**Parallel Processing: Python Dask**

**Reverse Lecture - 3**

**Team - 08**

**Guided by: Submitted by:**

**Prof. John Gash Shalabh Neema (014546259)**

**Pratyush Sharma(014514162)**

**Introduction**

Dask is a parallel computing library which doesn't simply help in parallelizing high level assortment, yet in addition parallelizes low level tasks or functions and can deal with complex interactions between these functions by making a tasks’ graph. This is similar to threading or multiprocessing modules of Python.

*What problem is being addressed here?*

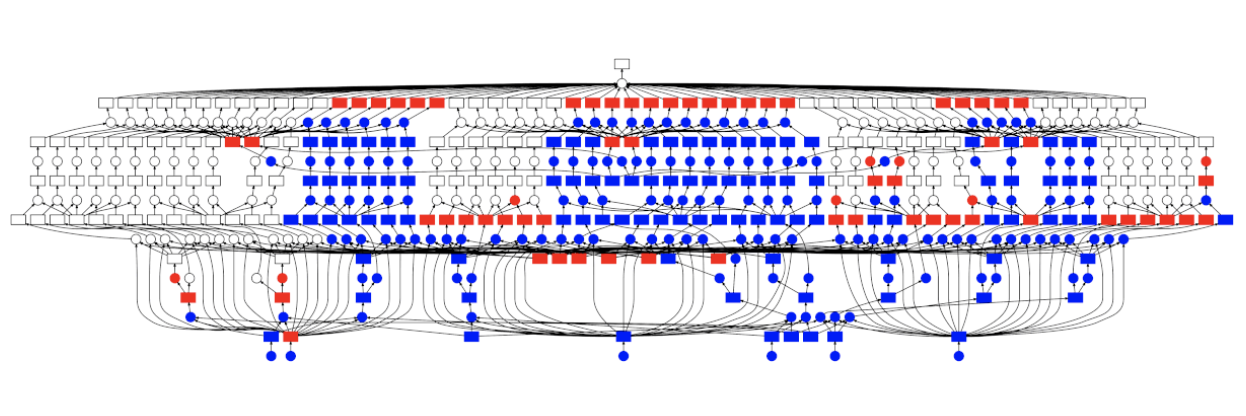
Analysts frequently use apparatuses like Pandas, Numpy and the remainder of the Python environment to dissect information. They like these instruments since they are effective, instinctive, and broadly trusted. In any case, when they decide to move to bigger datasets, they find that these instruments are not intended to scale past a single machine.

In other words, *tools like Pandas and Numpy do not support distributed systems.* This calls for a new tool where these things are addressed.

**Python Dask**

Dask provides ways to scale Pandas, and Numpy work processes all the more locally. It incorporates well with these instruments so it duplicates the greater part of their API and utilizes their information structures inside. Also, Dask is co-created with these libraries to guarantee that they advance reliably, limiting contact while changing from a nearby PC, to a core workstation, and afterward to a distributed cluster.

It parallelizes tasks given to it by making a graph of communications between the assignments. It will be extremely useful to picture what we are doing by utilizing Dask .visualize() method which is accessible with the entirety of its information types and with a complex chain of undertakings you register. This method will output a graph of your tasks, and if your undertakings have numerous nodes at each level, Dask will have the option to parallelize them.



**Data Types**

Every data type in Dask gives a distributed form of existing data types, for example, DataFrame from Pandas, ndarray's from numpy, and list from Python. These data types can be bigger than our memory, Dask will run calculations on your data parallely in Blocked way. Blocked as in they perform huge calculations by performing numerous little calculations. The following are the data types:

Array

Dask Array works on huge clusters, by isolating them into lumps and executing those squares parallely. It has a significant number of numpy techniques accessible which we can use to get speedup.

Arrays can be utilized when your clusters are extremely large (for example they won't fit into memory), in such situations numpy won't be able to take care of that. On the other hand, Dask can be used here, it separates a large cluster into small pieces and works on them in equal. Below is the diagram that compares a Numpy array and a Dask array.

