Final project : Analyzing Data with Spark

Maya Costantini, M2 MSIAM - Data Science

January 20, 2021

Abstract

The goal of this project is to analyze a dataset provided by Google analyzing the activity of a Google compute cell, i.e a set of machines during a period of 29 days. Due to the size of this dataset and its availability on Google Cloud, we will provide some analysis on the cell such as information on the computing capacities, on the task priorities or on the consumption of resources using Pyspark, the Python API made to write Python code which uses the Apache Spark framework. In order to retrieve and interpret useful information in the dataset named `clusterdata-2011-2`, the work of this project will be based first on some suggested analysis in the part 5 of the assignment, and will then be extended by reviewing an article titled *Borg: the Next Generation* [1], comparing the 2011 and 2019 versions of Google's cluster management system. In the following, this document will mainly focus on explanations of the approach to solve the questions and on the presentation of the results. The Python code will be provided in Appendix as well as on the compressed files sent with the report.

1 Presentation of the dataset

First, in order to get a clear overview of the data to process, let's provide a short description of the dataset. The name of the different folders containing the information we might need can be displayed using the `gsutil ls gs://clusterdata-2011-2/` command, which gives :

```
gs://clusterdata-2011-2/MD5SUM
gs://clusterdata-2011-2/README
gs://clusterdata-2011-2/SHA1SUM
gs://clusterdata-2011-2/SHA256SUM
gs://clusterdata-2011-2/schema.csv
gs://clusterdata-2011-2/job_events/
gs://clusterdata-2011-2/machine_attributes/
gs://clusterdata-2011-2/machine_events/
gs://clusterdata-2011-2/task_constraints/
gs://clusterdata-2011-2/task_events/
gs://clusterdata-2011-2/task_usage/
```

The size of the dataset being too consequent to be analyzed in its entirety (41 GiB), we will focus the analysis on the first csv file for each table which already contains an important quantity of information.

The `schema.csv` file contains useful general information about the variables used in the csv files of the dataset. The `job_events` and `task_events` tables principally give information to identify jobs and tasks composing the jobs, the *scheduling classes* they belong to (i.e. the code attributed to each job/task by the managing system to distribute them over the machines and to decide whether or not if it should be submitted, executed, failed...) and some variables to evaluate their need for resources in terms of CPU cores, RAM or disk storage.

2 First analyses to be conducted

In this part, we are going to answer the questions of the part 5.2 of the assignment by providing a first analysis of our dataset using Pyspark and the Google documentation describing the data.
In order to be able to locally read and write in the files of the dataset located in the `clusterdata-2011-2`, an approach is to download small parts of the necessary csv files containing the data in order to conduct the analysis.

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

The necessary information to answer this question can be found in the `machine_events` file of the dataset, which contains the `part-00000-of-00001.csv` file, that we download from the dataset bucket by executing `gsutil cp gs://clusterdata-2011-2/machine_events/part-00000-of-00001.csv.gz ./code` in the shell. The "machine events" part of the Google documentation (p. 4) describes the six variables used in the `part-00000-of-00001.csv` file. The values describing the capacity in terms of number of CPUs is located at the fifth column of the file.

By creating a RDD containing the values of the CPU capacity column, it is possible to plot the histogram showing the number of machines having a CPU capacity of 1, 0.5 or 0.25 (neglecting the 32 non available data) :

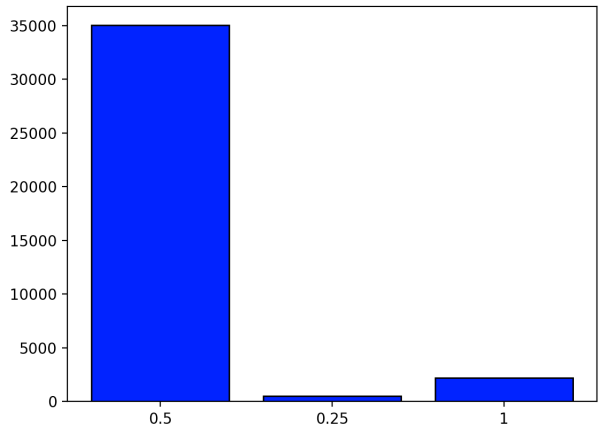

Figure 1: Histogram of the CPU capacities of the machines

As precised in the assignment, the CPU capacity of each machine is a value normalized between 0 and 1, where 1 represents the maximum capacity over all the machines and all the other values are computed according to this maximum capacity (for example, a capacity equal to 0.5 means that the machine has half the capacity of the machine with the maximum number of CPUs). Here, a large majority of machines (approximately 35000) have a 0.5 CPU capacity, whereas a smaller number has a maximum capacity equal to 1 or a 0.25 capacity.

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

We look at the `machine_events` table in order to compute the respective rates of CPU and memory capacities lost when an event of type REMOVE (code 1) occurs, which means that the machine was removed from the cluster due to maintenance (Google documentation p. 4).
After proceeding to the appropriate computation on the relevant variables in the table, we find that the CPU capacity percentage that was lost due to a REMOVE event was of 24% and of 24%,5 for the memory capacity percentage.

2.3 On average, how many tasks compose a job?

We look into the `task_events` table of the dataset, and the volume of the data being too important (500 csv files with more than 450000 rows each), we download only the first csv file by executing the `gsutil cp gs://clusterdata-2011-2/task_events/part-00000-of-00500.csv.gz ../data` command. Each line of the file contains thirteen variables describing a task, including the job it belongs to identified by an ID at the third column. By counting the number of times an ID appears in the table, we will then be able to approximate the average number of tasks that compose a job. The execution of the code gives that a job is composed on average of almost 92 tasks.

2.4 What can you say about the relation between the scheduling class of a job, the scheduling class of its tasks, and their priority?

Once again, we use the `part-00000-of-00500.csv` of the `task_events` table as well as the `job_events` table (once again, we only download one csv file for this table). We are interested in the information contained in the 'scheduling classes' columns (sixth column) of both the `task_events` and `job_events` tables, as well as in the 'priority' column of the `task_events` column (ninth column). We are going to analyze the data in those columns and compare them with the description of the event codes described at p. 6 of the Google Documentation.

