open and you will have to login in with your ETH address. Afterwards, you will give `google-cloud-sdk` permissions to your account and then in the command line you will pick a name for the project. When creating the project name use `cca-eth-2022-group-XXX` (where XXX is your group number). **Only one group member (who also redeemed the cloud credit coupon) should create the Google Cloud project.** This person will add other group members as Project Owners (see instructions below). After the other group members are added as Project Owners, they will simply select the existing project name when they run the `gcloud init` command. All group members will have access to the project and share the cloud credits.

Do not configure any default computer region and zone. For deploying a cluster on Google Cloud we will modify some of the instructions listed here, which will be given below.

After creating the project you can log into the google cloud console and will be prompted to select a billing account for the project. In the pop up choose `Billing account for education` as below and click `Set account`:



Set the billing account for project "test-project-eth-cloud"

Please select a **Google Cloud Platform** billing account to support this project. Some billing accounts may not be available. Learn more

Billing account *: Billing Account for Education

Any charges for this project will be billed to the account you select here.

Afterwards, you can try the command `gcloud compute zones list`. The first time you should get a prompt to enable the compute engine API that looks like this:

```
    API [compute.googleapis.com] not enabled on project [project number].
    Would you like to enable and retry (this will take a few minutes)? (y/N)?
```

After the API is enabled you can repeat the command which should now yield the following output:

```
$ gcloud compute zones list
NAME                       REGION                   STATUS
us-east1-b                 us-east1                 UP
us-east1-c                 us-east1                 UP
us-east1-d                 us-east1                 UP
us-east4-c                 us-east4                 UP
us-east4-b                 us-east4                 UP
us-east4-a                 us-east4                 UP
us-central1-c              us-central1              UP
us-central1-a              us-central1              UP
```

```
us-central1-f              us-central1              UP
us-central1-b              us-central1              UP
us-west1-b                 us-west1                 UP
us-west1-c                 us-west1                 UP
us-west1-a                 us-west1                 UP
europe-west4-a             europe-west4             UP
europe-west4-b             europe-west4             UP
europe-west4-c             europe-west4             UP
europe-west1-b             europe-west1             UP
europe-west1-d             europe-west1             UP
europe-west1-c             europe-west1             UP
europe-west3-c             europe-west3             UP
europe-west3-a             europe-west3             UP
europe-west3-b             europe-west3             UP
europe-west2-c             europe-west2             UP
europe-west2-b             europe-west2             UP
europe-west2-a             europe-west2             UP
```

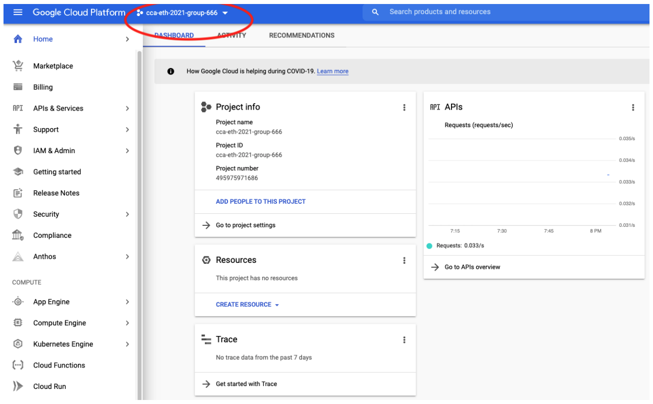
Then you will need to configure your default credentials using:

```
$ gcloud auth application-default login
```

This will redirect you to a browser window where you will login with the same account you used when you setup the `gloud init` command.

Giving your teammates owner permission to the project

After creating the `cca-eth-2022-group-XXX` project on Google Cloud, give your group members access to the project and cloud credits by navigating to the Google Cloud console menu. Make sure your project is properly displayed on the top left as below:



Project info

Project name
cca-eth-2021-group-666

Project ID
cca-eth-2021-group-666

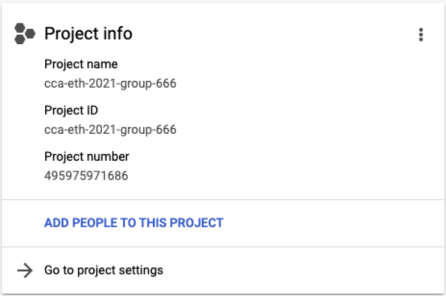
Project number
495975971686

This project has no resources

No trace data from the past 7 days

Requests: 0.033/s

In the project info click `Add people to this project`.



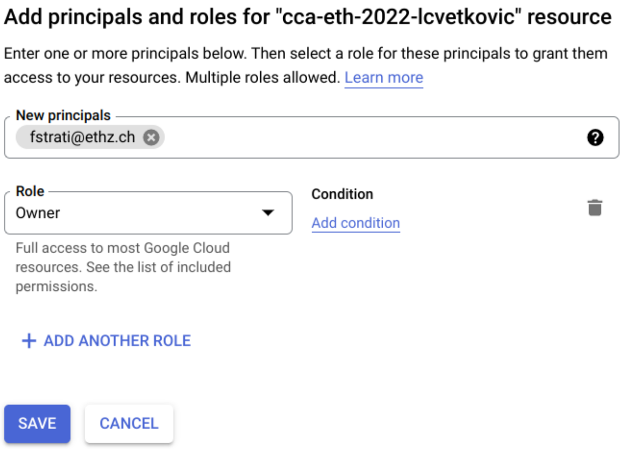
Project info

Project name
cca-eth-2021-group-666

Project ID
cca-eth-2021-group-666

Project number
495975971686

Type the email addresses of your teammates, select `Owner` as a role and click `Save`. **Note that your teammates should have created a google cloud account with their ETH address in advance to put them as project owners.**



Add principals and roles for "cca-eth-2022-lcvetkovic" resource

Enter one or more principals below. Then select a role for these principals to grant them access to your resources. Multiple roles allowed.

