Placement and Scheduling of Data Intensive Jobs in Edge-Cloud System

CS 528 Course Project
Harshit S. - 210101048, Dhanesh V. - 210101117, Ketan S. - 210101118

1 Problem Statement

Consider a set of J jobs (arrived at time 0) that need to be processed by a cloud consisting of N physical machines (i.e., nodes) that are **homogeneous**. Each job j has a deadline d_j and must access a set C_j of equal-sized chunks that can be run in parallel on the same machine or a different machine. The chunks are stored in a **distributed file system** on the cloud. Each node can host up to B data chunks and is equipped with S VMs, which implies each node can process S jobs simultaneously.

Let $C = \cup C_j$ be the set of all data chunks available at the central storage server. Replication of data chunks in different machines is allowed, and data chunk needs to be placed **only once** at the beginning and during runtime, we cannot place/replace/replicate the data chunk. The time for each job j to process a required data chunk is unit time, and it is the same for all the jobs. Only **one VM** can access a data chunk in a given time slot of the same physical machine.

The problem aims to minimize the **number of active nodes** ($N_a < N$) to process all the jobs.

2 Optimal Solution using Linear Programming

The given problem can be formulated into *Constrained Optimisation Problem* with **placement** and **deadline** constraints. We have modelled this using **Gurobipy** library in Python and output the number of time slots that have to be scheduled for each chunk before a particular deadline.

2.1 Formulation

Let the number of time slots for each chunk be **sorted** according to their deadline.

$$\text{Objective Function:} \quad \text{Minimize} \left(N_a = \sum_{n=1}^{N} u_n \right)$$

$$\text{Deadline Constraint:} \quad \sum_{c} p_{c,n} \leq B \cdot u_n$$

$$\text{Storage Constraint:} \quad \sum_{c} p_{c,n} \leq B \cdot u_n$$

$$\text{Availability Constraint:} \quad \sum_{c:p_{c,n}>0} f_{c,n,i} \leq d \cdot S \cdot u_n$$

$$\text{VM Access Constraint 1:} \quad f_{c,n,i} \leq d_i \cdot p_{c,n}$$

$$\text{VM Access Constraint 2:} \quad f_{c,n,i} \leq d_i \cdot u_n$$

Where,

- $\mathbf{x_n}$: **binary variable** that is 1 when a node n is active, 0 otherwise.
- $\mathbf{p_{c,n}}$: **Placement variable** which is 1 when chunk c is mapped to node n, 0 otherwise.
- **B**: Number of **chunks** a node can host.
- **S**: **Number of VMs** present in a node.

The last two constraints **upper bound** the number of time slots scheduled for a particular chunk c till deadline d_i to be maximum of d_i because if it exceeds d_i, by *pigeonhole principle*, two VMs will have to access the same chunk at the same time, which is not allowed according to the problem statement.
Multiplying by binary variables $p_{c,n}$ and u_n ensures data locality constraint that we can schedule a chunk only in an active node.

2.2 Time Complexity

Solving an ILP (Integer Linear Programming) is an **NP-Hard** problem; hence, it has exponential time complexity at the cost of providing an optimal solution.

2.3 Space Complexity

Space complexity is dependent on the number of constraints and equations generated:

$$O(C \times N \times D + S \times N \times D)$$

where:

- C is Size of chunk set.
- N is Maximum number of nodes.
- D is the number of deadlines.
- S is the number of Virtual Machines.

