CS 528: High Performance Computing

Project Report

Task:

Placement and Scheduling Data Intensive Jobs in Edge-Cloud System

Group members:

Lavanya Koleti (210101057)

Harshavardhani Thota (210101105)

Nikhitha Vanga (210101107)

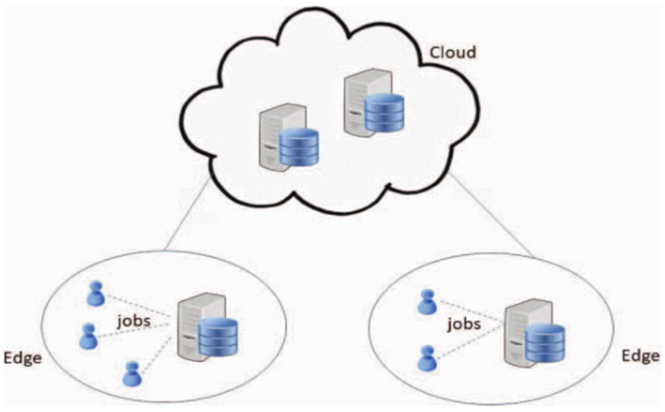
Instructor:

Prof. Aryabatta Sahu, Dept. of CSE, IITG

1 Problem Statement

Consider a computational environment comprising N homogeneous physical machines (nodes) within a cloud infrastructure. At time $t = 0$, a set of J jobs arrive for processing, each with a distinct deadline d_j. Each job j necessitates access to a set C_j of uniformly sized data chunks. We assume that each job j comprises $|C_j|$ tasks, each accessing a single data chunk, which can be executed in parallel on the same or different machines. These data chunks are stored within a distributed file system on the cloud.

Each node in the cloud can accommodate up to B data chunks and is equipped with S virtual machines (VMs). Consequently, each node can concurrently process S jobs. Let $C = \bigcup_j C_j$ represent the set of all data chunks available at the central storage server. Before processing begins, the necessary data chunks must be transferred to the physical machines. Replication of data chunks across different machines is permitted, with each data chunk being placed only once before any job execution commences. During runtime, data chunk placement, replacement, or replication is not allowed.



Edge-clouds environment

Figure 1: Edge clouds environment as depicted in the problem statement where data chunks are brought and stored at the edge and jobs are processed there.

The time required for each job j to process a data chunk is unit time, consistent across all jobs. The completion of a job j before its deadline d_j necessitates processing all the required data chunks in C_j before the deadline. Furthermore, only one VM can access a data chunk within a given time slot on the same physical machine.

The objective is to minimize the total number of active nodes (N_a) required to process all jobs. An active node is defined as a node that stores at least one data chunk and is processing at least one job. The constraint $N_a < N$ ensures that the number of active machines remains below the total number of available machines.

2 Main Idea

The problem at hand can be decomposed into two distinct subproblems: one focused on minimizing the number of active nodes while optimizing data chunk placement, and the other centered on minimizing active nodes while ensuring all jobs are scheduled before their deadlines. By jointly addressing these subproblems, we aim to formulate a unified approach that optimizes both job scheduling and data placement in cloud-based data processing environments.

This joint optimization aims to minimize the number of active nodes under task deadline and data locality constraints. Specifically, the optimization seeks to efficiently allocate tasks to physical machines, considering data locality to minimize data transfer overhead, and schedule job executions to meet specified deadlines.

We considered a cloud framework similar to MapReduce, where jobs are partitioned into small tasks that are processed in parallel by different VMs. Our goal is to minimize the total number of active nodes needed to complete the jobs satisfying a deadline constraint d_j for each job j, the data locality constraint, and physical resource constraints on each node. We consider a time-slotted model where jobs are scheduled to execute in fixed-length time slots. Since each node is equipped with S VMs, it has S slots available at each time t. Our control knobs in the optimization include data chunk placement, job scheduling, and cloud sizing. We first formulate this cloud right-sizing problem as an integer optimization.

Completing a job j before deadline d_j is equivalent to processing all the required chunks $c \in C_j$ before the deadline. When a chunk is accessed by multiple jobs, we need to guarantee that the chunk receives sufficient processing time (i.e., time slots) before each target deadline in order to support all the jobs. Therefore, we can formulate the job scheduling problem in terms of required processing time for each chunk. Denote $D^{\uparrow} = \bigcup_j \{d_j\}$ to be the set of D distinct deadlines. Without loss of generality, we assume the deadlines in $D^{\uparrow}$ are ordered, so that $d_i^{\uparrow} < d_s^{\uparrow}$ for all $i < s$ and $d_i^{\uparrow}, d_s^{\uparrow} \in D^{\uparrow}$.