New principals: fstrati@ethz.ch

Role: Owner

Full access to most Google Cloud resources. See the list of included permissions.

Condition

Deploying a cluster using kops

At this point you will deploy a cluster using `kops`. First of all you will need to create an empty bucket to store the configuration for your clusters. Do this by running:

```
$ gsutil mb gs://cca-eth-2022-group-XXX-ethzid/
```

... where `XXX` is your group number and `ethzid` is your ETH username. Then run the following command to have the `KOPS_STATE_STORE` command to your environment for the subsequent steps:

```
$ export KOPS_STATE_STORE=gs://cca-eth-2022-group-XXX-ethzid/
```

If you open another terminal this and other environmental variables will not be preserved. You can preserve it by adding it with an `export` command to your .bashrc. You should substitute the number of your group and your ETH username as before.
Small Hint: Since Windows users cannot use the "export" command, you can manually add this environment variable. This tip also applies to the following "export" commands and "PROJECT=‘gcloud config get-value project‘" commands.

For the first part of the exercise you will need a 3 node cluster. Two VMs will have 2 cores. One of these VMs will be the node where `memcached` and `iBench` will be deployed and another will be used for for the `mcperf` memcached client which will measure the round-trip latency of memcached requests. The third VM will have 8 cores and hosts the `mcperf` client which generates the request load for the experiments.

Before you deploy the cluster with `kops` you will need an ssh key to login to your nodes once they are created. Execute the following commands to go to your `.ssh` folder and create a key:

```
$ cd ~/.ssh
$ ssh-keygen -t rsa -b 4096 -f cloud-computing
```

Once you have created the key, go to lines 16 and 43 of the `part1.yaml` file (provided in the github link above) and **substitute the placeholder values with your group number and ethzid**. Then run the following commands to create a kubernetes cluster with 1 master and 2 nodes.

```
$ PROJECT=`gcloud config get-value project`
$ export KOPS_FEATURE_FLAGS=AlphaAllowGCE # to unlock the GCE features
$ kops create -f part1.yaml
```

We will now add the key as a login key for our nodes. Type the following command:

```
$ kops create secret --name part1.k8s.local sshpublickey admin -i ~/.ssh/cloud-computing.pub
```

We are ready now to deploy the cluster by typing:

```
$ kops update cluster --name part1.k8s.local --yes --admin
```

Your cluster should need around 5-10 minutes to be deployed. You can validate this by typing:

```
$ kops validate cluster --wait 10m
```

The command will terminate when your cluster is ready to use. If you get a `connection refused` or `cluster not yet healthy` messages, wait while the previous command automatically retries. When the command completes, you can type:

```
$ kubectl get nodes -o wide
```

... to get the status and details of your nodes as follows:

```
NAME                         STATUS   ROLES    AGE    VERSION   INTERNAL-IP   EXTERNAL-IP
master-europe-west3-a-2s21   Ready    master   3m2s   v1.19.7   10.156.0.63   34.107.107.152
memcache-server-jrk4         Ready    node     102s   v1.19.7   10.156.0.61   34.107.94.26
client-agent-vg5v            Ready    node     98s    v1.19.7   10.156.0.62   34.89.236.52
client-measure-ngwk          Ready    node     102s   v1.19.7   10.156.0.60   35.246.185.27
```

You can connect to any of the nodes by using your generated ssh key and the node name. For example to connect to the client-agent node, you can type:

```
$ gcloud compute ssh --ssh-key-file ~/.ssh/cloud-computing ubuntu@client-agent-vg5v \
                     --zone europe-west3-a
```

Running memcached and the mcperf load generator

To launch memcached using Kubernetes, run the following:

```
$ kubectl create -f memcache-t1-cpuset.yaml
$ kubectl expose pod some-memcached --name some-memcached-11211  \
                                    --type LoadBalancer --port 11211 \
                                    --protocol TCP
$ sleep 60
$ kubectl get service some-memcached-11211
```

Then run the following:

```
$ kubectl get pods -o wide
```

The output should look like:

```
NAME             READY   STATUS    RESTARTS   AGE   IP           NODE
some-memcached   1/1     Running   0          42m   100.96.3.3   memcache-server-zns8
```

Use the IP address above (`100.96.3.3` in this example) as the `MEMCACHED_IP` in the remaining instructions. Now ssh into both the `client-agent` and `client-measure` VMs and run the following commands to compile the `mcperf` memcached load generator:

```
$ sudo apt-get update
$ sudo apt-get install libevent-dev libzmq3-dev git make g++ --yes
$ sudo cp /etc/apt/sources.list /etc/apt/sources.list~
$ sudo sed -Ei 's/^# deb-src /deb-src /' /etc/apt/sources.list
$ sudo apt-get update
$ sudo apt-get build-dep memcached --yes
$ cd && git clone https://github.com/shaygalon/memcache-perf.git
$ cd memcache-perf
$ make
```

On the `client-agent` VM, you should now run the following command to launch the `mcperf` memcached client load agent with 16 threads:

```
$ ./mcperf -T 16 -A
```

On the `client-measure` VM, run the following command to first load the memcached database with key-value pairs and then query memcached with throughput increasing from 5000 queries per second (QPS) to 55000 QPS in increments of 5000:

```
$ ./mcperf -s MEMCACHED_IP --loadonly
$ ./mcperf -s MEMCACHED_IP -a INTERNAL_AGENT_IP  \
           --noload -T 16 -C 4 -D 4 -Q 1000 -c 4 -t 5 \
           --scan 5000:55000:5000
```

... where `MEMCACHED_IP` is from the output of `kubectl get pods -o wide` above and `INTERNAL_AGENT_IP` is from the Internal IP of the client-agent node from the output of `kubectl get nodes -o wide`. You should look at the output of `./mcperf -h` to understand the different flags in the above commands.