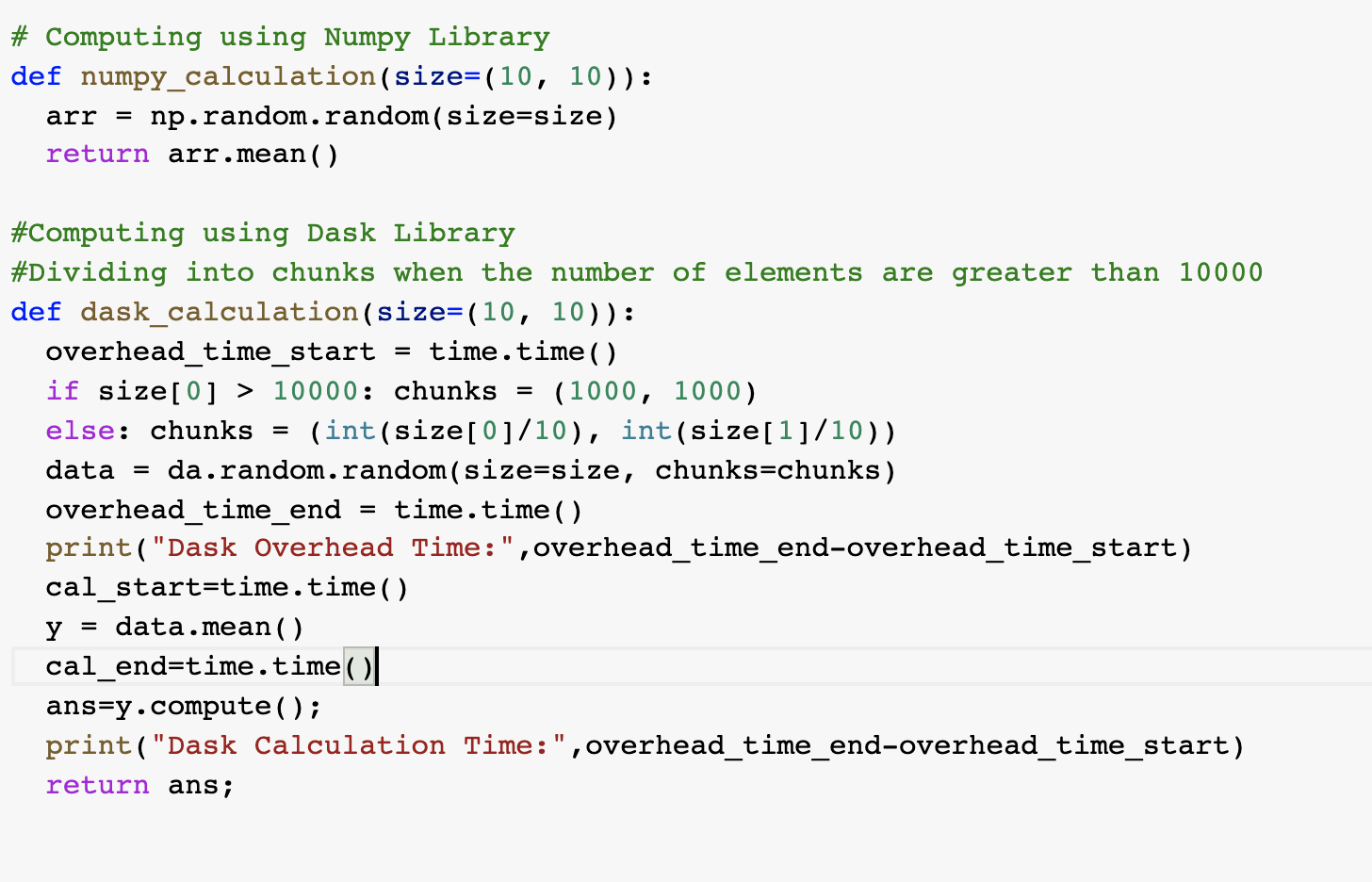
DataFrame

Similar to Dask Arrays, Dask DataFrames parallelize computation on very large Data Files, which won’t fit on memory, by dividing files into chunks and computing functions to those blocks parallely. Most of the functions available on Panda can be utilized in Dask as well. Below is the diagram that shows DataFrames that provide monthly data in DataFrame:

Bag

Dask Bags parallelise calculation on Python's list like objects which contain components of other data types. It is helpful when we are attempting to process some semi-organized information like JSON masses or log records.