Let's first examine the relationship between the most frequent scheduling class of tasks composing a job and of the scheduling class of the job. To this aim, we create two RDDs, one mapping the job ID with its scheduling class in the `job_events` table, and one mapping the job ID of a task to its scheduling class in the `task_events` table and reducing it by key by choosing the most common scheduling class of all the tasks composing the job. Then, we also reduce the union of the two RDDs by key, with a function in option that verifies if the value of the most common scheduling class among the tasks of a job corresponds to the scheduling class of a job. This reduce operation matches the job ID with the corresponding scheduling class in case the condition is verified, and matches it with -1 otherwise. The execution of the code returns the following result :

```
('2', <pyspark.resultiterable.ResultIterable object at 0x11b2944d0>)
('3', <pyspark.resultiterable.ResultIterable object at 0x11b294210>)
('1', <pyspark.resultiterable.ResultIterable object at 0x11b2c88d0>)
('0', <pyspark.resultiterable.ResultIterable object at 0x11b2ea710>)
```

which means that all of the scheduling classes of the jobs correspond to the most frequent scheduling class among the tasks that composes it (the scheduling classes here are either 0,1,2 or 3).

Now, let's get interested in the relationship between the scheduling class of a task and its priority. The 'priority' variable of a task is located at the ninth column of the `task_events` table, and assigns an integer between 0 (lowest priority) and 11 (highest priority) to the task. We are going to look at the most frequent priorities attributed to each scheduling class. For each scheduling class, we plot the histogram representing the distribution of the priority codes of the tasks :

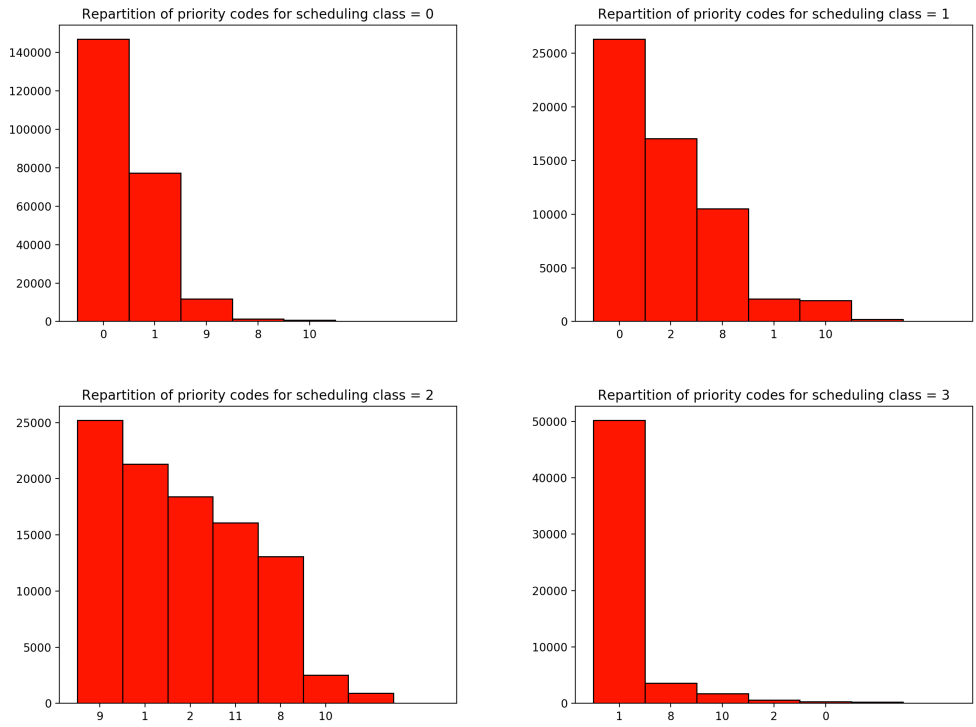

Figure 2: Priority codes distribution for each scheduling class of the tasks

In the figure above, we can see that the tasks having a scheduling class encoded by 0 have rather low priorities in general, with most frequent priority codes equal to 0 and 1. In the documentation, the scheduling class 0 corresponds to the SUBMIT event, which means that the task has become eligible for scheduling. For the scheduling class with the code 1, corresponding to a task that has been scheduled on a machine (SCHEDULE event), the most frequent priority codes are 0, 2 and 8, which shows that those tasks are either of low priority or of quite high priority. Quite surprisingly, for tasks of scheduling class 2 (EVICT, which means that the task has been descheduled to prioritize a more important task or because of a machine-related issue), the most frequent priority code for the tasks is 9. This could be explained by the fact that as indicated at the bottom of p. 6 of the Google Documentation, if the task just had been evicted it is more likely to be submitted immediately again. It is followed closely by codes 1 and 2, meaning that the evicted tasks are not systematically resubmitted right after their eviction. Finally for FAILed tasks (scheduling class 3), the most frequent priority is 1, which corresponds to a low priority certainly due to the failure of the task itself and not to an exterior event.

2.5 Do tasks with low priority have a higher probability of being evicted?

Let's examine the scheduling classes for tasks with low priority in the `task_events` table. To simplify our reasoning, we first consider the tasks with priorities inferior or equal to 3 by filtering the mapped pairs ('priority code of the task', 'scheduling class of the task') according to this condition. The histogram obtained by plotting the values of the pair shows that the dominant scheduling class among tasks with priority inferior to 3 is the EVICTED class (code 2) :

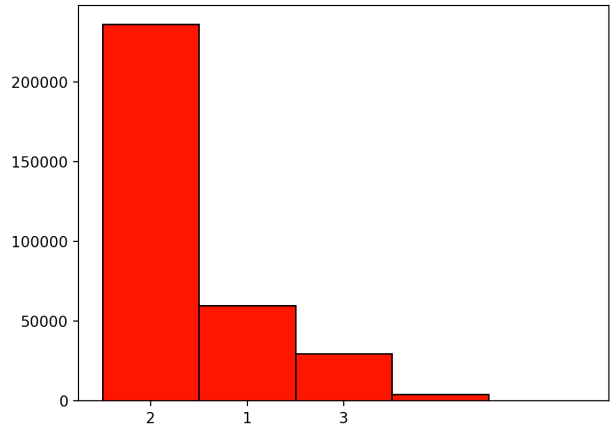

Figure 3: Distribution of the scheduling class codes for low priority tasks (inferior or equal to 3)

Now let's see if this extends for classes of higher priority, and observe if the EVICTED scheduled class is effectively more frequent among low priority classes :

