Analyzing Google Cluster Data with Spark

Yucheng XIAO

General

The code is in file navigation.scala in Navigation/src/main/scala folder. Scaka and RDD is used for this project.

File hierarchy is the following:

- MainFolder
  - clusterdata-2011-2
    - job_events, machine_attributes, etc.
  - Navigation
    - project
    - src
      - main
        - scala
          - navigation.scala
    - target

The code is carefully commented, with different parts for each question. All parts are individual and can execute on their own in main program. One can comment out the other part to verify a single question. The code proof for each statement is noted as "code fragment xxx" (will be shortened by "codefrag xxx" afterwards) in the file, and it will be green in this document.

To verify each question, uncomment the corresponding part in main() method.

Q1. What is the distribution of the machines according to their CPU capacity?

To check the code corresponding to this question, check the code pt1 part in the file.

First, we count the total number of machines in the list (identified by their unique machine ID): 12583 machines. codefrag 001

Then, we show the list of platforms available, and there's a total of 4: codefrag 002

- *GtXakjpd0CD41brK7k/27s3Eby3RpJKy7taB9S8UQRA=*
- *JQ1tVQBMHBAIISU1gUNXk2powhYumYA+4cB3KzU29l8=*
- *HofLGzk1Or/8lldj2+Lqv0UGGvY82NLoni8+J/Yy0RU=*
- *70ZOvysYGtB6j9MUHMPzA2Iy7GRzWeJTdX0YCLRKGVg=*

In the following paragraph I'll refer them as platform GtX, platform JQ1, platform Hof and platform 70Z. During the whole period, the number of machines for each platform is: codefrag 003

- *platform GtX: 798 Machines*
- *platform JQ1: 32 Machines*
- *platform Hof: 11659 Machines*
- *platform 70Z: 126 Machines*

Here we can do a simple calculation: 798 + 32 + 11659 + 126 = 12615 machines, which is 32 more than the actual number, surprisingly equal to the number of machines in platform JQ1. This led us to dive a bit deeper into the machines in JQ1 platform:

By executing codefrag 004, we found that all machines in JQ1 appeared as well in other platforms. Then we found that all the machines in JQ1 have no CPU resources at all. (The value is null, not 0) codefrag 005

Therefore, we can infer that platform JQ1 might be a temporary platform for machines, and the machines will transfer to other platforms.

The machines in platform GtX all has a CPU resource of 1.00 in its 1414 machine events. codefrag 006

The machines in platform Hof all has a CPU resource of 0.50 in its 19773 machine events. codefrag 007

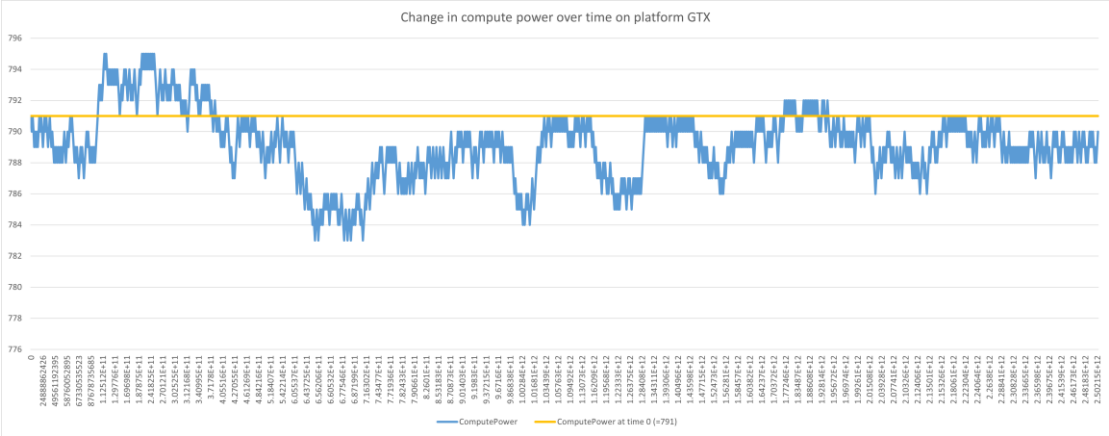
The machines in platform 70Z all has a CPU resource of 0.25 in its 224 machine events. codefrag 008

Q2. What is the percentage of computational power lost due to maintenance (a machine went offline and reconnected later)?

Since platform JQ1 has no CPU resource at all, we'll exclude it from the following context.