**Implementation**

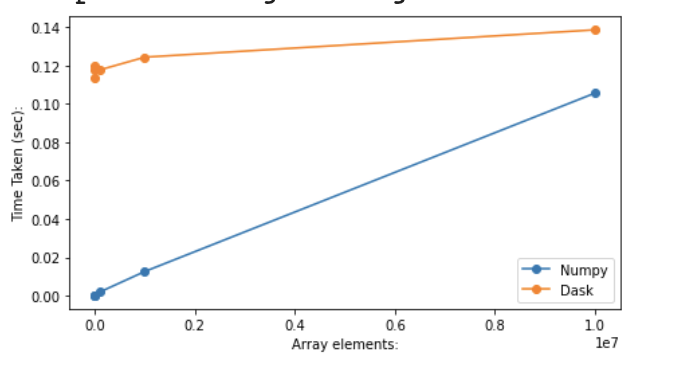
****

**Observations**

We have computed a small operation of performing the mean of the elements using simply numpy and dask. Below are the observation and graphs.

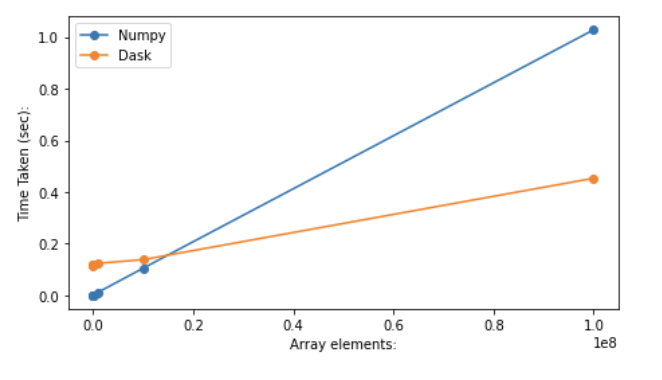
|  |  |  |  |
| --- | --- | --- | --- |
| **No of Array elements** | **Numpy Time** | **Dask Time** |  |
| 100 | 0.0001077651978 | 0.1194717884 |  |
| 1000 | 0.0001976490021 | 0.1137516499 |  |
| 10000 | 0.0002510547638 | 0.1178436279 |  |
| 100000 | 0.001843214035 | 0.117609024 |  |
| 1000000 | 0.01255488396 | 0.1242620945 |  |
| 10000000 | 0.1055629253 | 0.1385157108 | Graph 1 |
| 100000000 | 1.02756381 | 0.4533610344 | Graph 2 |
| 1000000000 | 10.84640551 | 3.982755899 | Graph 3(Numpy line) |
| 10000000000 | Unsupported memory | 45.79357505 | Graph 3(Dask line) |

Graph 1



Above is the graph plotted with the computation of 10 million elements. Here we can see that the numpy performance is better for the small numbers. But as we are increasing the number of elements we can see the difference between the times are coming closer. It seems that Dask will perform better as the number increases.

Graph 2



Above is the graph plotted with the computation of 100 million elements. Here we can see that the Dask has performed better than numpy.

***Analysis of the confusion arose during the presentation:***

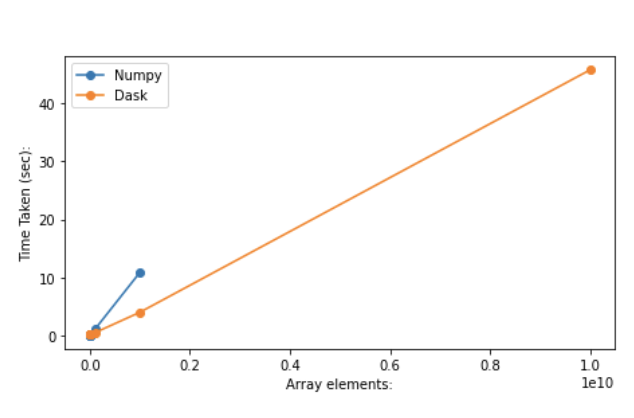
It was observed that the intersection point between Dask line and numpy line is coming at just below 0.2 units in x-axis while the point was not observed in Graph 1 for the whole 1 unit.

**Reason**: It was because of the scaling of x-axis units with each graph.

In Graph 1st, there were a total of 10 million elements present , so each 0.2 unit represents 2 million elements.

