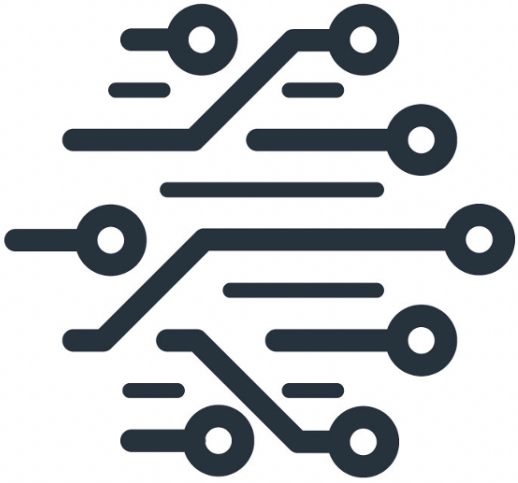
May 2023

Data Center - Challenge 2

Group members:

Susanna Bravi (1916681)
Simone Facchiano (1919922)
Maria Vittoria Vestini (1795724)



GROUP ABRAMSON

1 Formal descriptions of algorithms

1.1 Explorative analysis of the data

Figure 1

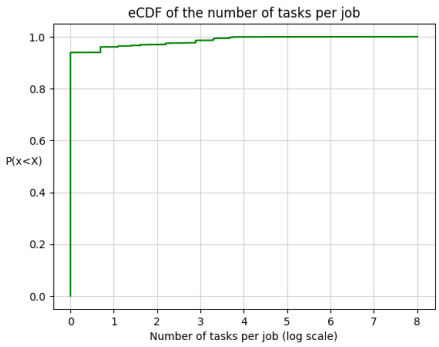


Figure 2

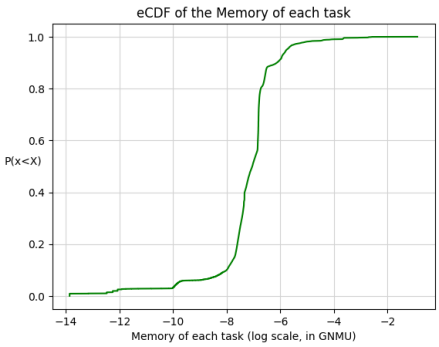
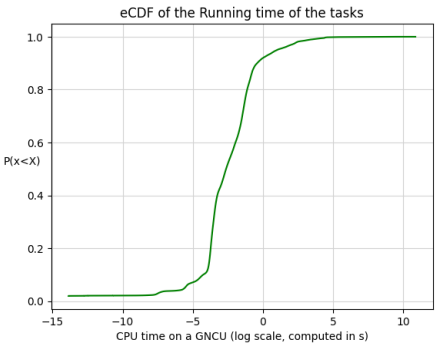


Figure 3



The dataset taken under analysis concerns a fraction of workload offered to the computational cluster of Google. The variables present are the Job ID and Task IDs, the arrival time of task measured in microseconds, the running time in seconds required to run the task on a GNCU and the amount of memory required to run the task, expressed in GNMU. The reported graphs of the eCDFs (x-axis: possible values of the random variable, y-axis: cumulative probability corresponding to each value) were transformed to a logarithmic scale for better interpretability. Looking at the difference between the first and last arrival time, we can see that the portion of the data analyzed covers a period of about 31 days, while in terms of the number of servers, 64 were considered.
As a first analysis, we calculated how many tasks each job consists of. From the eCDF Figure 1 it appears that 94% are composed of only one task, while the job with the most tasks has 2998.
The trends of the Figures 2 and 3 turn out to be similar. 92% of the tasks have *CPU* less than 1 second, and thus in general the tasks take little time to process.
From Figure 2, 91% of the tasks take less than e^{-6} GNMU in memory.
Therefore, we can conclude that our dataset consists mainly of small tasks with rather low running times, and this indicates to us that it might be a good technique to give precedence to tasks with lower running time.