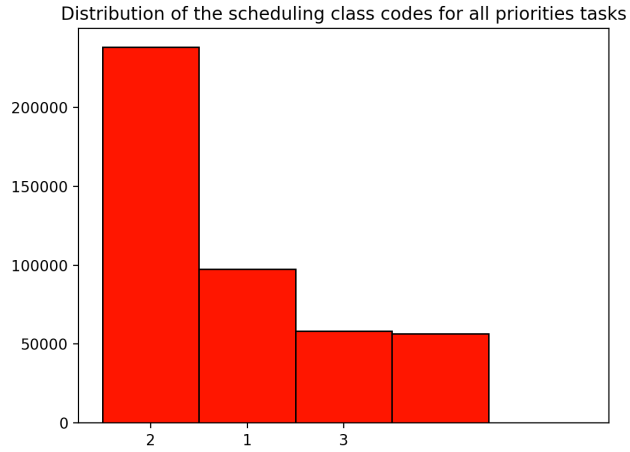

Figure 4: Distribution of the scheduling class codes for all priorities tasks

The plot shows that the scheduling classes for tasks of all priorities are distributed in a similar way as for tasks with a low priority, with a lot of them being evicted, scheduled or failing.

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

We focus on the `task_events` table in order to study the distribution of the jobs over the different machines, and more precisely to see if it is frequent for the management system to split several tasks of a job over different machines either than grouping them in the same one. The histogram above shows the number of machines on which the different tasks of a job have been run :

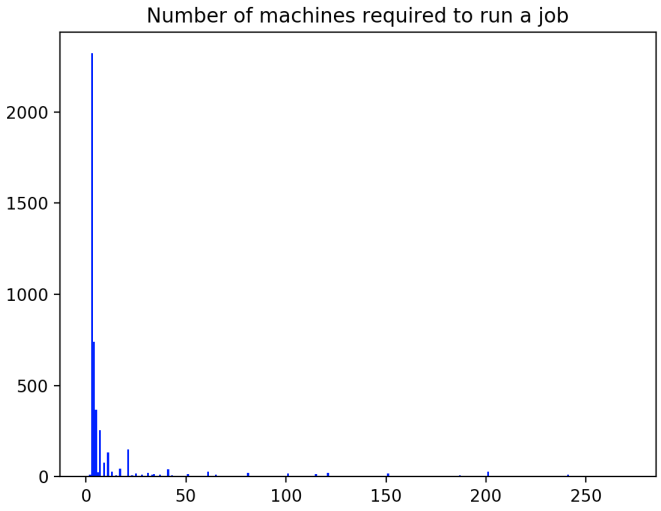

Figure 5: Number of machines on which the tasks of a job have been run

It seems that the tasks constituting a same job are often run on different machines, the most frequent number of machines being of 2. The tasks usually seem to be distributed over a relatively small number of machines (inferior to 50) and don't exceed 250 machines.

2.7 Are the tasks that request the more resources the one that consume the more resources?

The information about the resources requested by tasks can be found in the `task_events` table (columns 10, 11 and 12, with resources requests respectively for CPU cores, RAM and local disk space), whereas information about the actual consumption of resources is located in the `task_usage` table. We evaluate the tenth column of the `task_events` file which gives the resource request for CPU cores for each task. As we only dispose from data about the consumption of CPU cores resources on both tables, we will use them to evaluate the more general relation between need and consumption of computing resources for each task. In order to identify the tasks included in both files, we put as a key in the RDD the string `'machine ID, task ID within the job'`. After creating a RDD with values corresponding to `('machine ID, task ID within the job', ['mean CPU request', 'mean CPU consumption'])` and filtering the data to remove a few incorrect entries, we get the following graph showing the CPU resources requests versus the sample CPU usage (column 20 of `task_usage` file) :

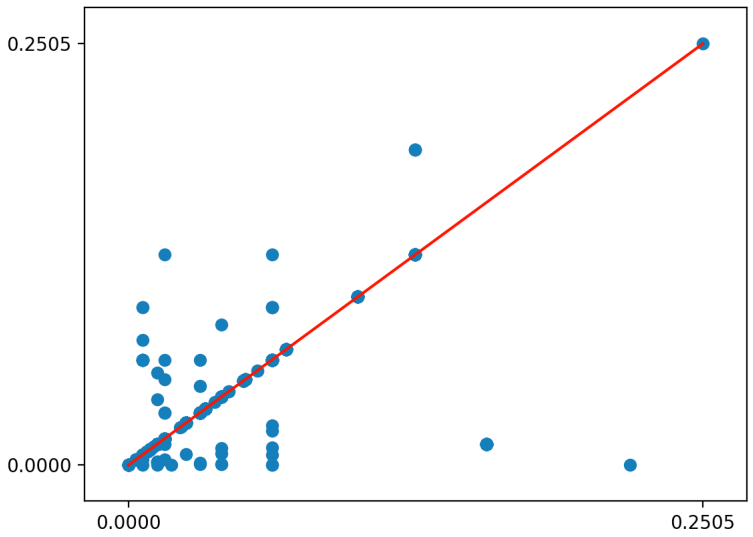

Figure 6: Tasks request for CPU resources versus actual sampled CPU consumption

The red curve plotted on the graph corresponding of a line of equation $y = x$ show that in general, the consumption of CPU resources by a task seems to be proportional to the CPU resources originally requested by this task.

2.8 Is there a relation between the amount of resource consumed by tasks and their priority?

To provide an answer to this question, we choose to compute the cumulative CPU resources consumed for each priority code present in the `task_events` table, which can be summarized in the following barplot :

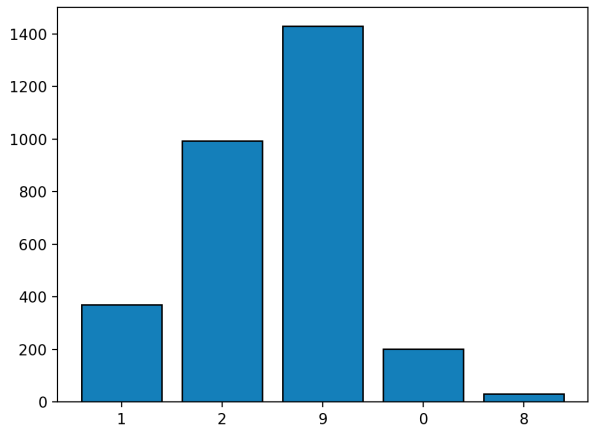

Figure 7: Cumulative CPU resources (in CPU cores/s) consumed for each priority code in the data set

The tasks that required the most CPU resources are the ones with the highest priority code within the data set (9) with approximately 1400 CPU cores/s necessary to their execution in total. Then come the codes 2 and 1, which consumed respectively in total approx. 1000 CPU cores /s and 400 CPU cores/s, surpassing the tasks with priorities 0 (approx. 200 CPU cores/s) and 8. This plot shows that the consumption of CPU resources might be higher for high priority tasks (as stated in p.9 of the Google Documentation), but that lower priority ones can eventually consume a more consequent amount of resources than expected.