Introducing Resource Interference

Now we are going to introduce different types of resource interference with iBench microbenchmarks. Run the following commands:

```
$ kubectl create -f interference/ibench-cpu.yaml
```

This will launch a CPU interference microbenchmark. You can check it is running correctly with:

```
$ kubectl get pods -o wide
```

(wait until READY 1/1 and STATUS Running shows before starting a run).

When you have finished collecting memcached performance measurements with CPU interference, you should kill the job by running:

```
$ kubectl delete pods ibench-cpu
```

You can apply the above three steps for any of the six `ibench-cpu, ibench-l1d, ibench-l1i, ibench-l2, ibench-llc`, and `ibench-membw` interference microbenchmarks. For Part 1 you will perform experiments to investigate the effect of the different types of interference. After now having followed this tutorial, you are able to run those experiments. First, start with reading the information of what to run for Part 1 in the project report template.

Deleting your cluster

IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project, you can easily re-launch the cluster with the instructions above.
To delete your cluster, run on your local machine the command:

```
$ kops delete cluster part1.k8s.local --yes
```

2 Part 2

In Part 2 of this project, you will run six different throughput-oriented ("batch") workloads from the PARSEC benchmark suite: `blackscholes`, `fft`, `dedup`, `ferret`, `freqmine`, and `canneal`. You will first explore each workload's sensitivity to resource interference using iBench on a small 2 core VM (`e2-standard-2`). This is somewhat similar to what you did in Part 1 for `memcache`. Next, you will investigate how each workload benefits from parallelism by measuring the performance of each job with 1, 3, 6, 12 threads on a large 8 core VM (`e2-standard-8`). In the latter scenario, no interference is used.

Follow the setup instructions below to deploy a Google Cloud cluster and run the batch applications. In your project report, answer the questions in Section ??.

2.1 Setup

In order to complete this Part of the project, we will have to study the behavior of PARSEC in two different contexts. For both, we will require that `kubectl`, `kops` and `gcloud sdk` are set up. This should already be the case if you have completed Part 1.

We have provided you with a set of `yaml` files which are useful towards spawning `kubectl` jobs for workloads and interference. The interference files are the same as in Part 1, **but you must change the nodetype from memcached to parsec**. The PARSEC workloads are in the `parsec-benchmarks/part2a` folder in the github repo. All these files cover the workloads in the PARSEC suite, as well as the `iBench` interference sources relevant for this part: `cpu`, `l1d`, `l1i`, `l2`, `llc`, `memBW`.

2.1.1 PARSEC Behavior with Interference

For the first half of Part 2, you will have to set up a single node cluster consisting of a VM with 2 CPUs. For this, we will employ `kops` and make use of the `part2a.yaml` file (make sure to update the file with values for your GCP project and configBase):

```
$ export KOPS_STATE_STORE=<your-gcp-state-store>
$ export KOPS_FEATURE_FLAGS=AlphaAllowGCE
$ PROJECT=`gcloud config get-value project`
$ kops create -f part2a.yaml
$ kops update cluster part2a.k8s.local --yes --admin
$ kops validate cluster --wait 10m
$ kubectl get nodes -o wide
```

If successful, you should see something like this:

```
NAME                        STATUS   ROLES    AGE    VERSION   INTERNAL-IP   EXTERNAL-IP
master-europe-west3-a-9nxl  Ready    master   3m2s   v1.19.7   10.156.0.46   34.107.0.118
parsec-server-s28x          Ready    node     104s   v1.19.7   10.156.0.47   35.234.110.58
```

Now you should be able to connect to the `parsec-server` VM using either `ssh`:

```
$ ssh -i ~/.ssh/cloud-computing ubuntu@35.234.110.58
```

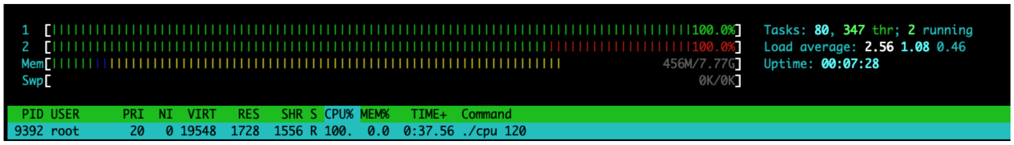
Or by using `gcloud`:

```
$ gcloud compute ssh --ssh-key-file ~/.ssh/cloud-computing ubuntu@parsec-server-s28x \
                     --zone europe-west3-a
```

For this part of the study we will sometimes require to set up some form of interference, and also deploy a `PARSEC` job. For this example, we will use the `PARSEC fft` job together with `iBench` CPU interference. Here is where we will use `kubectl` together with some of the `yaml` files we provide. The following code snippet spins up the interference, and runs the `PARSEC fft` job:

```
$ kubectl create -f interference/ibench-cpu.yaml  # Wait for interference to start
$ kubectl create -f parsec-benchmarks/part2a/parsec-fft.yaml
```

Make sure that the interference has properly started **before** running the `PARSEC` job. One way to see if the interference and the `PARSEC` job has started refers to `ssh`-ing into the VM and using the `htop` command to inspect running processes. You should see an image like below:



```
  1  [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]   Tasks: 80, 347 thr; 2 running
  2  [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]   Load average: 2.56 1.08 0.46
  Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||          456M/7.77G]   Uptime: 00:07:28
  Swp[                                                                            0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 9392 root       20   0 19548  1728  1556 R 100.  0.0  0:37.56 ./cpu 120
```

You can get information on submitted jobs using:

```
$ kubectl get jobs
```

In order to get the output of the `PARSEC` job, you will have to collect the logs of its pods. To do so, you will have to run the following commands:

```
$ kubectl logs $(kubectl get pods --selector=job-name=<job_name> \
        --output=jsonpath='{.items[*].metadata.name}')
```

Note that the job name needs to match the one you get from `kubectl get jobs`.