2.4 Discussion

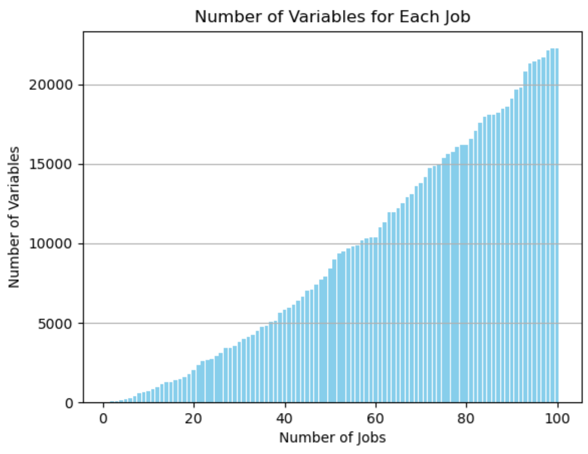

Figure 1: Number of Variables for *ILP* formulation with $B = 3, S = 4$

- It has **exponential time complexity**; hence it can't be used for real-world large problems.

- It generates a lot of variables (exponential as seen in Figure 1) as it considers every possible combination, so the program **runs out of memory** for test cases with more than 1000 jobs.
- Hence, a polynomial, heuristic bounded algorithm is needed to solve this problem most efficiently.

3 First try: Tackling only placement constraint

In this, we try to minimize the number of active nodes by greedily placing the chunks till B chunks are filled in a node.

3.1 Algorithm

Algorithm 1 Deadline Oblivious Algorithm

```
function DEADLINEOBLIVIOUS(B, S, deadlines)
    m ← List of Nodes                                        ▷ Initialize with 1 Node
    for each deadline in deadlines do
        jobEndTime ← 0, currentIndex ← Index of last machine in m, currentChunks ← 0
        for each chunk in deadline.chunks do
            startTime ← Start time of current node
            if currentChunks < B then
                endTime ← startTime + chunk.duration
                Update end time of current node
jobEndTime ← max(jobEndTime, endTime)
                Add chunk to current node
            else
                currentIndex ← currentIndex + 1, endTime ← startTime + chunk.duration
                Update end time of new node, jobEndTime ← max(jobEndTime, endTime)
                Add chunk to new node
            end if
        end for
        deadline.lastChunkTime ← max(deadline.lastChunkTime, jobEndTime)
    end for
    return ans
end function
```

3.2 Problems

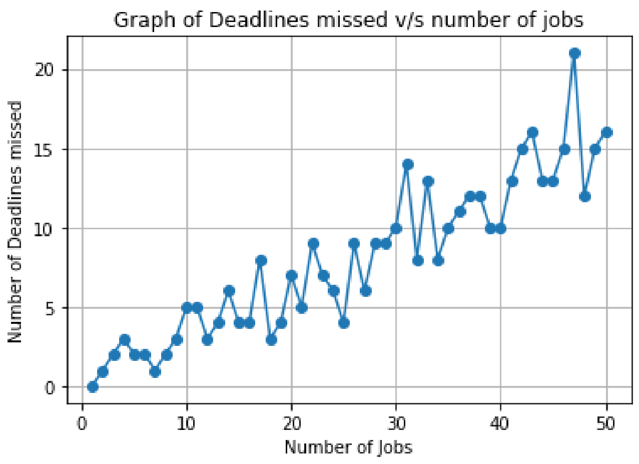

Figure 2: Using deadline oblivious algorithm with $B = 2, S = 2$

- This algorithm results in missing deadlines because we are trying to consider only placement constraints, due to which it can be called *deadline oblivious* as shown in the graph 2.
- Hence, a feasible algorithm is needed considering both deadline and placement constraints.

4 Feasible Solution using First Fit Algorithm

4.1 Algorithm

Algorithm 2 First Fit

```
 1: function FirstFit(B, S, Jobs and their chunks)
 2:     for each job in jobs_to_chunks do
 3:         for each chunk in job do
 4:             while chunks left to schedule do
 5:                 for each machine in GlobalMachines do
 6:                     if whole chunk timeslot fits in machine then
 7:                         Find earliest suitable timestamp for chunk in machine and schedule it.
 8:                     end if
 9:                 end for
10:                 if all chunks are scheduled then
11:                     Exit loop
12:                 else
13:                     Create new machine, push to GlobalMachines.
14:                     continue scheduling remaining chunks
15:                 end if
16:             end while
17:         end for
18:     end for
19: end function
```