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

We are interested in the `machine_events` and `task_usage` tables. In order to examine the connection between high CPU resources consumption and some events, we plot the histograms of the events code appearing in three different percentiles (99th, 90th and 50th) of CPU consumption :

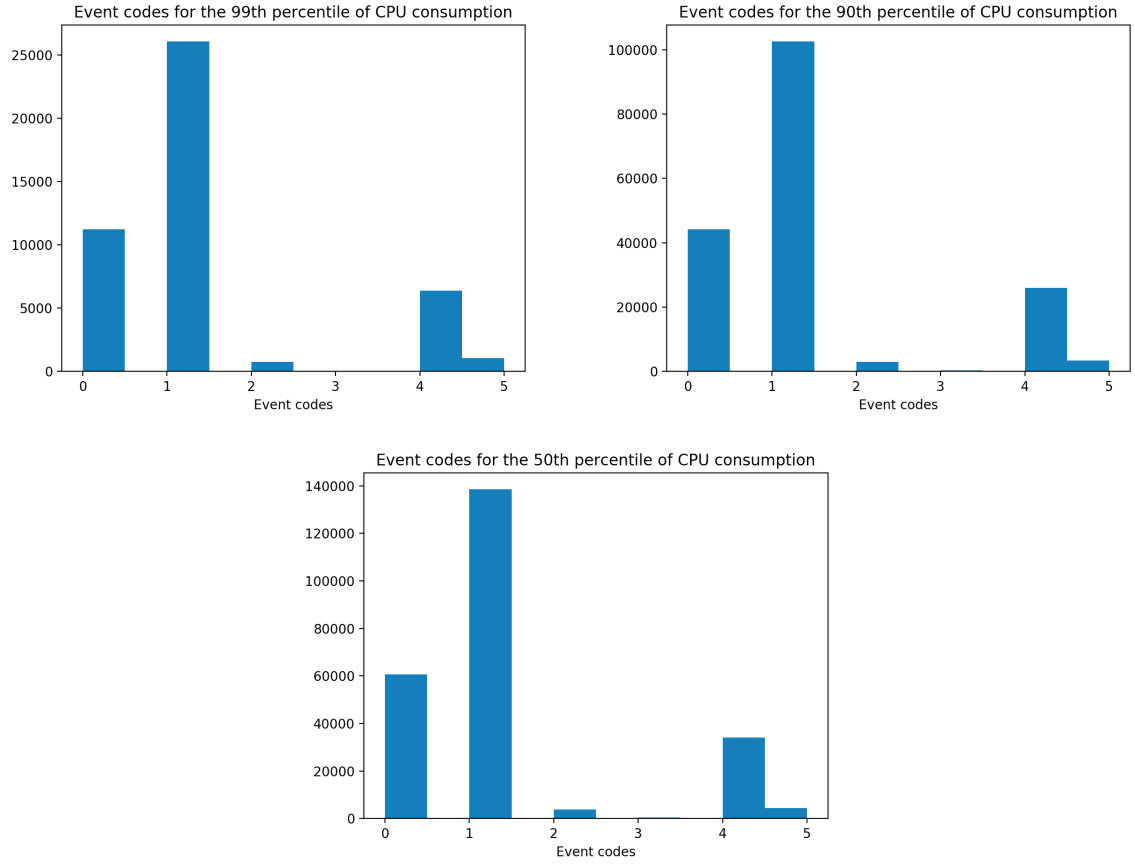

Figure 8: Distribution of event codes for the 99^{th}, 90^{th} and 50^{th} percentiles of CPU resources consumption

As we can see in the histogram, it seems that the distribution of the event codes doesn't change proportionally to the percentile of CPU resources consumption that we choose. In particular, it seems that EVICT events are not correlated with a high peak of CPU resources consumption.

3 Extension of the work : a review of *Borg: the Next Generation*

3.1 Presentation of the article

This article written in 2019 follows the analysis of the 2011 Borg cluster that we studied during this project. Borg is the cluster management system used by Google in its data centers that distributes the workload over the different units. It is composed of a central cluster scheduler and of several local management agents that each run a different node, such a unit being called a cell. The goal is to investigate on the evolution of Borg during those eight years by comparing its current properties and features with those in use back to 2011. The trace published by Google in 2019 is a record of the activity of eight different Borg clusters (versus one cluster for the trace published in 2011) for the month of May 2019. The article is mainly organized in five sections that analyze the new features of the 2019 trace, the changes in the distribution of the workload, the scheduling rate, the new distribution of job sizes and the efficiency of vertical autoscaling.

3.2 Main results and conclusions

Globally, the 2019 Borg system works in a similar way than its 2011 version, where the clusters receive job creation demands generated by users, analyze the meta-information that comes with the executables like the processor / memory resources needed or the number of tasks the job should be divided into and appropriately distributes those tasks between the running machines. In order to facilitate the scheduling of the different tasks by the management system, a priority code is attributed to each one of them; as we saw in the first part of this project, the priority codes for the 2011 Borg system went from 0 (lowest priority) to 11 (highest priority) whereas they go from 0 to 450 for the 2019 version of Borg. This increment in the number of priority codes allows to classify the tasks into more precise scheduling groups that we call "tiers", which go from the "Free tier" for the lowest priority tasks to the "Production tier" and "Monitoring tier" for the most urgent tasks to execute.