In the machine_event, there exists 3 types of events: ADD, REMOVE and UPDATE. The REMOVE events can occur by either maintenance or failures, but here we'll treat all REMOVE case as maintenance. Also, since all events in the same platform has the same CPU resources, we can ignore the UPDATE events as well, for it does not change the CPU resources.

We first focus on the platform GtX. The essential step is to get the compute power at each timestamp. Check codefrag 009 for the detailed step. Then we can have this image of computer power over time on platform GTX (please zoom in to get a clearer view):

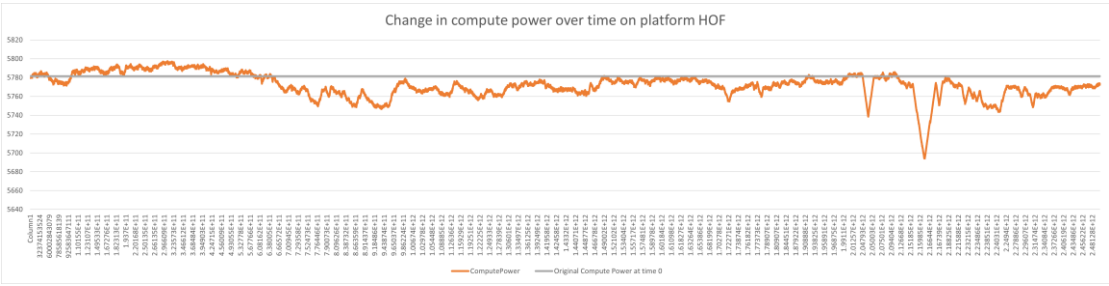


Here's some stats of compute power of platform GTX: codefrag 010

- initialComputePower of platform GTX (at timestamp 0) is 791.0
- Average compute power of platform GTX is 789.1522435897435
- Min compute power of platform GTX is 783.0
- Max compute power of platform GTX is 795.0

Note that in actual practice the average compute power should be calculated by integral, given that the timestamp can only represent the compute power at each time point. But for the sake of simplicity, I'll use the average (mean) value of all the computer power at each time point. We can get that we lost at most 0.779% of compute power compare to the average, 1.011% of compute power compare to the initial power.

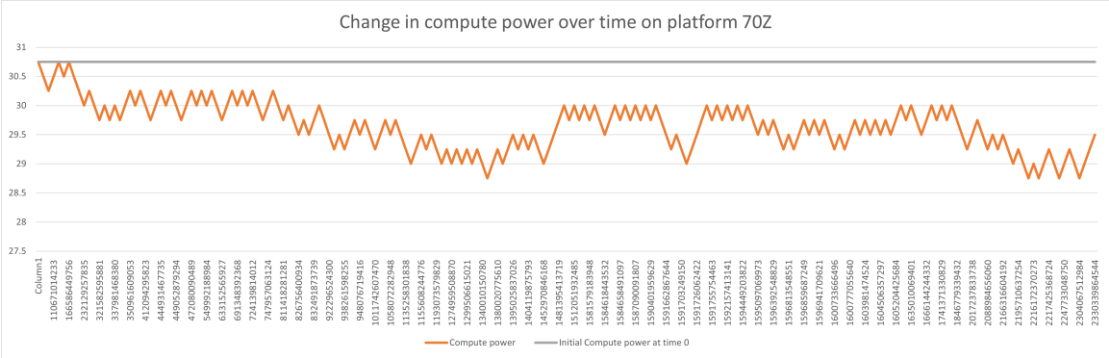
Then do the same stuff to platform HOF: codefrag 011



- initialComputePower of platform HOF (at timestamp 0) is 5781.5
- Average compute power of platform HOF is 5772.2092407835435
- Min compute power of platform HOF is 5694.0
- Max compute power of platform HOF is 5797.5 codefrag 012

We can get that we lost at most 1.355% of compute power compare to the average, 1.513% of compute power compare to the initial power.