Run experiments sequentially and wait for one benchmark to finish before you spin up the next one. *Since fft, ferret, freqmine, and canneal jobs take over 5 minutes to complete with the native dataset, you must use a smaller dataset called simlarge (use -i simlarge in the YAML file) for these jobs. Use the native dataset (-i native) for dedup and blackscholes jobs. The .yaml files provided have the correct configuration.* Once you are done with running one experiment, make sure to terminate the started jobs. You can terminate them all together using:

```
$ kubectl delete jobs --all
$ kubectl delete pods --all
```

Alternatively, you can do so one-by-one using the following command:

```
$ kubectl delete job <job_name>
```

IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project, you can easily re-launch the cluster with the instructions above. To delete your cluster, use the command:

```
$ kops delete cluster part2a.k8s.local --yes
```

2.1.2 PARSEC Parallel Behavior

For the second half of Part 2, you will have to look into the parallel behavior of PARSEC, more specifically, how does the performance of various jobs in PARSEC change as more threads are added (more specifically 1, 3, 6 and 12 threads). For this part of the study, no interference is used.

You will first have to spawn a cluster as in section 2.1.1, however, this time use the part2b.yaml file we provided (make sure to update the file with values for your GCP project and configBase). Once more, this will be a single node cluster with an 8 CPU VM. You will have to vary the number of threads for each PARSEC job. To do so, change the value of the -n parameter in the relevant yaml files. You should run all these benchmarks with the *native* dataset. The corresponding .yaml files are in parsec-benchmarks/part2b of the github repo.

The fft benchmark does not scale to our target number of threads (3,6,12), as it only runs with thread numbers equal to powers of two. For this reason you only need to measure single thread performance for the fft benchmark.

Other relevant instructions for this task can be found in section 2.1.1.

IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project, you can easily re-launch the cluster with the instructions above. To delete your cluster, use the command:

```
$ kops delete cluster part2b.k8s.local --yes
```

3 Part 3

In Part 3, you will put the previous two parts of the project together. You will now co-schedule the latency critical memcached application from Part 1 and all six batch applications from Part 2 in a heterogeneous cluster consisting of VMs with different numbers of cores. Your cluster will consist of a VM for the Kubernetes master (same as in Part 1), 3 VMs for the memcperf clients (2 agents and 1 measure machine), and 3 heterogeneous VMs (each with 2, 4, and 8 cores) which will be labelled `node-a-2core`, `node-b-4core`, `node-c-8core`, respectively, and used to run memcached and the PARSEC batch applications. Note that these VMs also have different configurations (as you can see in the `part3.yaml` file): `node-a-2core` is of type `n2d-highcpu-2`, `node-b-4core` is of type `n2d-highmem-4`, and `node-c-8core` is of type `e2-standard-8`. The number of CPUs, the CPU platform, and the amount of memory differ in these VMs, which is something that you should take into account in your scheduling policy.

Your goal is to design a scheduling policy to minimize the time it takes for all six batch workloads to complete while guaranteeing a tail latency service level objective (SLO) for the long-running memcached service. It might be helpful to take into account the characteristics of the PARSEC jobs that you noted in Part 2 (e.g. speedup across cores, total running time). For this part of the project, the memcached service will receive requests from the client at a steady rate and you will measure the request tail latency. Your scheduling policy should minimize the total time to complete all the PARSEC apps, without violating a strict service level objective for memcached of **2 ms** 95th percentile latency at **30K QPS**. You must ensure that all six PARSEC apps succeed because a job may stop due to errors (e.g., out of memory). Use the native dataset size for all PARSEC jobs. At every point, you must use as many resources of your cluster as possible.

To design and implement your scheduling policy, you will experiment with different job collocations and resource management strategies using mechanisms in Kubernetes. Use what you learned about the performance characteristics of each application in Parts 1 and 2 to decide with which degree of parallelism to run each workload and which applications to collocate on shared resources.

Hint: You may modify the YAML files provided, write a script for launching the batch jobs, or apply any other techniques you choose, as long as you describe them clearly in your report. You can choose which jobs to collocate, which degree of parallelism to use for the batch and memcached workloads, and when to launch particular batch jobs. Use any Kubernetes mechanism you wish to implement a scheduling policy. You may find node/pod affinity and/or resource requests/limits particularly useful. You may also want to use `taskset` in the container command arguments to pin containers to certain CPU cores on a node. Keep in mind that a job may fail due to lack of resources. Use `kubectl describe jobs` to monitor jobs.