4.2 Results

Graph for Number of active nodes v/s number of jobs using the first-fit algorithm is as displayed in the figure:-

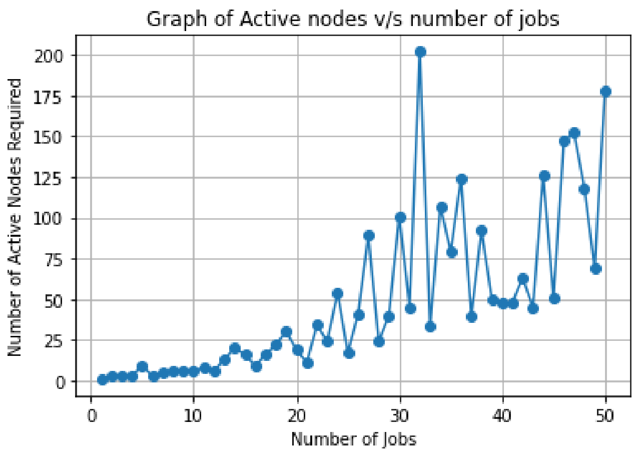

Figure 3: Using First Fit Algorithm with B=3 and S=4

Thus, there is a general trend of number of active nodes getting increased for increase in number of jobs. Since we have taken same B and S and have random number of chunks accessed, we get some anomalies in the graph.

4.3 Discussion

This might produce a feasible schedule, but it won't be optimal as, first, we are trying to find the first machine that will fit *all* timeslots needed by a chunk. However, we could've got better performance by trying to fit as much as we can and then only moving to a new machine. That's exactly what we have tried with Best Fit algorithm.

5 Feasible Solution using Best Fit Algorithm

5.1 Algorithm

In Best Fit algorithm, we iterate over each job and its associated chunks. For each chunk of a job, it tries to schedule it as much as it can in existing machines. If a chunk cannot be fully scheduled in existing machines before the deadline, it creates new machines to accommodate the remaining chunks.

Algorithm 3 Best-Fit Algorithm

```
Input: $B$, $S$, $K$
Initialize $jobs\_to\_chunks$, $deadlines$, $m = 1$, and $v$ as empty vectors/maps
for each job $x$ from 1 to $K$ do
    Read job details: job ID, deadline, number of chunks, and their IDs
    Store deadline for job $x$ in $deadlines$
    Update $jobs\_to\_chunks$ with count of chunks for job $x$
end for
Initialize $v$ as vector of $m$ machine and $m$ as number of machines 1.
for each job $x$ in $jobs\_to\_chunks$ do
    for each chunk $y$ in $x$ do
        Initialize $phase1$ as true
        while $y$ has chunks remaining do
            for each machine $i$ in $v$ do
                if A chunk is already mapped or we have a free space to host that chunk in a machine then
                    Schedule chunks for job $x$ in existing machines
                end if
            end for
            if all chunks for job $x$ are scheduled then
                Break the loop
            end if
            Move to phase 2: create new machines to finish scheduling remaining chunks
        end while
    end for
end for
for each machine $x$ in $v$ do
    Print the schedule for machine $x$
end for
```

5.2 Results

In this comparison, we will examine the outcomes of the Best Fit and First Fit algorithms using a sample test case. Suppose a server can only hold a maximum of **one chunk**, and each node can accommodate **three Virtual Machines**. We need to schedule two jobs in a way that Job 1, which requires access to two data chunks with ID **2** and has a deadline of **25**, can be completed. Job **2**, on the other hand, must schedule four chunks, three with ID **2** and one with ID **1**, and has a deadline of **1**.