1.2 Dispatching algorithm

In a Data Center, a Dispatcher is a system that can manage the optimal allocation and distribution of tasks to the various available servers in order to reduce the time required to process requests and send responses. The main objective of a Dispatcher is to optimize the Response Time R of the different incoming jobs, that is the time required to complete all the tasks in a job. In the context of Dispatching we identify two different classes of algorithms. On the one hand we have pull schemes, in which servers report their status to the Dispatcher, and on the other hand push schemes, in which the Dispatcher is the one who interrogates the servers to know their status and decide on the schedule.
The LWL (Least Work Left) algorithm belongs to the latter class. In this algorithm, upon the arrival of a new task the Dispatcher sends a message to the servers to obtain information regarding their expected residual work (*Unfinished Work*), so as to assign the task to the server with less work. The ties are resolved randomly.

The LWL algorithm presents problems related to memory, as there may be tasks that are computationally light but require a large amount of memory space, which can lead to a considerable increase in queue length. In addition, such an algorithm requires a large number of messages between the servers and the Dispatcher ($L = 2 \cdot N \cdot M$, where N is the number of servers and M the number of tasks), which in a Data Center with thousands and thousands of servers results in a huge number of communications.

In response to such critical issues, we decided to implement an algorithm that takes inspiration from JBTg-d (Join Below Threshold). In JBT a threshold r, initialized to 0, is considered to identify

whether a server is free or busy. A free server is marked as 1, 0 otherwise. The Dispatcher then holds a vector of length equal to the number of servers in which the status of the servers are reported. At first, the Dispatcher will randomly assign the first incoming task to one of the free servers, which will be marked as 0. The Dispatcher, as long as it receives jobs to complete, continuously checks which servers are free and assigns tasks to them. The real advantage of this algorithm is that the servers report their status only when their queue is less than the current threshold, i.e., only when they can be marked as free, so that the Dispatcher can mark them as free. At this point, every T instants of time the threshold r is updated based on the lengths of the queues: a number K of servers is sampled, and the shortest length among the sampled buffers is set as the new threshold $r = \min_{n_i \in \{n_1,\dots,n_k\}} |Queue(s_{n_i})|$, where s_{n_i} are the k sampled servers. This is as far as the plain JBT is concerned.

In the algorithm we designed, however, the idea was to base the threshold not on queue length as in the original algorithm, but on the amount of Unfinished Work of the servers. The threshold is then updated every T instants of time, but also when all servers are marked as busy.
In particular, the task arrival frequency, λ, is also calculated. A frequency of less than 1 is a symptom of not much traffic, and the threshold will be updated with a more conservative one, so that moments when traffic will become busy again can be identified sooner. In the case, on the other hand, where all servers are busy and λ is greater than 1, the 50th percentile (i.e., the median) of the Unfinished Work of all servers will be taken as the new threshold. This choice is due to the fact that in this case we are dealing with a heavy traffic situation where the queue grows fast and therefore our threshold is less conservative, so that it does not have to be recalculated immediately.

1.3 Server Scheduling algorithm

Unlike Dispatching, Scheduling is concerned with managing and planning the allocation of resources within the server in the Data Center. The goal is again to minimize execution time, and to make full use of the computing resources available.
The algorithm used as a baseline is First Come First Served (FCFS), probably the simplest of the Scheduling algorithms. It is based on a priority system such that the server executes tasks following the order in which they enter the buffer, and thus only takes into account their arrival time without considering the amount of work required to complete them.

However, as is often the case, the adoption of such a simple algorithm can come at a sacrifice in terms of performance, and for this reason FCFS does not turn out to be the best from a performance point of view.
This is due in particular to the priority used by this algorithm, which is part of the so-called non-preemptive priorities, which do not contemplate stopping the processing of a task until it is finished. This choice can lead to delays, due to the fact that a very demanding task that arrived first can keep subsequent tasks that would take far less time to complete blocked.