3.1 Setup

Run the following command to create a kubernetes cluster with 1 master and 6 nodes. Make sure to update the `part3.yaml` file with the name of your project and your ConfigBase.

```
$ export KOPS_STATE_STORE=<your-gcp-state-store>
$ PROJECT='gcloud config get-value project'
$ export KOPS_FEATURE_FLAGS=AlphaAllowGCE # to unlock the GCE features
$ kops create -f part3.yaml
```

We are ready now to deploy the cluster by typing:

```
$ kops update cluster --name part3.k8s.local --yes --admin
```

Your cluster should need around 5-10 minutes to be deployed. You can validate the cluster with the command:

```
$ kops validate cluster --wait 10m
```

The command will terminate when your cluster is ready to use. Afterwards you can type

```
kubectl get nodes -o wide
```

to get the status and details of your nodes as follows:

```
NAME                          STATUS   ROLES    AGE   VERSION   INTERNAL-IP     EXTERNAL-IP
client-agent-a-d81z           Ready    node     23m   v1.19.7   10.156.15.222   35.234.120.124
client-agent-b-xpt7           Ready    node     23m   v1.19.7   10.156.15.224   34.107.4.82
client-measure-x1xw           Ready    node     23m   v1.19.7   10.156.15.223   35.242.212.158
master-europe-west3-a-cdp2    Ready    master   24m   v1.19.7   10.156.15.225   34.89.196.131
node-a-2core-qtrb             Ready    node     23m   v1.19.7   10.156.15.221   34.89.217.203
node-b-4core-gq6s             Ready    node     23m   v1.19.7   10.156.15.220   34.107.20.21
node-c-8core-3kz9             Ready    node     23m   v1.19.7   10.156.15.226   34.107.23.202
```

To connect to any of the machines you can run:

```
$ gcloud compute ssh --ssh-key-file ~/.ssh/cloud-computing ubuntu@<MACHINE_NAME> \
                     --zone europe-west3-a
```

Modify the memcached and PARSEC batch job YAML files from Parts 1 and 2 of the project and use the `kubectl create` commands to launch the workloads in the cluster. Write scripts to launch the jobs. The memcached job should start first and continue running throughout the whole experiment while receiving constant load of 30K QPS from the `mcperf` client). After you have started memcached and the client load, you can start the PARSEC batch jobs in whichever order you choose. Your goal is to minimize the time from the start of the first PARSEC job to the time the last PARSEC job completes while ensuring that the 95th percentile latency for memcached remains below 2ms.

To setup the `mcperf` load generator on the `client-agent` and `client-measure` machines, follow the instructions from Part 1. Instead of sweeping request throughput as you did in Part 1, you now want to generate load at a constant rate of approximately 30K QPS while reporting latency periodically (e.g., every 20 seconds). To do this, run the following command on the `client-agent-a` machine:

```
$ ./mcperf -T 2 -A
```

and the following command on the `client-agent-b` machine:

```
$ ./mcperf -T 4 -A
```

and the following command on the `client-measure` VM:

```
$ ./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
```

You can get the execution time of the batch jobs by parsing the JSON output of the `kubectl` command that returns information about each job, including its start and completion time. To do that, after all jobs have completed run:

```
$ kubectl get pods -o json > results.json
$ python3 get_time.py results.json
```

where `get_time.py` is a python script that we have provided you with.
IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project, you can easily re-launch the cluster with the instructions above.
To delete your cluster, use the command:

```
$ kops delete cluster part3.k8s.local --yes
```

3.2 Questions

The questions for Part 3 are stated in the project report template.

4 Part 4

In Part 4, you will co-schedule PARSEC batch jobs on a single 4-core server running memcached. In contrast to Part 3, we will now dynamically vary the load on the long-running memcached service, such that the number of cores needed by the memcached server to meet the tail latency service level objective (SLO) ranges from 1 to 2 cores. Your will design a scheduling policy that grows and shrinks the resource allocation of memcached and opportunistically uses temporarily available cores to complete the PARSEC jobs as quickly as possible. Your scheduling policy must guarantee a memcached tail latency SLO of 1.5ms 95th percentile latency. For this part you will be using a cluster consisting of 4 nodes: a 2 core VM cluster master, a 4 core high memory VM for the `memcached` server and PARSEC jobs, a 16 core VM for the `mcperf` agent, and a 2 core VM for the `mcperf` measurement machine.

You will need to implement your own controller to launch jobs and dynamically adjust their available resources, based on your scheduling policy. In this part of the project, we will not use Kubernetes because Kubernetes does not provide an API to change a container's resource allocation during run time. Instead, you will use Docker to launch containers to run the PARSEC benchmarks and dynamically adjust their resources. For memcached, we provide instructions for installing and running memcached directly on the VM (rather than in a Docker container) and using the `taskset` command to dynamically adjust resources. The reason we do not use Docker to run memcached in this part of the assignment is because we have observed that memcached's resources are not effectively constrained with `docker --cpuset-cpus`, since most of the processing in memcached application is network packet processing, which executes in kernel threads. Your controller should monitor CPU utilization and/or other types of resources and metrics to decide if resources need to be adjusted to meet the SLO. Your controller should make dynamic resource allocation decisions, such that the PARSEC batch jobs are completed as quickly as possible while still enforcing memcached's SLO.

For this part we also provide an augmented version of `mcperf` which is capable of generating random loads on the memcached server, as well as specific load traces. We provide instructions on how to set this up further below in Section 4.1.1.

Implementing the controller and scheduling policy

We recommend implementing your controller in python and using the Docker Python SDK to manage containers. Alternatively, you may implement the controller in Go using the Docker Go SDK. You can find examples of managing containers using the Docker SDK, for both Python and Go. If you plan on using such an SDK, you might find it useful to use the shell command `sudo usermod -a -G docker <your-username>`. This will allow you to use the SDK programmatically without encountering permission errors. You will also be able to run docker commands without using `sudo`.