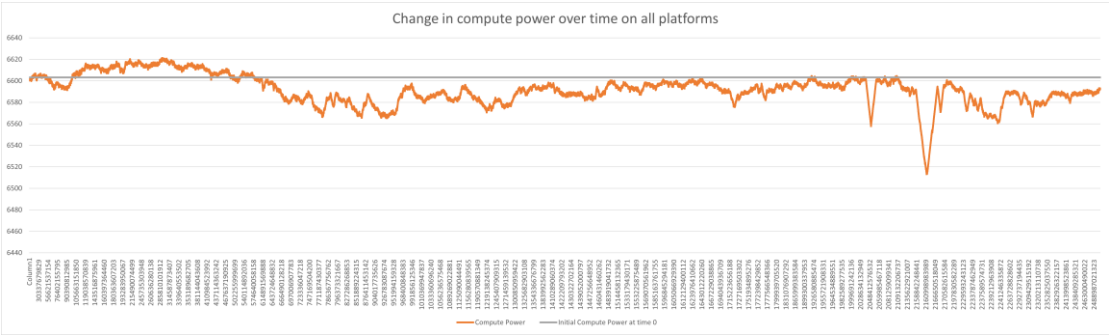
And do the same stuff to platform 70Z: codefrag 013



- platform 70Z initComputePower (at timestamp 0) is 30.75
- Average compute power of platform 70Z is 29.646634615384624
- Min compute power of platform 70Z is 28.75
- Max compute power of platform 70Z is 30.75 codefrag 014

We can get that we lost at most 3.024% of compute power compare to the average, 6.504% of compute power compare to the initial power.

Now we should consider all the platform together: codefrag 015



- initComputePower (at timestamp 0) is 6603.25
- Average compute power of all platforms is 6591.958459087837
- Min compute power of all platforms is 6513.5
- Max compute power of all platforms is 6621.25 codefrag 016

We can get that we lost at most 1.190% of compute power compare to the average, 1.359% of compute power compare to the initial power (surprisingly not that much).

Q3. What is the distribution of the number of jobs/tasks per scheduling class?

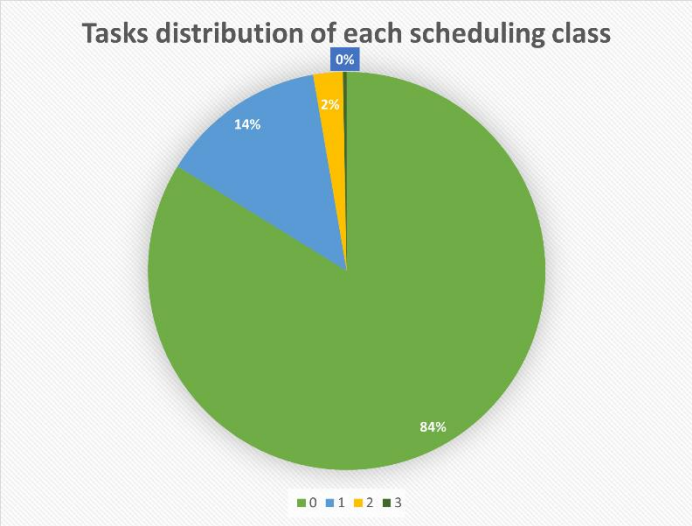
In the task_event, there is a total of 672,004 jobs with 25,424,731 tasks codefrag 017. However, when I tried to create distinct (jobID, taskIndex, scheduling class) tuple, it tells me there's 25,424,734 of them codefrag 017, which shows that a single task can have multiple scheduling class. Given that a single task of a job can run on multiple machines, this seems to be quite normal for a part of the

task can be latency sensitive and other parts are not that sensitive.

Given that there's 500 task_event files with each of them contains over 400,000 entries, this procedure will take quite some time. But in the end, we have the following data:

- scheduling class 0: 21,304,623 tasks,
- scheduling class 1: 3,431,474 tasks,
- scheduling class 2: 612,045 tasks,
- scheduling class 3: 76,592 tasks.

Adding the 4 numbers and we get 25,424,734 tasks, which matches the number above. We can create a graph about this data:



Q4. Do tasks with a low scheduling class have a higher probability of being evicted?

We start by doing some general analysis: there's a total of 144,648,288 task_events, and 5,864,353 of them are EVICT event. From the execution of codefrag 19, we have the following data:

- Scheduling class 0: 3.84% (4,400,303 EVICT event out of 114,598,888 events)
- Scheduling class 1: 6.21% (1,121,147 EVICT event out of 18,055,014 events)
- Scheduling class 2: 2.70% (300,910 EVICT event out of 11,158,598 events)
- Scheduling class 3: 5.02% (41,993 EVICT event out of 835,788 events)

From these data, we can infer that the scheduling class of a task have no direct influence on the probability of being evicted.

Q5. In general, do tasks from the same job run on the same machine?

The data to analysis are still in task_events files. For each distinct job, we count the number of machines used, and then count the number of jobs running on only one machine and the number of jobs running on multiple machines. This could give a view about the question.