In Graph 2, the number of elements increased from 10 million to 100 million, so each 0.2 units represent 20 million elements. In Graph 2 , we can observe that the intersection point is between 0.1 and 0.2 which implies that it has come after 10 million elements. Hence it was not visible in graph 1 and so early at graph 2.

Graph 3

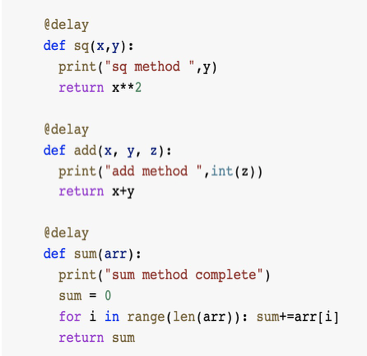


Above is the graph plotted with the computation of 10 billion elements. Numpy was not able to perform at such high numbers as it required too much memory. So we have plotted a numpy line of 1 billion elements to show the limit. While Dask was able to compute the 10 billion elements utilizing the power of parallel computing.

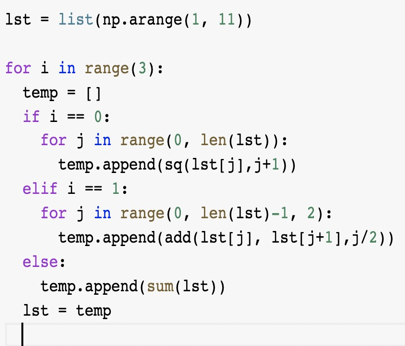
**Dask Delay- Lazy Evaluations**

In case the tasks that you are working on are not that complex enough and we don’t wish to use these High Level Collections, at that point we can utilize Low Level Schedulers which help us to parallelize our calculations utilizing *dask.delayed* interface. In other words, this function does lazy computation.