In addition to *running* containers, you will need to *update* containers while they are running. Updating a container refers to dynamically adjust properties of the container, such as the CPU allocation. You can read more about updating containers in the Docker update command documentation. You can update docker containers using Docker SDK commands. In case you find it helpful for your scheduling policy (though it is not required), you can also pause and unpause containers. Pausing a container has the effect of temporarily stopping the execution of the processes in the container (i.e., releasing CPU resources) while retaining the container's state (i.e., keeping the container's memory resources). Unpausing containers resumes the execution of processes in the container.

Your controller should run on the 4-highmem core memcached server and monitor CPU utilization. The controller should use CPU utilization statistics to make dynamic scheduling decisions. You can monitor CPU utilization on the server by reading and post-processing data from `/proc/stat` files on the VM. There are also language specific options for monitoring metrics, such as psutil for Python.

In addition to CPU utilization, you may use other inputs for your scheduling policy, if you wish. This is not required, but may let you implement an even better policy. Please explain in your report any additional inputs you choose to consider in your scheduling policy at the controller.

Evaluating the scheduling policy

You will evaluate your scheduling policy with a dynamic `mcperf` load trace that we provide (see instructions below). You should use `mcperf` to investigate the performance of your scheduling policy with various load traces (e.g., try different random seeds and time intervals). Experimenting with various load traces will allow you to analyze when and why your policy performs well and understand in which scenarios the policy does not adapt appropriately.

Generating the plots

In this part of the project, you will be asked to generate some plots which often require that you aggregate data gathered from different VMs. This can be challenging since you'll need to temporally correlate this data across the VMs. A straight forward way to do this is to save the Unix time whenever you log an event, as this time is roughly synchronized across VMs. You can further use other information such as dynamic mcperf's `--qps_interval` or `-t` parameter (see documentation here). Our dynamic mcperf version should also by default print the simulation's start and end Unix times in the output logs. Another alternative is to use the shell command `date +%s`. These times can then be used when generating the plots to synchronize events that take place on different VMs.

4.1 Setup

4.1.1 Installation

Run the following command to create a kubernetes cluster with 1 master and 3 nodes.

```
$ export KOPS_STATE_STORE=<your-gcp-state-store>
$ PROJECT='gcloud config get-value project'
$ export KOPS_FEATURE_FLAGS=AlphaAllowGCE # to unlock the GCE features
$ kops create -f part4.yaml
```

We are ready now to deploy the cluster by typing:

```
$ kops update cluster --name part4.k8s.local --yes --admin
```

Your cluster should need around 5-10 minutes to be deployed. You can validate the cluster with the command:

```
$ kops validate cluster --wait 10m
```

The command will terminate when your cluster is ready to use. Afterwards you can type:

```
kubectl get nodes -o wide
```

To get the status and details of your nodes as follows:

```
NAME                     STATUS   ROLES    AGE     VERSION   INTERNAL-IP   EXTERNAL-IP
client-agent-bf7q        Ready    node     111s    v1.19.7   10.138.0.33   35.230.78.193
client-measure-5v6m      Ready    node     116s    v1.19.7   10.138.0.32   35.227.161.236
master-us-west1-a-kh69   Ready    master   3m23s   v1.19.7   10.138.0.34   35.247.63.197
memcache-server-qmql     Ready    node     111s    v1.19.7   10.138.0.31   34.83.56.78
```

You will first need to manually install memcached on the `memcache-sever` VM. To do so, you must first use the following commands:

```
$ sudo apt update
$ sudo apt install -y memcached libmemcached-tools
```

To make sure the installation succeeded, run the following command:

```
$ sudo systemctl status memcached
```

you should see and ouput similar to the following being produced:

```
 memcached.service - memcached daemon
   Loaded: loaded (/lib/systemd/system/memcached.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2021-04-01 08:21:26 UTC; 10min ago
     Docs: man:memcached(1)
 Main PID: 11796 (memcached)
    Tasks: 10 (limit: 4915)
   CGroup: /system.slice/memcached.service
           -11796 /usr/bin/memcached -m 64 -p 11211 -u memcache -l 127.0.0.1 ...
```

You will need to expose the service to the outside world, and increase its default starting memory. To do so, open memcached's configuration file using the command:

```
$ sudo vim /etc/memcached.conf
```

To update memcached's memory limit, look for the line starting with `-m` and update the value to 1024. Similarly, to expose the memcached server to external requests, locate the line starting with `-l` and replace the localhost address with the internal IP of the `memcache-sever` VM. You can also specify the number of memcached threads here by introducing a line starting with `-t` followed by the number of threads. Save the file, then execute the next command to restart memcached with the new configuration:

```
$ sudo systemctl restart memcached
```

Running `sudo systemctl status memcached` again should yield an output similar as before, however, you should see the updated parameters in the command line. If you completed these steps successfully, memcached should be running and listening for requests on the VMs internal IP on port 11211.

To install the augmented version of mcperf on the other 2 VMs, one can follow the instructions below:

```
$ sudo apt-get update
$ sudo apt-get install libevent-dev libzmq3-dev git make g++ --yes
$ sudo apt-get build-dep memcached --yes
$ git clone https://github.com/eth-easl/memcache-perf-dynamic.git
$ cd memcache-perf-dynamic
$ make
```

On the `client-agent` VM, you should now run the following command to launch the `mcperf` memcached client load agent with 16 threads:

```
$ ./mcperf -T 16 -A
```

On the `client-measure` VM, run the following command to first load the memcached database with key-value pairs and then query memcached with a dynamic load generator which will produce a random throughput between 5000 and 100000 queries per second during each interval. The throughput target will change and will be assigned to another QPS for the next interval. In the example the interval duration is set to 10 seconds, whilst the overall execution time is 30 seconds, which will result in three different QPS intervals:

```
$ ./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 10 \
           --qps_interval 2 --qps_min 5000 --qps_max 100000
```

The `INTERNAL_MEMCACHED_IP` and `INTERNAL_AGENT_IP` are the internal IPs of the `memcache-sever` and `client-agent` retrieved from the output of `kubectl get nodes -o wide`.