Moreover, other differences between the 2011 and the 2019 versions of Borg have been described in the article, among which we find the greater CPU versus memory capacity that the 2019 version acquired compared to the older version. The bubble chart (Figure 1, p.3) of the article illustrates how the new version has diversified its CPU to memory ratio. Thanks to the `machine_events` table, it is possible to confirm that change by plotting the corresponding bubble chart for the values of the 2011 version :

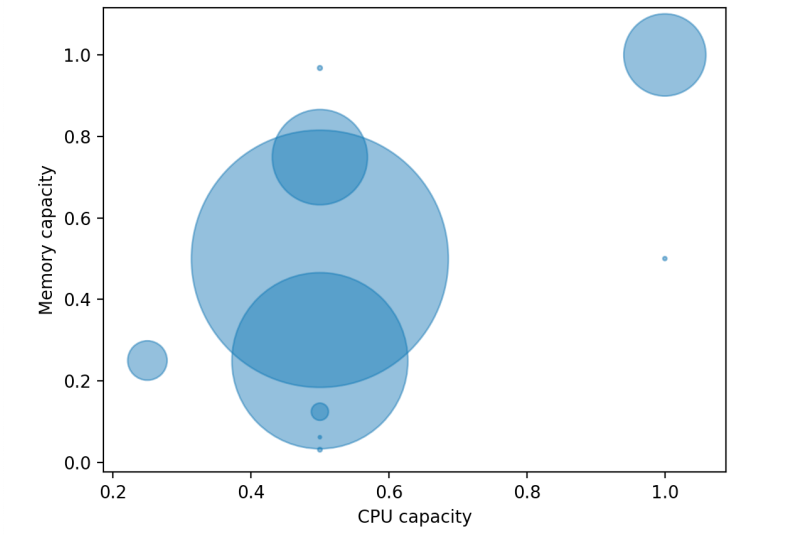

Figure 9: CPU vs. Memory ratio in the 2011 Borg trace

In comparison, the chart for the 2019 version looks more homogeneous and the CPU capacities look in particular more diverse and less centered in 0.5 (the values for both capacities and charts have been normalized between 0 and 1). The diversification of the CPU to memory ratio in the new version might enable to find more appropriate machines to run particular tasks that are not executed in an optimal way in machines with a more common ratio. Therefore, an interesting concept developed by Google in the 2019 version of Borg is the GCU (Google Compute Unit), which corresponds to an abstract unit representing a same amount of computational power in the system, allowing the user to handle the increased heterogeneity of the computational capacities of the machines. This unit with be used throughout the article normalized between 0 and 1 and renamed NCU for Normalized Compute Unit.

Contrarily to the 2019 trace, the 2011 trace omitted information about alloc sets, which allow the user to reserve some computing resources in advance for tasks or jobs scheduled later (the trace was published before the article [2] describing the use of those alloc sets was). The new system allows to treat this information and so to have a greater flexibility in the scheduling of tasks by mapping those into interpreted alloc instances. The new trace also takes into account parent-child relations between jobs, allowing further investigation on failure events, and considers the role of new batch schedulers which pre-process the data into a queue before passing it to Borg. Another useful feature of the 2019 Borg version is the handling of vertical autoscaling of workloads via the *Autopilot* system, which is able to evaluate the amount of computing resources to allow to each task. This constitutes a considerable improvement compared to the older version because it allows to operate tasks by requesting an optimal amount of resources while avoiding failures due to an excess of requested resources or tasks accumulation phenomena caused by a smaller amount of resources originally requested [3].

The Autopilot system was introduced in order to automatically distribute the computing resources among the tasks, to avoid the user to enter directly the requested amount of resources. In the 2019 trace, the system can be configured in three different versions which are *not autoscaled*, *fully autoscaled* or *autoscaled with constraints*. The study of the quantity $\frac{\max(0, \text{allocated NCU} \quad \text{max NCU})}{\text{allocated NCU}}$ corresponding to the *peak NCU slack* (definition p.11, Section 8) has shown that all the Autopilot configurations are globally more efficient in distributing computing resources than manual configurations, and that the *fully autoscaled* configuration is the most efficient in reducing this quantity.

The Figure 3, p.4 of the article clearly illustrates the global increase of the average amount of resources used as a fraction of a cell capacity for the eight cells of the 2019 trace compared to the cell of the 2011 trace. Indeed, the amount used in the 2011 trace never exceeds 40% of the total cell capacity, whereas this value is surpassed for all eight cells of the 2019 trace up to approximately 90% for the cell h. This barplot constitutes the proof that the strategies and tools developed to make the new Borg system more flexible in terms of data management and scheduling have been successful and now allow to operate a greater part of the available computational power. The allocation of computing resources as shown in Figure 5, p.5 also show a significant increase between the 2011 version and most of the cells of the 2019 version, especially for the production tier which manages the tasks with the highest priority codes.

In both versions of Borg, we can notice that the allocation of CPU and memory resources often surpass the actual limit defined for each of them (Figure 4, p.5). This voluntary excess in the allocation of resources is caused by the choice to use *statistical multiplexing*, i.e. the sharing of a common resource for several tasks considering that each task will has a significant probability to use less computing power than requested. In the 2019 trace, the over-commitment of tasks has greatly increased for CPU allocation, sometimes allocating 160% of the maximal CPU limit to tasks, and even more for the memory allocation compared to the 2011 trace which goes up to approximately 170% for the new version. Thus, the conclusion of the results presented in the Section 4 of the article dealing with resource utilization show that the actual progress made by the 2019 version of Borg is rather related to the internal distribution of the workload on machines of fixed capacities than to an actual extension of the fraction of resources used to process the tasks.

The large amount of failure codes found in the 2011 trace is mainly explained by the fact that the parent-children relations between some jobs are not taken into account in the old Borg version. What some articles studying the 2011 trace call "failures" in a general manner are in fact distinct types of events such as evictions, kills, fails or finish events when a task has been run successfully, with some of those events actually triggered by users, and not by (rare) failures of the hardware or the system. Being able to trace the parent-children relations between jobs helps explain the high "failure" rate of tasks in the 2011 trace, caused by a dependency of the children tasks to the parent tasks which thus accumulate twice a larger number of "kill" events than jobs with no relations. The jobs and tasks submission rates have also been multiplied by more than three between the 2011 and 2019 system versions, which lead the authors to investigate on the time the management system takes in its new version to distribute and schedule those. In particular, they chose to study the time the first task of a job positioned for running to be executed and found out that the median scheduling delays actually decreased, and that the tiers that are the longest to be executed are those which contain the more tasks.