Table 1: Schedule for Best Fit Algorithm in 4 Machines

Machine	**Time**	**Chunk**	**Job**	**VM**
M_1	0	C_2	J_2	VM_1
M_1	23	C_2	J_1	VM_1
M_1	24	C_2	J_1	VM_1
M_2	0	C_1	J_2	VM_1
M_3	0	C_2	J_2	VM_1
M_4	0	C_2	J_2	VM_1

Table 2: Schedule for First Fit Algorithm in 5 Machines

Machine	**Time**	**Chunk**	**Job**	**VM**
M_1	23	C_2	J_1	VM_1
M_1	24	C_2	J_1	VM_1
M_2	0	C_1	J_2	VM_1
M_3	0	C_2	J_2	VM_1
M_4	0	C_2	J_2	VM_1
M_5	0	C_2	J_2	VM_1

These results show that the Best fit algorithm requires **fewer node** servers than the first fit algorithm, as it distributes the chunk more optimally between the servers. More particularly, the First Fit algorithm could not schedule C_2 of J_1 in M_1 because it schedules in an existing machine only if all of its chunks can be scheduled, which isn't the problem with Best fit algorithm.

5.3 Time complexity

The time complexity of the provided code for job scheduling in a cloud environment using the best-fit algorithm can be expressed as follows:

$$O(K \times N \times M \times min(D, N))$$

where:

- K is the **number of jobs**.
- N is the **average number of chunks** per job.
- M is the **number of machines**.
- D is the **maximum deadline**.

5.4 Space Complexity

The time complexity of the provided code for job scheduling in a cloud environment using the best-fit algorithm can be expressed as follows:

$$O(K \times N + S \times N \times M)$$

where:

- K is the **number of jobs**.
- N is the **average number of chunks** per job.
- M is the **number of machines**.
- S is the **number of Virtual Machines**.

5.5 Discussion

One of the main drawbacks of the Best-fit algorithm is that it may not minimize the number of node servers used, leading to **over-provisioning** of resources as it doesn't consider only the availability of machines **locally** for every deadline instead of having a global view. This can result in increased costs and resource wastage.

6 Main Polynomial Solution

In this, we proceed in two parts; First, we consider chunks together before a deadline and then **map them to Nodes** and in the Second part, we map them to a particular VM taking care of the constraints that (i) No **two VMs** can access same chunk at same time slot and (ii) Replication of chunks in the same node is not allowed. The pseudocode is outlined below:

6.1 Part 1: Mapping Chunks to Nodes

Algorithm 4 Chunk Scheduling Algorithm

```
1: Output: mapping of chunks to a node and time slot assigned before every deadline
2: Initialize variables and data structures
3: Read input values $B$, $S$, $K$, and chunks accessed by jobs
4: Sort chunks in deadline_chunks based on time slots left
5: for $d$ in deadlines do
6:     Get chunks to schedule in $v$ for current deadline
7:     while $v$ is not empty do
8:         Sort $v$ based on time slots left
9:         Find first indices for which sum of time slots left is $\geq S \cdot d$ using sliding window
10:        Reuse free chunk slot in previous machine to optimize it
11:        Schedule chunks in new/existing machines using SCHEDULE function
12:    end while
13:    for machine in machines_scheduled do
14:        for next_deadline in deadlines after $d$ do
15:            Schedule chunks for next deadline in the same machine using SCHEDULE function
16:        end for
17:    end for
18: end for
```

Algorithm 5 schedule

```
function SCHEDULE($d$,chunks to be scheduled, $m$)
    Output: Assignment of number of time slots to feasible chunk in $m$ before deadline $d$.
    Sort chunks based on timeslots left for each chunk at deadline $d$ and machine $m$
    while chunk set is not empty do
        Compute number of slots to schedule for current chunk
        if slots can be scheduled then
            Schedule slots for chunk in machine $m$
            Update slot counters and chunk timeslots
            if All slots scheduled for machine $m$ then
                Update last filled deadline for machine $m$
                break
            end if
        else
            Schedule remaining slots partially and Update slot counters and chunk timeslots
            if All slots scheduled for machine $m$ then
                Update last filled deadline for machine $m$
                break
            else
                Move to next chunk in queue
            end if
        end if
    end while
end function
```