For more information on the dynamic load generator, and the available options it provides, check the guide in the README.md of the public repository.

PARSEC jobs can be started using Docker. For instance, one can start the `blackholes` job on core 0 (`--cpuset-cpus="0"` parameter) and with 2 threads (`-n 2` parameter) using the following command:

```
docker run --cpuset-cpus="0" -d --rm --name parsec \
    anakli/parsec:blackscholes-native-reduced \
    ./bin/parsecmgmt -a run -p blackscholes -i native -n 2
```

You can also inspect the yaml files for the PARSEC jobs from the previous parts to further understand their command lines. You can find the rest of the docker images here. Make sure to use the these image versions:

- splash2x.fft: `anakli/parsec:splash2x-fft-native-reduced`
- freqmine: `anakli/parsec:freqmine-native-reduced`
- ferret: `anakli/parsec:ferret-native-reduced`
- canneal: `anakli/parsec:canneal-native-reduced`
- dedup: `anakli/parsec:dedup-native-reduced`
- blackscholes: `anakli/parsec:blackscholes-native-reduced`

IMPORTANT: you must delete your cluster when you are not using it! Otherwise, you will easily use up all of your cloud credits! When you are ready to work on the project, you can easily re-launch the cluster with the instructions above.
To delete your cluster, use the command:

```
$ kops delete cluster part4.k8s.local --yes
```

4.1.2 Setting resource limits

`taskset` is an essential command for setting the CPU affinity of processes. For instance, running `taskset -a -cp 0-2 <pid>` will bind all threads (`-a` switch) of the running process indicated by `<pid>` (`-p` parameter) to the CPUs 0, 1 and 2 (`-c` parameter). One can also use this command when starting up processes. More information on `taskset` can be obtained here.

For Docker, the `--cpuset-cpus` parameter is used to set the cores a container is able to use. This parameter can be set both when spinning up a container (e.g. `sudo docker run --cpuset-cpus="0-2" ...`) or updated when a container is already running (e.g. `docker container update --cpuset-cpus="0-2" CONTAINER`).

You are also free to use other means to dynamically adjust resource allocation for your running jobs. This can refer to resources other than CPU cores.

4.2 Questions

The questions for Part 4 are stated in the project report template.

5 FAQ

- When running `kops create`:
    - if you get the following error: `failed to create file as already exists: gs://cca-eth-2022-group-XXX-ethzid/part1.k8s.local/config. error: error creating cluster: file already exists`, you need to delete the contents of your Google Cloud storage bucket, the recreate it with the following commands:

      ```
      $ gsutil rm -r gs://cca-eth-2022-group-XXX-ethzid/
      $ gsutil mb gs://cca-eth-2022-group-XXX-ethzid/
      ```

    - if you get the following error: `Error: error creating cluster: error writing Cluster "part1.k8s.local": error from acl provider "k8s.io/kops/acl/gce": error querying bucket "...": googleapi: Error 404: The requested project was not found., notFound`, make sure you have set the credentials correctly:

      ```
      $ gcloud auth application-default login
      ```

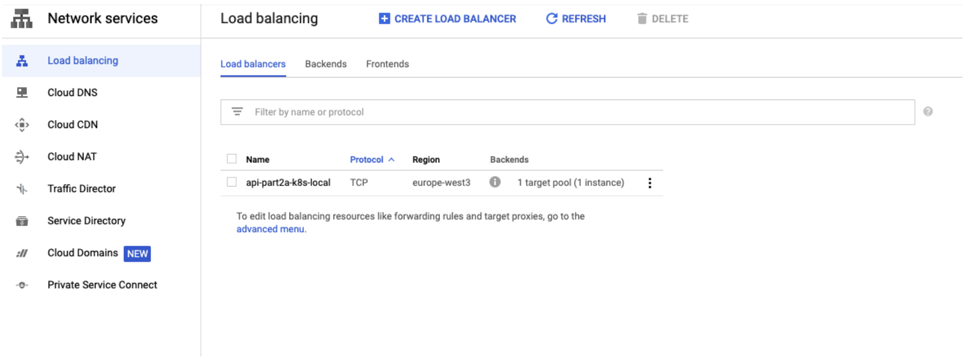
- When ssh-ing into a cluster node, if you get an error like

  ```
  WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
  ...
  Offending ED25519 key in /Users/username/.ssh/known_hosts:9
  ...
  Host key verification failed
  ```

  then you need to run `ssh-keygen -R <host>` where `<host>` is the IP address of the server you want to access.

- If `kubectl` commands prompt you for a username and password, delete the cluster and recreate it from scratch.