The key idea is that solving the CRED (cloud right-sizing with execution deadlines and data locality) problem requires a joint optimization of job scheduling and chunk placement that addresses both execution deadline and data locality constraints in a collaborative fashion. So we will first focus on a special case where all jobs require equal execution deadline. Then iteratively apply equal deadline algorithm to incrementally find the chunk placement and time-slot schedules to meet each deadline one by one. More precisely, after finding a solution for placing chunks $c \in C_1, ..., C_i$ to meet deadlines $d_1^{\uparrow}, ..., d_i^{\uparrow}$, we reuse the already placed chunks on existing nodes (if there are remaining computation resources available) and minimize the number of new nodes we need to add in order to support $F_{c,i+1}$ for all chunks $c \in C_{i+1}$. This process continues until all the deadlines are considered.

3 Solving CRED with Equal Deadlines

Consider the chunk set C_i for d with size $|C_i|$. We sort all chunks in descending order based on the number of required time slots for d and record the order in an array, R_d. The chunk recorded at the head of R_d has the largest number of required time slots for $d_i^{\uparrow}$. We denote $H_{B,d_i^{\uparrow}}$ as the first set of B contiguous chunks, from the tail of the array (the basic idea is that when chunks with small deadlines are scheduled first, they can be finished early and we can remove more chunks), with the total number of required time slots larger than or equal to $S \cdot d$, at the beginning of an iteration. We denote $C_{b,i}$ as a set of b chunks from C_i.

- When $\sum_{c \in H_b} F(r)_c > S \cdot d$, we place $H(r)_{B,d}$ into a node and call SCHEDULE for time-slots' scheduling. The condition $c \in H_b F(r)_c > S \cdot d$ guarantees that $H(r)_{B,d}$ exists. By choosing $H(r)_{B,d}$, we can schedule at most $S \cdot d$ time slots in each node. Choosing $H(r)_{B,d}$ and calling SCHEDULE guarantees that we can take care of as many chunks as possible while scheduling maximum possible time slots in each iteration. If the remaining number of required time slots for chunk c is 0, we can remove the chunk c from the chunk set C and reduce the size of the chunk set C.
- When $\sum_{c \in H_b} F(r)_c < S \cdot d$, we can place any B chunks into a node and SCHEDULE and remove all of them.

The basic idea of SCHEDULE is that by scheduling time slots from the chunks with the smallest number of required time slots, we can remove more chunks. The inputs to SCHEDULE are a set of chunks and the number of time slots needed for d. The number of time slots needed is the number of time slots waiting for scheduling in a node. We denote the number of time slots needed in the rth iteration as $NTS(r)_d$. We always take the number of slots that are required for a chunk as minimum of actual slot required and deadline as if more than deadline slots are given to the same node, that implies that a virtual machine is accessing the same chunk of data which is not allowed hence we can only schedule at most of deadline slots of the same chunk on a physical node. Then schedule the chunks from the given chunk dataset according to time slots available.

Algorithm 1 CRED Algorithm

```
1: **Input:** $C$, $B$, $d$, $S$
2: **Output:** $\hat{N}$
3: $C(r) = C$
4: while $C(r) > 0$ do
5:     sort chunks based on the number of required time slots
6:     if $\sum_{c \in H_b} F(r)_c > S \cdot d$ then
7:         place $H(r)_{B,d}$ into one node
8:         SCHEDULE($H(r)_{B,d}$, $S \cdot d$)
9:     else
10:        break
11:    end if
12: end while
13: while $C(r) > 0$ do
14:     place $C_B$ into one node
15:     SCHEDULE($C_B$, $S \cdot d$)
16: end while
```

Algorithm 2 Algorithm 2: SCHEDULE(C, NTS)

```
1: Sort $C$ based on the remaining number of required time slots for $d$ in ascending order
2: for $c = 1$ to $|C|$ do
3:     $f_c = \min\{()\} d, c_i)$
4:     if $k - NTS_d > 0$ then
5:         $k = k - NTS_d$
6:         $NTS_d = 0$
7:         break
8:     else
9:         $NTS_d = NTS_d - k$
10:        $k = 0$
11:        $c_i = c_i - k$
12:        if $c_i = 0$ then
13:            Remove chunk from set $C$
14:        end if
15:    end if
16: end for
```