Implementation

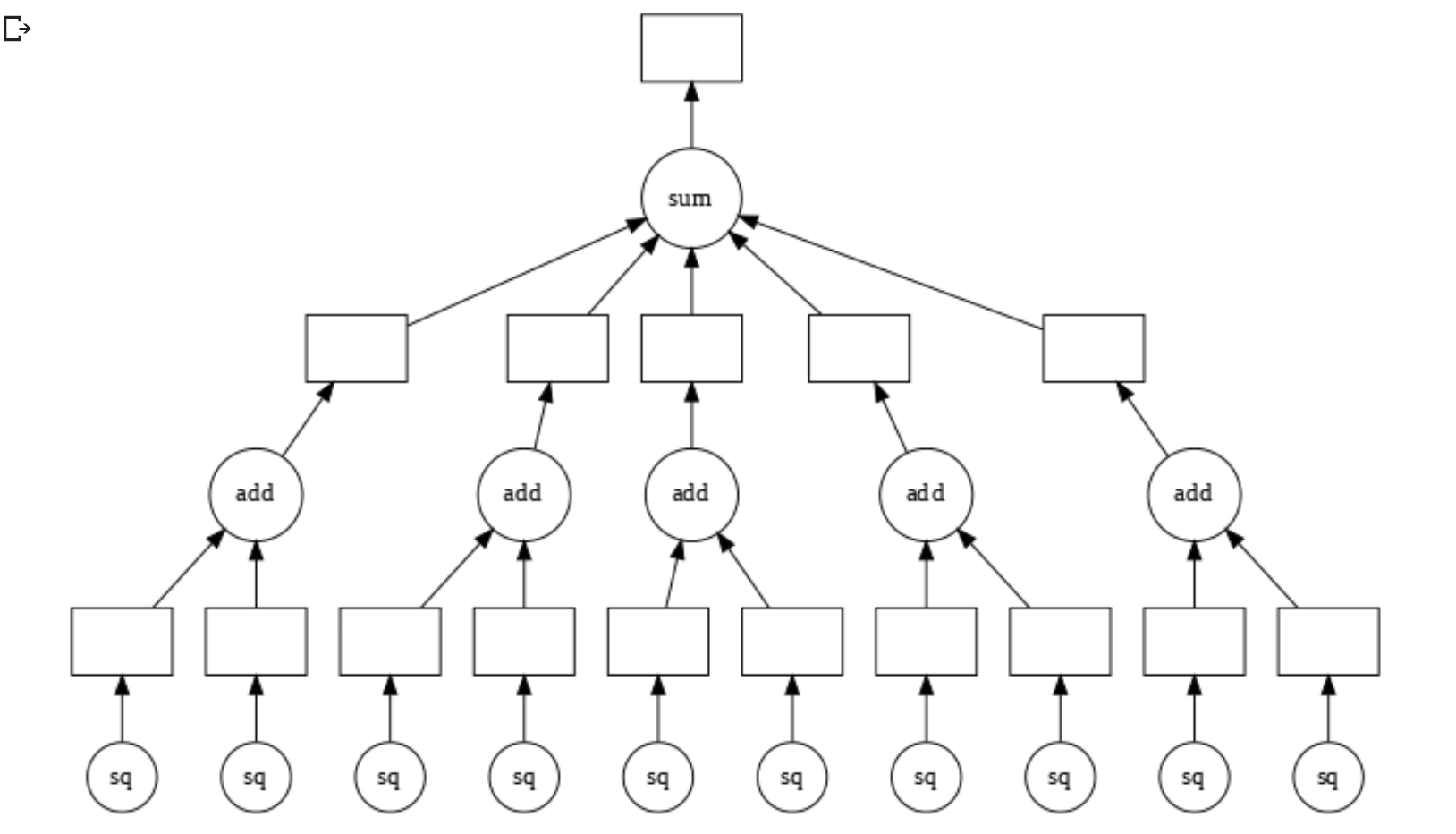


*Adding the delay annotation over the methods.*



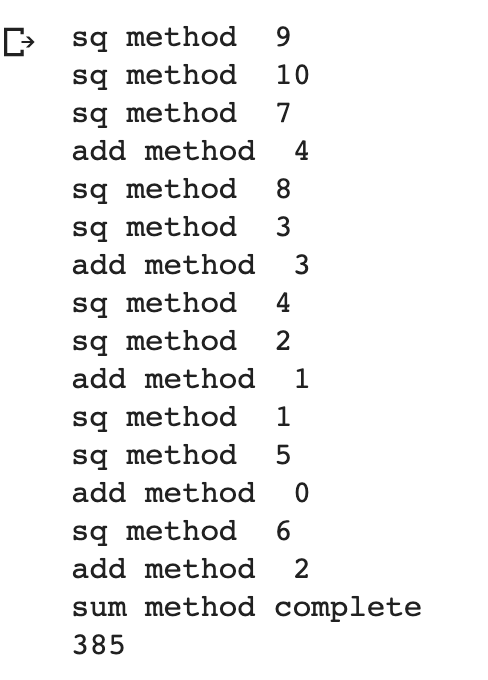
*Constructing a list type object in which we are performing computation at 3 steps .*

In the first layer, we are taking squares of 10 numbers and passing it to the second number.In the second layer, we are taking two consecutive numbers from the output of the first layer and passing the 5 computed values in the third layer. In the third layer we are adding the 5 numbers for the final output.Below is the visualization graph of the list.



*Visualization Graph for above list object*

Output:



*Output displaying characteristics of Lazy Evaluations*

From the output above we can observe that Dask has not computed the functions right away , rather it has generated a graph for the tasks and effectively incorporated the interactions between the function and shown the characteristics of Lazy evaluation

**Conclusion**

Dask has performed significantly better than numpy when the number of elements are higher. For the smaller level collections, dask.delay can help in implementing Lazy evaluation which will compute only when it is required. Dask supports most of the numpy operations and thus many operations of numpy and pandas can be efficiently optimised. Dask can significantly improve the performance of any process that can be divided into smaller isolated chunks .