During the programming, we noticed that a task can be run on multiple machines. Also, there are some tasks without a machine assigned, but it still has request some amount of computing resources (CPU, memory, etc.), which explains why the following stats on "Jobs with machine" (670,877 jobs) is slightly lower than the actual number of jobs (672,004 total).

From the execution of codefrag 20, we have the following data:

- Jobs with machine: 670,877
- Jobs running on one machine:491,996 (73.34%)
- Jobs running on multiple machines: 178,881 (26.66%)

Overall, more than 2/3 of jobs runs on the same machines, so we can infer that the scheduler tends to arrange the tasks from the same machine. This approach seems to be reasonable, considering most of parallel programs needs access to memory / storage that other tasks modified, and access them locally on one machine is way faster than fetching the data from machine-machine connections.

Q6. Are the tasks that request the more resources the one that consume the more resources?

For this question I'll refer resources to CPU resources. First, we need to define how much resource can be called as "more resources". Therefore, we need to figure out the max, min, mean and median CPU resource request for all tasks.

Run codefrag 021 and we have the following results:

- Min CPU resource request is 0.0
- Max CPU resource request is 0.5
- Avg CPU resource request is 0.05177356236213231
- Median CPU resource request is 0.02737

According to those stats, it's a reasonable decision to set the 75% median CPU resource request as the indicator for more / less resources.

- 75% Median CPU resource request is 0.06055

So, in the following tests, the tasks will be split into 2 parts: those tasks requests less than 0.06 CPU resources, and those requests more.

Then we turn our eyes into the resource usage in task_usage files by doing some analysis and create a CPU usage table with format ((jobID, taskIndice), average CPU usage): codefrag 022

- Min CPU mean usage is 0.0
- Max CPU mean usage is 0.5520106666666666
- Avg CPU mean usage is 0.011860686410295005

Due to the large amount of data (and limited amount of memory), calculating 75% median will take ages, so

At the beginning of codefrag 023, we count that there are 31,882,167 tasks requires more resources and 112,765,830 tasks requires less, which is around 1:4, not far from the 1:3 ratio as expected.

In codefrag 023, 2 lists of tuples (jobID, taskIndex) are created for those requires more CPU and those requires less. By joining the list with the CPU usage table created by codefrag 022, we can get the mean CPU usage rate for high resource request tasks: codefrag 024

- Mean CPU usage rate for high res request tasks is:0.01475
- Mean CPU usage rate for low res request tasks is: 0.00808

Comparing to the CPU usage of low resource request tasks, the CPU usage rate for high resource request task is significantly higher, even though it's only slightly above the average CPU mean usage (0.012) calculated by codefrag 022. From all the data above, we can infer that tasks which request the more resources does consume more CPU resources than those tasks which requires less, but most of the high CPU requested tasks don't even use a quarter of CPU resources is requires.

Q7. Can we observe correlations between peaks of high resource consumption on some machines and task eviction events?

To analyze this problem, first we need to define the term "high resource consumption". Since we're trying to find the relation between task eviction event, using a high mean CPU usage rate will not be a sensible idea. For this problem we have to look at maximum CPU usage in task_usage files. This indicates the sudden burst of CPU better than the mean CPU usage rate.

The main idea of the analysis is the following:

- Create a list of EVICT tasks, with corresponding machineID and timestamp (the list will be called evictList in the following paragraphs); codefrag 027
- Create a RDD which contains the start & end measure time, machineID and CPU max usage, filter the CPU max usage so all entries are above the set threshold. codefrag 028
- Foreach high CPU resource consumption entry:
  - Find whether the machine ID is in evictList or not,
  - Check if the timestamp is inside the entry start & end time

        - If both passes, then we reckon that this high CPU resource consumption occurs when a EVICT event happens, add 1 to variable “hit” codefrag 029
    - Compare the variable *hit* with *high_res_comp* and get a conclusion.

Then we have to define how high the threshold will be for “high” consumption. Given that both task_event and task_usage are extremely large files contains countless entries, I decide to limit the top 500 entries with highest max CPU usage for task_usage files. This means that any entries in task_usage with values less than 27.5 will be ignored codefrag 026. Even so the program takes more than 6 hours to finish due to having multiple filters to a large RDD.

- Here’s the result: codefrag 027 028 029
    - High res comp count: 501
    - Hit count: 64

So only 12.77% of high resource consumption corresponds to an EVICT event, which we can infer that EVICT event does not have a major influence on high resource consumption.

Q8. Relationship between job scheduling delay and their corresponding scheduling class?