- If for any reason you cannot delete the cluster with the `kops` command do the following:
    - Go to `console.cloud.google.com`
    - Type in the search bar the term "Load balancers". You should be redirected to a page similar to the one below:

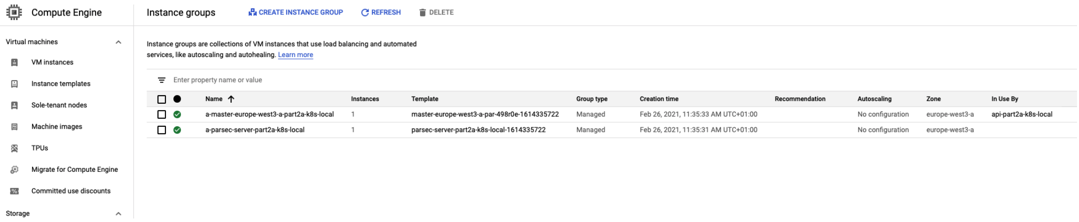


□ Name	Protocol ^	Region	Backends	
□ api-part2a-k8s-local	TCP	europe-west3	1 target pool (1 instance)	⋮

To edit load balancing resources like forwarding rules and target proxies, go to the advanced menu.

    - Select and delete the load balancer.

  - Then type in the search bar the term “Instance groups”. You should be redirected to a page similar to the one below:



Instance groups

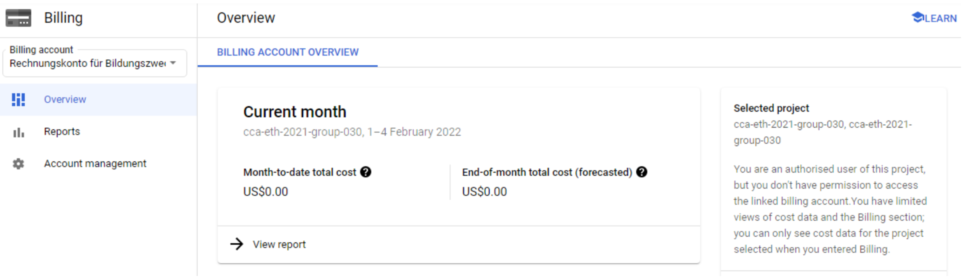
Instance groups are collections of VM instances that use load balancing and automated services, like autoscaling and autohealing. Learn more

□	●	Name ↑	Instances	Template	Group type	Creation time	Recommendation	Autoscaling	Zone	In Use By
□	✓	a-master-europe-west3-a-part2a-k8s-local	1	master-europe-west3-a-par-498r0e-1614335722	Managed	Feb 26, 2021, 11:35:33 AM UTC+01:00		No configuration	europe-west3-a	api-part2a-k8s-local
□	✓	a-parsec-server-part2a-k8s-local	1	parsec-server-part2a-k8s-local-1614335722	Managed	Feb 26, 2021, 11:35:31 AM UTC+01:00		No configuration	europe-west3-a	

  - Select and delete all the instance groups.
  - Delete your Google Cloud storage bucket by typing:
    `$ gsutil rm -r gs://cca-eth-2022-group-XXX-ethzid/`
  - Also under ”External IP addresses” check there are no charges for left over static IPs.

- If your Google Cloud Credits are disappearing even though no charges appear on your Billing Overview, make sure you have unselected ”Promotions”

  - Go to `console.cloud.google.com`
  - Type in the search bar the term “Account Overview”. Select ”Go to linked billing account” if prompted. You should be redirected to a page similar to the one below:



Overview

Billing account: Rechnungskonto für Bildungszwe‹

Current month

cca-eth-2021-group-030, 1–4 February 2022

Month-to-date total cost: US$0.00

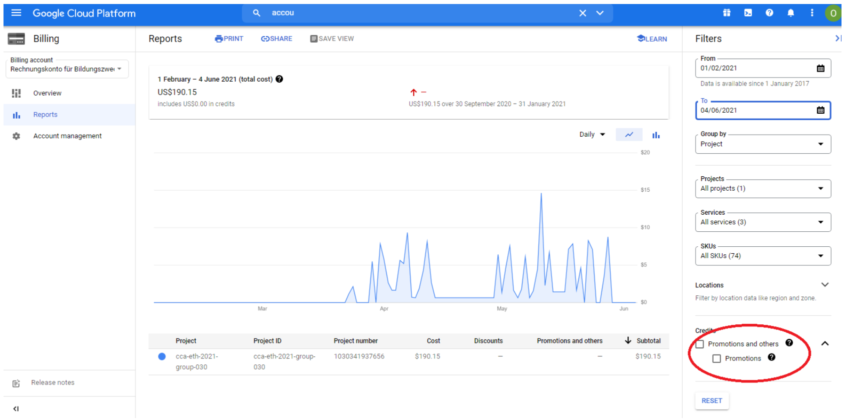
End-of-month total cost (forecasted): US$0.00

Selected project

cca-eth-2021-group-030, cca-eth-2021-group-030

You are an authorised user of this project, but you don't have permission to access the linked billing account.You have limited views of cost data and the Billing section; you can only see cost data for the project selected when you entered Billing.

  - Click on ”View report”.
  - Make sure you unclick ”Promotions and Other” as shown below and select a reasonable To/From time range:



Reports

Billing account: Rechnungskonto für Bildungszwe‹

1 February – 4 June 2021 (total cost)

US$190.15

includes US$0.00 in credits

↑ –

US$190.15 over 30 September 2020 – 31 January 2021

Project	Project ID	Project number	Cost	Discounts	Promotions and others	↓ Subtotal
cca-eth-2021-group-030	cca-eth-2021-group-030	1030341937656	$190.15	–	–	$190.15

From: 01/02/2021

Data is available since 1 January 2017

To: 04/06/2021

Group by: Project

Projects: All projects (1)

Services: All services (3)

SKUs: All SKUs (74)

□ Promotions and others

□ Promotions

- If you run out of credits for your project, please email cloud-arch-ta@lists.inf.ethz.ch to request additional cloud credits.