Resource consumption under the management of the new Borg version has also been studied in-depth in the article. In particular, we are interested in the variability of the resources consumed by unit of time in the system that is measured in NCU-hours and NMU-hours (for Normalized Memory Unit), going back to more abstract units. The coefficient $C^2 = \frac{\text{variance}}{\text{mean}^2}$ is particularly convenient to study this variability knowing that it is not sensible to a re-scaling of the data. In the current Borg version, this coefficient is equal to 23000 for CPU and to 43000 for memory, which is considered extremely high if compared to similar measurements made in the late 1990's and early 2000's in supercomputing centers. It is interesting to notice that the CPU and memory resources consumption per hour follow a Pareto law, but with an even more extreme distribution than as stated in the Pareto principle (80% of the load comes from 20% of the tasks), with approximately 1% of the jobs constituting 99% of the CPU load and 93% of the memory load. The results are similar for the 2011 trace, but with a less heavy-tailed distribution of the load among jobs.

But what exactly are the implications of the value of the coefficient C^2? The Section 7.3 informs that this coefficient is actually proportional to the mean queueing delay for job resource demand on a *M/G/1 queue*, i.e. a Markovian queue (regulated by a Poisson process) with a General distribution of service times and a single server [4], which is illustrated here by the Pollaczek-Kinchin formula (where ρ is the load) :

$$\mathbb{E}[\text{queueing delay}] = \frac{\rho}{1-\rho}\frac{C^2+1}{2}$$

This formula translates in the fact that a high C^2 coefficient will lead to high queueing delays due to the greater diversity of jobs sizes, that will "stuck" the smaller and faster to execute part of the jobs (99% of the jobs) behind the 1% part of the jobs requiring the more resources. Therefore, an improvement that could me made for Google's Borg scheduling system is to make sure that the jobs which that are the least demanding in terms of both resources are run first, in order to let the heaviest jobs be computed after (the most demanding jobs in terms of CPU capacity are often the most demanding in terms of memory too).

Finally, this first study of the 2019 Borg trace has highlighted the progress made in a span of eight years in the rapidity and efficiency of the scheduling of tasks and jobs and of their running delay compared to the 2011 trace. A number of new features in the current system have participated in those improvements, such as the diversification of the processors and memory machine capacities, of the distribution of jobs in a larger number of categories, a more aggressive statistical multiplexing for the allocation of resources, as well as more "user-friendly" features such as the handling of vertical autoscaling and the consideration of parent-children relations between jobs, giving more insight about the reasons of job failures. The results presented using normalized, more universal units have enabled to make precise comparisons between the old and new versions of Borg, enhancing both the progress that was made in terms of delays and scheduling but also pointing out eventual improvements that could be made in a future version of the system, such as a prioritisation of the 99% of jobs with small resource requests compared to the 1% more heavy to process jobs. Other eventual paths to follow for a further analysis have been proposed by the authors, like the in-depth study of some phenomena linked to scheduling, to the limits of over-commitment or to inter-cell variation.

A Code for the Part 2

```
import time
import pyspark
from pyspark import SparkContext
import matplotlib.pyplot as plt
from collections import Counter
from random import sample
import numpy as np

### Some parts of the code are inspired by the code provided for the first lab on Spark


### Setting a SparkContext and the adjusting the verbosity of the error messages :
sc = SparkContext('local[1]')
sc.setLogLevel("ERROR")
```

A.1 What is the distribution of the machines according to their CPU capacity?

```
#Creation of the RDD and keeping the RDD in memory :
machine_events = sc.textFile('../data/part-00000-of-00001.csv')
machine_events.cache()

machine_events_index = 4
machine_events_entries = machine_events.map(lambda x: x.split(','))

# RDD of ('CPU capacity', 'number of machines with this CPU capacity') :
CPU_machines_distribution = machine_events_entries.map(lambda x: (x[machine_events_index],1)
                                                       ).reduceByKey(lambda a,b: a+b).filter(lambda x
                                                       : x[0] in ['0.25', '0.5', '1']) # (The fourth
                                                       CPU capacity value is non available (empty
                                                       string) so I chose to filter it)

plt.bar(CPU_machines_distribution.keys().collect(), CPU_machines_distribution.values().
                                                       collect(), edgecolor='black', color = 'blue')
plt.show()
```

A.2 What is the percentage of computational power lost due to maintenance (a machine went offline and reconnected later)?

```
# Analysis of the CPU capacity lost due to a REMOVE (1) event :

# Computation of the total CPU capacity :
total_CPU_capacity = machine_events_entries.map(lambda x: x[CPU_capacity_index]).filter(
                                                       lambda x: x != '').reduce(lambda a,b: float(a)
                                                       +float(b))
print(total_CPU_capacity) # Result = 19858.0

event_type_index = 2

# Computation of CPU capacity on machines where REMOVE events occured :
remove_CPU_capacity = machine_events_entries.map(lambda x: (x[event_type_index], x[
                                                       CPU_capacity_index])).filter(lambda x: x[0] ==
                                                        '1').reduceByKey(lambda a,b: float(a)+float(b
                                                       )).values().collect()
print(remove_CPU_capacity) # Result = 4764.0


# Computation of the total memory capacity :
memory_capacity_index = 5
```

```
total_memory_capacity = machine_events_entries.map(lambda x: x[memory_capacity_index]).
                                                filter(lambda x: x != '').reduce(lambda a,b:
                                                float(a)+float(b))
print(total_memory_capacity) # Approximate result = 17996

# Computation of memory capacity on machines where REMOVE events occured :
remove_memory_capacity = machine_events_entries.map(lambda x: (x[event_type_index], x[
                                                memory_capacity_index])).filter(lambda x: x[0]
                                                 == '1').reduceByKey(lambda a,b: float(a)+
                                                float(b)).values().collect()
print(remove_memory_capacity) # Approximate result = 4404
```

A.3 On average, how many tasks compose a job?

```
# Creation of the RDD and keeping the RDD in memory :
task_events = sc.textFile('../data/task-events-part-00000-of-00500.csv')
task_events.cache()

task_events_index = 2
task_events_entries = task_events.map(lambda x: x.split(','))

jobs_ID = task_events_entries.map(lambda x: x[task_events_index])
jobs_ID_count = jobs_ID.map(lambda x: (x,1)).reduceByKey(lambda a,b: a+b)

jobs_ID_values = []

for value in jobs_ID_count.values().collect() :
    jobs_ID_values.append(value)

average_nb_tasks = sum([int(i) for i in jobs_ID_values])/len(jobs_ID_values)
print(average_nb_tasks)
```