**References**

→ <https://ml.dask.org>

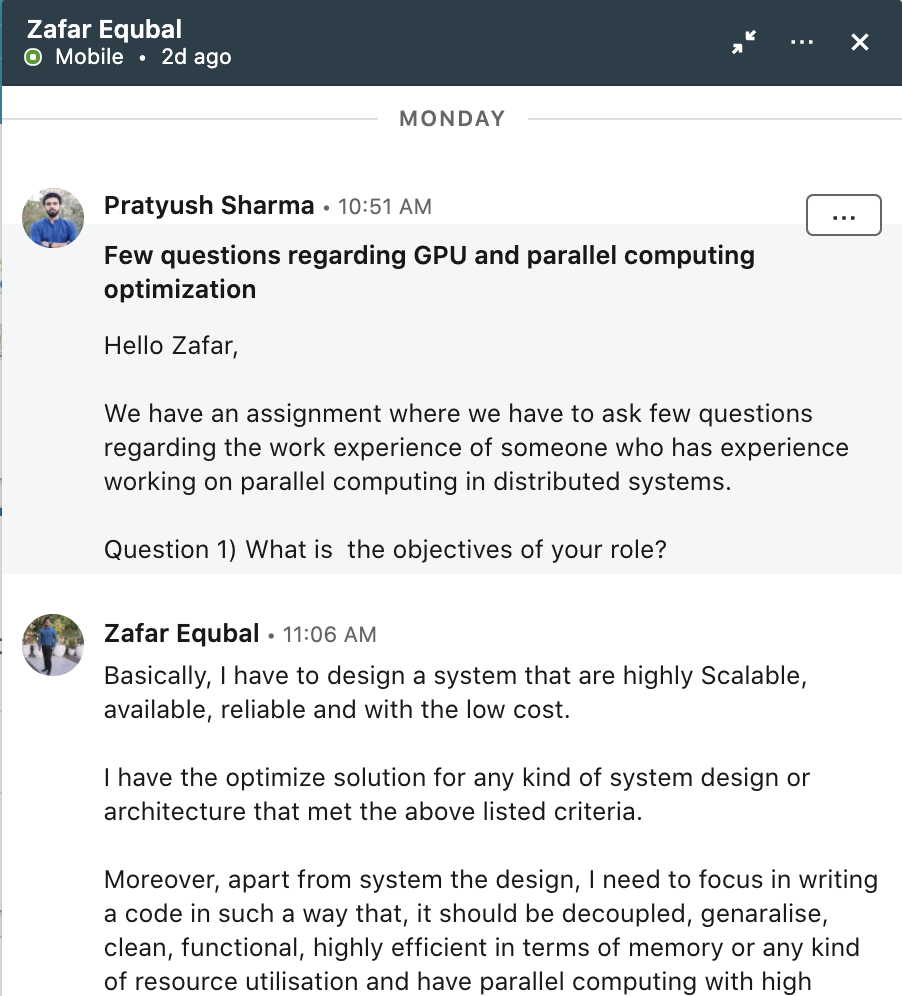
→ <https://docs.dask.org/en/latest/>

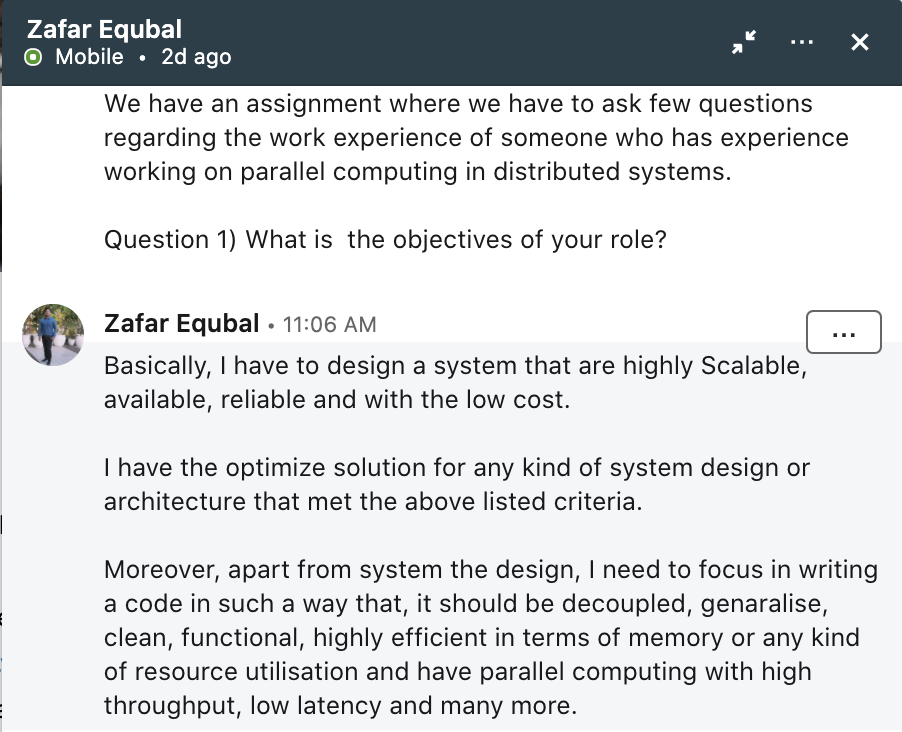
**Extra Credit(**[**https://www.linkedin.com/in/zafarequbal/**](https://www.linkedin.com/in/zafarequbal/)**):**