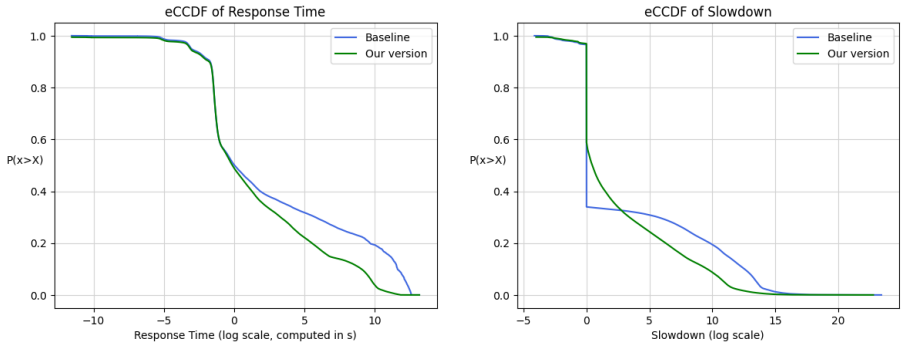
In order to deal with problems such as this, we have referred to an algorithm that is based on preemptive priority, that allows a node to stop processing a task when another task with a higher priority level appears within the buffer. This priority will no longer refer to the arrival time, but rather to the amount of Unfinished Work to complete the task.
This algorithm is called Shortest Remaining Processing Time (SRPT). As soon as a new task enters the queue, its Unfinished Work is compared with that of the task currently running in the server: if the latter is greater, processing is stopped and the new task takes over within the server. It can be shown that SRPT minimizes the Mean System Time for a single server given the tasks assigned to it, and that the following relationship holds true:

$$E(S_{SRPT}) \leq E(S_{FCFS}) \tag{1}$$

2 Performance metric values

2.1 Job response time R and Job slowdown S

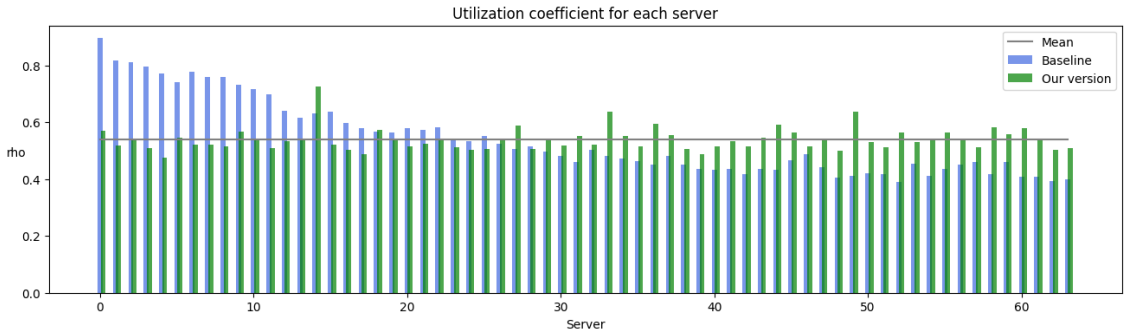
The metrics considered are Response Time R and Job Slowdown S. The Reponse Time is the time from the arrival of the first task of the job until all the tasks of that job have been completely served. Job Slowdown, on the other hand, is the ratio of the job's response time to the sum of the service times of all the tasks in that job. A Slowdown greater than 1 is a symptom that some tasks had to wait for a period of time in the queue. We report below the empirical Complementary Cumulative Distribution Function of the two indices for both the baseline and our algorithm.



For Response Time we notice that the trend of the curves is initially very similar, but shows a marked improvement for longer times. This means that thanks to our algorithm there are fewer jobs with high response times. We can highlight this by also reporting the average values of this index for the two algorithms: $\overline{R}_{baseline} = 27603.90$ seconds and $\overline{R} = 2823.27$ seconds.
Similar discussion for Job Slowdown. Again, our algorithm shows improved performance compared to baseline. Specifically, $\overline{S}_{baseline} = 1241675.57$ and $\overline{S} = 188248.81$.
Therefore, based on these metrics, we can conclude that our algorithm seems to show improved performance.

2.2 Utilization coefficient of server k

$\overline{\rho}_{baseline} = 0.538424$ and $\overline{\rho} = 0.538422$ so the average of the utilization coefficients turns out to be very similar for both algorithms.



The biggest difference, that can be noticed from the plot above, is how the distribution of work within the servers changes with the two algorithms. In fact in our algorithm we see the ρ_k are almost all close to the mean value while those in the baseline are more unbalanced.

2.3 Messaging load L

For the load message calculation, we considered messages between the Dispatcher and the servers regarding servers status updates. The value of $\overline{L}_{baseline} = 128.0$ *messages*, since in each iteration the Dispatcher asks informations to all servers, and the value in our algorithm is $\overline{L} = 1.41$ *messages*.
So in our algorithm the mean number of messages is definitely smaller than in the baseline.