A.4 What can you say about the relation between the scheduling class of a job, the scheduling class of its tasks, and their priority?

```
# Creation of the RDD and keeping the RDD in memory :
job_events = sc.textFile('../data/job-events-part-00000-of-00500.csv')
job_events.cache()

job_events_entries = job_events.map(lambda x: x.split(','))

job_sched_class_index = 5
job_ID_index = 2

# RDD with the pairs ('job ID', 'scheduling class') for each job in the job_events table :
ID_sched_class_pair_job = job_events_entries.map(lambda x: (x[job_ID_index], x[
                                                job_sched_class_index]))

task_sched_class_index = 7

# RDD with the pairs ('job ID', 'most frequent scheduling class') for each task in the
                                                task_events table :
ID_sched_class_pair_task_most_common = task_events_entries.map(lambda x: (x[
                                                task_events_index], x[task_sched_class_index])
                                                ).reduceByKey(lambda x,y: Counter(y).
                                                most_common()[0][0])

union = sc.union

# RDD with pairs ('job ID', 'scheduling class' if the most frequent scheduling class of the
                                                tasks == the scheduling class of their job, '-
                                                1' otherwise)
```

```
task_job_sched_class_comparison = ID_sched_class_pair_job.union(
                                                ID_sched_class_pair_task_most_common).
                                                reduceByKey(lambda a,b: a if a==b else -1)

diff_sched_class_count = task_job_sched_class_comparison.groupBy(lambda x: x[1])

priority_task_index = 8

# list containing pairs ('scheduling task number', 'list of tasks priorities')
sched_class_priority_pair_task = task_events_entries.map(lambda x: (x[task_sched_class_index
                                                ], x[priority_task_index])).groupByKey().map(
                                                lambda x : (x[0], list(x[1]))).collect()

sched_class_priority_pair_task_dict = {}

for element in sched_class_priority_pair_task :
    sched_class_priority_pair_task_dict[element[0]] = element[1]


for k in sched_class_priority_pair_task_dict.keys() :
     plt.hist(sorted(sched_class_priority_pair_task_dict[k], key=Counter(
                                                    sched_class_priority_pair_task_dict[k]).
                                                    get, reverse=True), bins=range(len(set(
                                                    sched_class_priority_pair_task_dict[k]))+
                                                    2), edgecolor='black', color = 'red',
                                                    align='right')
     plt.title(f'Repartition of priority codes for scheduling class = {k}')
     plt.show()
```

A.5 Do tasks with low priority have a higher probability of being evicted?

```
low_priority_tasks = task_events_entries.map(lambda x: (x[priority_task_index], x[
                                                task_sched_class_index])).filter(lambda x: int
                                                (x[0]) <= 3)

plt.hist(sorted(low_priority_tasks.values().collect(), key=Counter(low_priority_tasks.values
                                                ().collect()).get, reverse=True), bins=range(
                                                len(set(low_priority_tasks.values().collect())
                                                )+2), edgecolor='black', color='red', align='
                                                right')
plt.title('Distribution of the scheduling class codes for low priority tasks (inferior or
                                                equal to 3)')
plt.show()

all_tasks_sched_class = task_events_entries.map(lambda x: (x[priority_task_index], x[
                                                task_sched_class_index]))

plt.hist(sorted(all_tasks_sched_class.values().collect(), key=Counter(all_tasks_sched_class.
                                                values().collect()).get, reverse=True), bins=
                                                range(len(set(all_tasks_sched_class.values().
                                                collect()))+2), edgecolor='black', color='red'
                                                , align='right')
plt.title('Distribution of the scheduling class codes for all priorities tasks')
plt.show()
```

A.6 In general, do tasks from the same job run on the same machine?

```
machine_index = 4

# RDD of pairs ('job ID', 'machine ID') :
job_ID_machine_pairs = task_events_entries.map(lambda x: (x[job_ID_index], x[machine_index])
                                                ).groupByKey()

nb_of_machines_job = []
```

```
for element in job_ID_machine_pairs.values().collect() :
    nb_of_machines_job.append(len(list(element)))

plt.hist(sorted(nb_of_machines_job, key=Counter(nb_of_machines_job).get, reverse=True), bins
                                                =range(len(set(nb_of_machines_job))+2), color=
                                                'blue', align='right')
plt.title('Number of machines required to run a job')
plt.show()
print(Counter(nb_of_machines_job))
```

A.7 Are the tasks that request the more resources the one that consume the more resources?

```
task_usage = sc.textFile('../data/task-usage-part-00000-of-00500.csv')
task_usage.cache()

task_usage_entries = task_usage.map(lambda x: x.split(','))

machine_index = 4
task_index = 3
CPU_request_index = 9
sampled_CPU_usage_index = -1

# RDD with ('task index', 'CPU request') :
CPU_request = task_events_entries.map(lambda x: (x[machine_index] + ',' + x[task_index], x[
                                                CPU_request_index]))


# RDD with ('task index', 'sampled CPU usage') :
CPU_usage = task_usage_entries.map(lambda x: (x[machine_index] + ',' + x[task_index], x[
                                                sampled_CPU_usage_index]))

union = sc.union

# RDD with ('task index', ['CPU request', 'sampled CPU usage'])
CPU_request_vs_usage = CPU_request.union(CPU_usage).reduceByKey(lambda a,b: a + ',' + b).
                                                filter(lambda x: len(x[1].split(',')) == 2)

request, usage = [], []

for element in CPU_request_vs_usage.collect() :
    request.append(element[1].split(',')[0])
    usage.append(element[1].split(',')[1])

new_request, new_usage = zip(*sorted(zip(request, usage)))
new_request, new_usage = [float(i) for i in new_request], [float(i) for i in new_usage]

plt.scatter(new_request, new_usage)
x = [0,new_request[-1]/2, new_request[-1]]
y = x
plt.plot(x,y, color='red', linestyle='-')
plt.xticks([new_request[0], new_request[-1]])
plt.yticks([new_usage[0], new_usage[-1]])
plt.show()
```