*Interview with Mr. Zafar Equbal()*

*Solution Architect at Karza Technologies Private Limited*

We had the opportunity of speaking with Mr. Zafar regarding his work on parallel computing. The first thing that we asked was regarding the objective of his role at the company





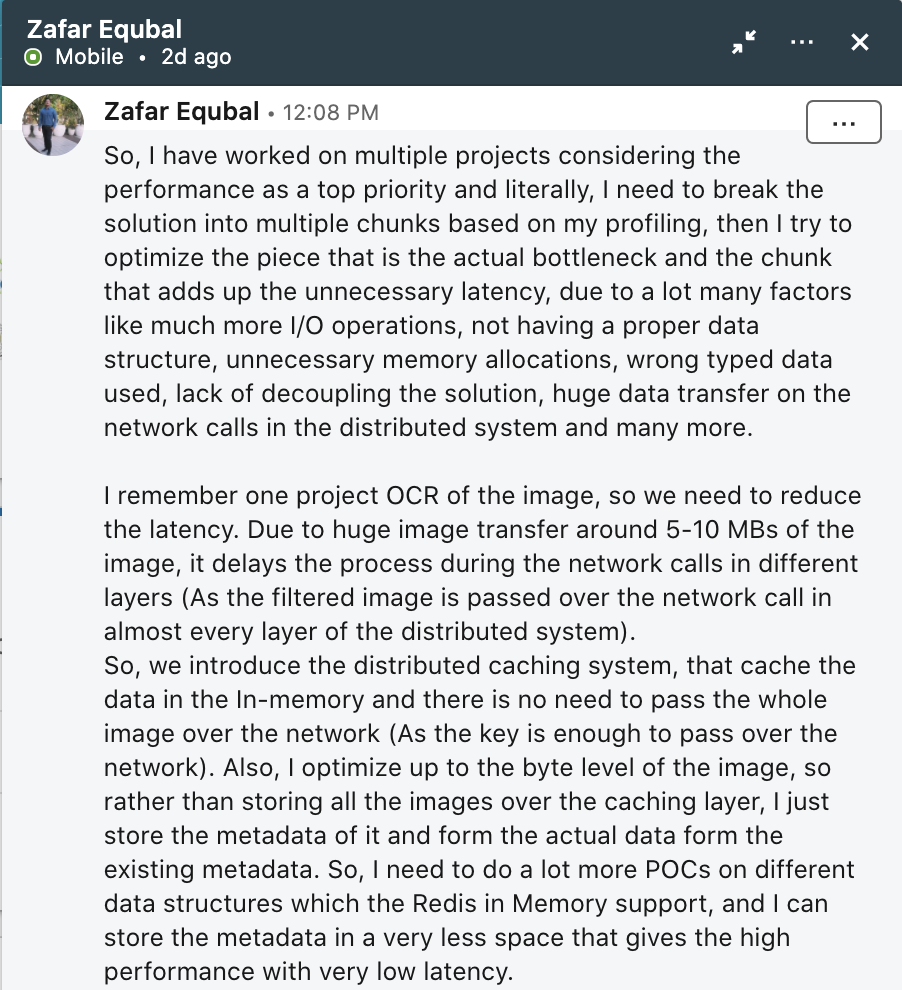
Secondly, we asked him about the skills/job that he possesses which relate distributed computing.

He answered being an expert in distributed computing, he could easily architect a robust system and an efficient work flow for any Enterprise. He also talked about how he could cope with complex use cases using parallel computing.

He also gave us an example of designing an event driven process in a distributed system and how he leveraged his knowledge on serverless systems to come up with a solution for this.

For the third question, we asked him about the factors that he take under consideration for choosing one and trading off others between the following, Memory, Data Transfer and Optimization and how did you achieve that(with example)

→ He answered, he always prefers *optimization.* He also explained to us how he approaches a problem and tries to solve it.

****