- In this algorithm, we tackle the constraint that no two VMs can access the same chunk in the same time slot in step 5 of Algorithm 5 by calculating the number of time slots needed by a chunk c to be scheduled in a machine m before deadline d as:

  $$\boxed{\min\left(d - \text{num. of slots scheduled so far in same machine for } c, \text{num. of slots needed in actual}\right)}$$

  This is because if more than the minimum of these two slots are already scheduled, the *pigeonhole principle* tells us that two VMs should access the same chunk c in the same time slot.

- We have tried to optimize the number of machines by trying to:

  1. Schedule time slots required for a chunk in later deadlines in the **same machine** if possible.
  2. **Reuse** empty B slots in the same machine if it's possible in the next iteration.

6.2 Part 2: Mapping Chunks to VM within a Node

The algorithm below is a polynomial algorithm that will map all slots needed to be scheduled in a machine before each deadline to each VM of the machine given the **valid** timeslots that each chunk should get before a deadline in a machine. Here, we iterate between deadlines and schedule time slots between the current and previous deadlines. We first try to **fill** one VM and then move on to the next VM. Constraints such as no two VMS should access a chunk together are managed here. We start with the first VM and map all chunks possible between the current and previous deadlines for this VM, then move on to the next VM. In Case All

VMs are full the slots left are pushed to be scheduled between the next two previous deadlines. This ensures no chunks miss their deadlines.

Algorithm 6 ScheduleVMs

```
function SCHEDULEVMS($m$)
    Output: Schedule of each VM in machine $m$.
    for $d$ in decreasing order of deadlines do
        Sort chunks mapped to machine $m$ before deadline $d$ based on timeslots needed.
        for $c$ in sorted chunks do
            $VM_{id} \leftarrow 1$, $slots \leftarrow$ Slots scheduled for this chunk
            if $slots > d - \text{prevd}$ then
                Push extra slots to the previous deadline.
            end if
            while $slots > 0$ do                                      ▷ Schedule all slots
                if $VM_{id} \geq S$ then
                    Push slots to the next deadline and break
                end if
                Assign chunk $c$ to VM $VM_{id}$ for the $(\text{prevd} + i)^{th}$ timeslot.
                if $i \geq d - \text{prevd}$ then
                    Reset $i$ and increment $VM_{id}$.
                end if
            end while
        end for
    end for
end function
```

6.3 Results

This section outlines the various insights that we can get from our algorithm, which might be useful in real-world applications.

6.3.1 Utilisation per Node

Table 3: **Node utilisation** for 12 Jobs, $S = 5$ and $B = 10$ For $N_a = 4$

M1	M2	M3	M4
31.2%	29.6%	12%	12%
68.05%	63.8%	77.8%	30.55%
77.8%	46.3%	46.03%	10.67%
61.33%	49.33%	21.33%	52%

It is observed that machines that come earlier generally have higher utilisation rates than those coming after them, and it is quite evident from our algorithm, too, as the first produced machines are passed through loops to allow as much reuse as possible. Although this relation is not always true, as is observed in some of the examples it is a weak generalisation.

6.3.2 Utilisation per VM

Table 4: **VM utilisation** for 12 Jobs, $S = 5$ and $B = 10$ For First Machine

VM1	VM2	VM3	VM4
84%	32%	8%	4%
76%	24%	12%	8%
78.2%	73.9%	73.9%	60.8%
91%	60%	47.8%	0%

VM utilisation inside each machine is relatively consistent and decreases monotonically. This is evident from our greedy approach as we try to fill in 1st VM and move on to the next only if the current VM is full, thus resulting in a proper decreasing function.