A.8 Is there a relation between the amount of resource consumed by tasks and their priority?

```
# RDD of pairs ('job ID - task ID within job', 'most frequent priority code') :
task_priority = task_events_entries.map(lambda x: (x[job_ID_index] + '-' + x[task_job_index]
                                                , x[priority_task_index])).reduceByKey(lambda
                                                a,b: Counter(b).most_common()[0][0]).filter(
                                                lambda x: x[1][0] != ',')

# RDD of ('job ID - task ID within job', ['most frequent priority code', 'CPU resources
                                                consumed']) :
priority_vs_resources = task_priority.union(task_consumption_CPU).reduceByKey(lambda a,b: (
                                                str(a) + ',' + str(b)).split(',')).filter(
                                                lambda x: isinstance(x[1], list) and len(x[1])
                                                 == 2)

priority_vs_resources_cumulative_values = priority_vs_resources.values().reduceByKey(lambda
                                                a,b: float(a)+float(b))

priority_values = priority_vs_resources_cumulative_values.keys().collect()
resources_values = priority_vs_resources_cumulative_values.values().collect()

plt.bar(priority_values, resources_values,edgecolor='black')
plt.show()
```

A.9 Can we observe correlations between peaks of high resource consumption on some machines and task eviction events?

```
machine_ID_usage_index = 4
mean_CPU_usage_index = 5

# RDD with ('machine ID index', 'mean CPU consumption') :
machine_CPU_consumption = task_usage_entries.map(lambda x: (x[machine_ID_usage_index], x[
                                                mean_CPU_usage_index])).reduceByKey(lambda a,b
                                                : a + ',' + b)

h1 = {}

for element in machine_CPU_consumption.collect() :
    h1[element[0]] = element[1].split(',')
    h1[element[0]] = [float(i) for i in h1[element[0]]]

for k in h1.keys() :
    h1.update({k : sum(h1[k])/len(h1[k])})

# Mean CPU consumption :
m = []
for v in h1.values() :
    m.append(v)

m.sort()

# 99th percentile of CPU consumption first value :
last_percentile_value_CPU = m[int(99/100 * len(m))]

h1_2 = {}

for k in h1.keys() :
    if h1[k] >= last_percentile_value_CPU :
        h1_2[k] = h1[k]


machine_ID_events_index = 4
machine_event_type_index = 5
```

```
# RDD with ('machine ID', 'EVICT event') :
machine_evict_events = task_events_entries.map(lambda x: (x[machine_ID_events_index], x[
                                                    machine_event_type_index])).reduceByKey(lambda
                                                     a,b: a + ',' + b)

h2 = {}

for element in machine_evict_events.collect() :
    if element[0] in h1_2.keys() :
        h2[element[0]] = [int(i) for i in element[1].split(',')]

l = list(h2.values())
l = [item for sublist in l for item in sublist]

plt.hist(l, align='mid')
plt.title('Event codes for the 99th percentile of CPU consumption')
plt.xlabel('Event codes')
plt.show()


# Same thing for the 90th percentile :
last_percentile_value_CPU2 = m[int(90/100 * len(m))]

h1_3 = {}

for k in h1.keys() :
    if h1[k] >= last_percentile_value_CPU2 :
        h1_3[k] = h1[k]

h2_2 = {}

for element in machine_evict_events.collect() :
    if element[0] in h1_3.keys() :
        h2_2[element[0]] = [int(i) for i in element[1].split(',')]

l2 = list(h2_2.values())
l2 = [item for sublist in l2 for item in sublist]

plt.hist(l2, align='mid')
plt.title('Event codes for the 90th percentile of CPU consumption')
plt.xlabel('Event codes')
plt.show()


# Same thing for the 50th percentile :
last_percentile_value_CPU3 = m[int(50/100 * len(m))]

h1_4 = {}

for k in h1.keys() :
    if h1[k] >= last_percentile_value_CPU3 :
        h1_4[k] = h1[k]

h2_3 = {}

for element in machine_evict_events.collect() :
    if element[0] in h1_4.keys() :
        h2_3[element[0]] = [int(i) for i in element[1].split(',')]

l3 = list(h2_3.values())
l3 = [item for sublist in l3 for item in sublist]

plt.hist(l3, align='mid')
plt.title('Event codes for the 50th percentile of CPU consumption')
plt.xlabel('Event codes')
plt.show()
```

B Code for the Part 3

```
import pyspark
from pyspark import SparkContext
import matplotlib.pyplot as plt
from operator import add


### Setting a SparkContext and the adjusting the verbosity of the error messages :
sc = SparkContext('local[1]')
sc.setLogLevel("ERROR")

#Creation of the RDD and keeping the RDD in memory :
machine_events = sc.textFile('../data/part-00000-of-00001.csv')
machine_events.cache()

CPU_capacity_index = 4
memory_capacity_index = 5

machine_events_entries = machine_events.map(lambda x: x.split(','))

# Creating the RDD of ('CPU capacity', 'Memory capacity') for each machine identified by a
                                                machine ID :
CPU_vs_memory = machine_events_entries.map(lambda x: (x[CPU_capacity_index], x[
                                                memory_capacity_index])).filter(lambda x: x !=
                                                 ('','')) # Filtering the unavailable values
                                                ('','')

machines_count = CPU_vs_memory.map(lambda x: (x,1)).reduceByKey(add)

x, y = [], []

for element in machines_count.keys().collect() :
    x.append(float(element[0]))
    y.append(float(element[1]))

z = []

for element in machines_count.values().collect() :
    z.append(float(element))

plt.scatter(x,y,s=z, alpha=0.5)
plt.xlabel('CPU capacity')
plt.ylabel('Memory capacity')
plt.show()
```

References

[1] N. Deng Md E. Haque Z.G. Qin S. Hand M. Harchol-Balter J. Wilkes M. Tirmazi A. Barker. *Borg: the Next Generation*. URL: https://dl.acm.org/doi/pdf/10.1145/3342195.3387517.

[2] M. Korupolu D. Oppenheimer E. Tune A. Verma L. Pedrosa and J. Wilkes. "Large-scale cluster management at Google with Borg". In: *10th European Conference on Computer Systems (EuroSys'15)* (2015), 18:1–18:17.

[3] J. Swiderski P. Zych P. Broniek J. Kusmierek P. Nowak B. Strack P. Witusowski S. Hand J. Wilkes K. Rzadca P. Findeisen. *Autopilot: workload autoscaling at Google*. URL: https://dl.acm.org/doi/epdf/10.1145/3342195.3387524.

[4] Wikipedia. *M/G/1 queue*. URL: https://en.wikipedia.org/wiki/M/G/1_queue.