There are a total of 2,012,242 job events. Each job can have multiple entries for different type of events (SUBMIT, SCHEDULE, EVICT, etc.). Here we focus on the time difference between SUBMIT and SCHEDULE event.

The average delay for jobs is: (unit μs) codefrag 031

- Average schedule delay: 8,131,010.17
- Schedule class 0 average delay: 4,748,997.97
- Schedule class 1 average delay: 4,271,158.21
- Schedule class 2 average delay: 16,951,990.52
- Schedule class 3 average delay: 5,158,415.73

However, these data include a large number of jobs that is already running at time 0 (meaning that these tasks are scheduled before logging started). So, to better analyze the picture, we have to take these tasks out of context.

Then we calculate the average without the jobs already scheduled at time 0: (unit μs) codefrag 031

- Average schedule delay: 8,263,881.10
- Schedule class 0 average delay: 4,827,696.33
- Schedule class 1 average delay: 4,301,573.92
- Schedule class 2 average delay: 17,146,094.25
- Schedule class 3 average delay: 11,671,817.42

By these data we can conclude that the scheduling class level has no significant

effect on scheduling delay. With an average schedule delay of around 8 seconds, google cloud seem to be quite a responsive platform :)

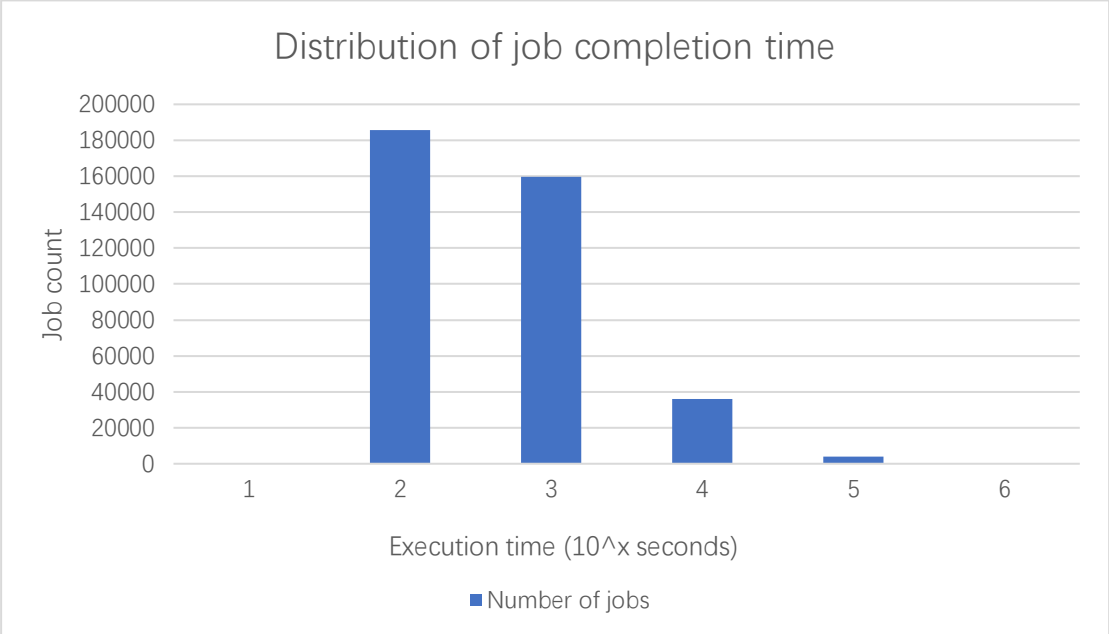
Q9. Average completion time of jobs and their distribution?

To solve this question, we have to find what is the completion time of a job. From Job_Event table, we find the corresponding timestamp for a SCHEDULE event and a FINISH event and then calculate the time difference, the time difference can be defined as job completion time. Note that SCHEDULE event does not mean that the job starts running, so this is merely an estimation of completion time. We start by getting the average completion time to get a global image of the question. codefrag 032

- Job average completion time: 6.597292431424966E8 (unit μs)

This shows that the average completion time is around 660 seconds, which is around 11 minutes. Then we dive deeper into the distribution of job execution time by counting them by log(exectionTime) codefrag 033

- (0,2) 0.001%
- (1,185750) 48.174%
- (2,159670) 41.410%
- (3,36138) 9.372%
- (4,3947) 1.024%
- (5,76) 0.020%
- Total 385,583 100.000%



From the figure above, we can conclude that most of the jobs (around 90% of them) finishes between 10-1000 seconds. Also comparing to total number of jobs (672,004 total),