6.3.3 Number of Active Nodes

The graph for **active nodes** used v/s number of jobs for random simulated test cases is shown below:

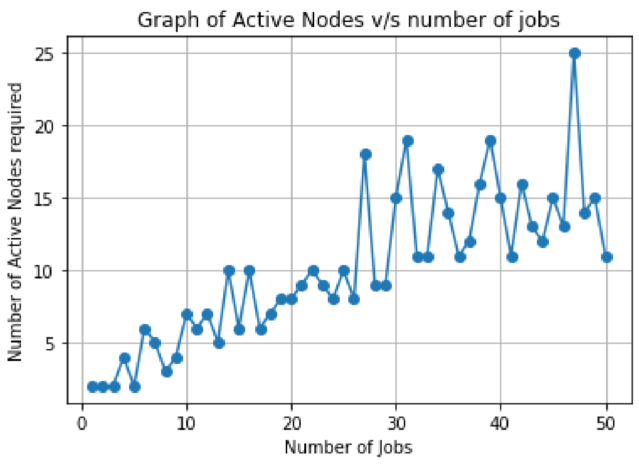

Figure 4: Using Main Polynomial Algorithm with B=3 and S=4

6.3.4 Number of Replications

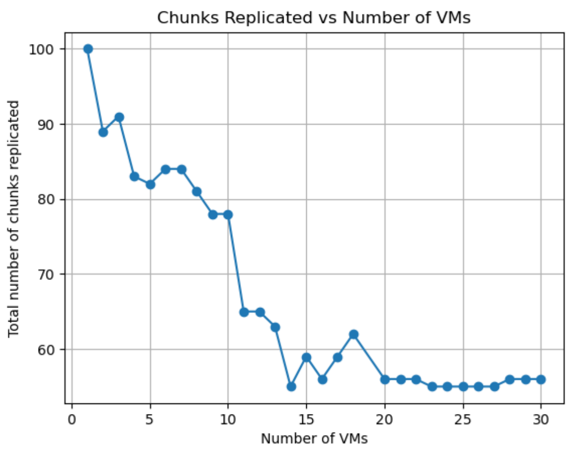

Figure 5: Number of replications for 30 Jobs and $B = 20$

The number of replications naturally **decreases** with an increase in the number of VMs as now each machine can handle more chunks. But after a certain number, there is no decrease as it is no longer possible to decrease replication; these replications are required to fulfil deadlines as in each machine, only one VM can access one chunk in a given timeslot. Each machine can, at most, access a chunk of time before the deadline d. Thus, we must replicate to meet deadlines when slots required are more than d. This metric would be important as, in real-life applications, there would be costs associated with replicating a chunk to ensure synchronisation and concurrency.

6.4 Time Complexity

The time complexity of the Part 1 Schedule algorithm can be expressed as follows:

$$O(C^2 \times \log C)$$

where:

- C is the **maximum number of Chunks** in a deadline.

The time complexity of the Part 2 Schedule VMs algorithm can be expressed as follows:

$$O(D \times (C \log C + S)) = O(D \times C \log C)$$

where:

- D is the **number of deadlines**.
- C is the **maximum number of Chunks** in a deadline.
- S is the **Number of Vms** in a machine.

Thus overall time complexity of algorithm can be expressed as follows:

$$O(M \times D \times C \log C + C^2 \times \log C)$$

where:

- M is the number of Machines or Nodes used.

6.5 Space Complexity

The space complexity of the Part 1 Schedule algorithm can be expressed as follows:

$$O(D \times C)$$

where:

- D is **the number of deadlines**.

The space complexity of the Part 2 Schedule VMs algorithm can be expressed as follows:

$$O(D_{Max} \times S)$$

where:

- D_{Max} is the **maximum deadline**.
- S is the **Number of Vms** in a machine.

7 Conclusion

Thus, we have proposed an algorithm to solve the given problem closest to the optimal solution as much as possible. We have also formulated the problem as Integer linear programming to get the optimal solution possible.