4 Solving CRED with different deadlines

We followed a heuristic algorithm to solve CRED with multiple, arbitrary deadlines. Our idea is to find the chunk placement and time-slot schedules in already present active nodes then iteratively apply CRED-S to schedule rest of chunks that require more time slots. More precisely, after reusing the already placed chunks on existing nodes (if there are remaining computation resources available), we find a solution for placing chunks $c \in C_1, ..., C_i$ to meet deadlines $d_1^{\uparrow}, ..., d_i^{\uparrow}$ and optimize for the next deadline $d_{i+1}^{\uparrow}$ and minimize the number of new nodes we need to add in order to support $F_{c,i+1}$ for all chunks $c \in C_{i+1}$. This process continues until all the deadlines are considered. The algorithm is summarized in CRED-M. $\hat{N}_i$ denotes the number of nodes needed for scheduling jobs with deadline $d_i^{\uparrow}$.Assume there are D distinct deadlines. We consider chunk placement and time slots scheduling of distinct deadlines one-by-one, from $d_1^{\uparrow}$ to $d_D^{\uparrow}$. For deadline $d_i^{\uparrow}$, CRED-M first schedules time slots for deadlines from $d_i^{\uparrow}$ to $d_D^{\uparrow}$.

Algorithm 3 CRED_M Algorithm

```
 1: **Input:** $jobs$, $B$, $S$
 2: **Output:** $\hat{N}$
 3: sort $jobs$ based on their deadlines
 4: initialize $data\_deadline$ as a map of integers to vectors of integers
 5: $\hat{N} \leftarrow 0$
 6: for each $job$ in $jobs$ do
 7:     for each $chunk$ in $job.chunks$ do
 8:         add $chunk$ to $data\_deadline[job.deadline]$
 9:     end for
10: end for
11: for each $deadline$ and $chunks$ in $data\_deadline$ do
12:     calculate $R$ from $chunks$
13:     if $active\_nodes$ is empty then
14:         $\hat{N} \leftarrow ans + cred\_s(R, deadline, B, S)$
15:     else
16:         for each $R\_value$ in $R$ do
17:             handle $R\_value$ with active nodes
18:             if $R\_value$ is still not 0 then
19:                 for each $active\_node$ in $active\_nodes$ do
20:                     attempt to assign chunks to active nodes
21:                 end for
22:             end if
23:         end for
24:         sort and reverse $R$
25:         while last element in $R$ list has first element equal to 0 do
26:             remove last element from $R$ list
27:         end while
28:         $ans \leftarrow ans + cred\_s(R, deadline, B, S)$
29:     end if
30: end for
31: print state of $active\_nodes$ for debugging
32: return $ans$
```

5 Evaluation

5.1 Examples and optimality

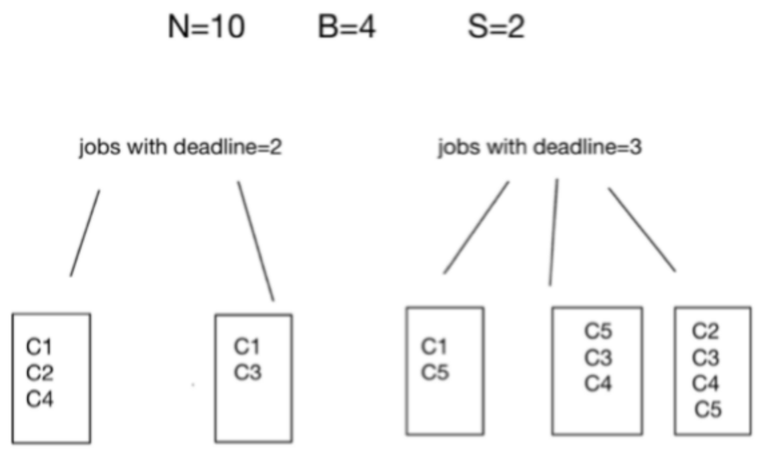


Figure 2: An example with number of nodes=10, virtual machines on each node=2 and limit of datachunks that can be placed on each node=4 and jobs requirements as shown.

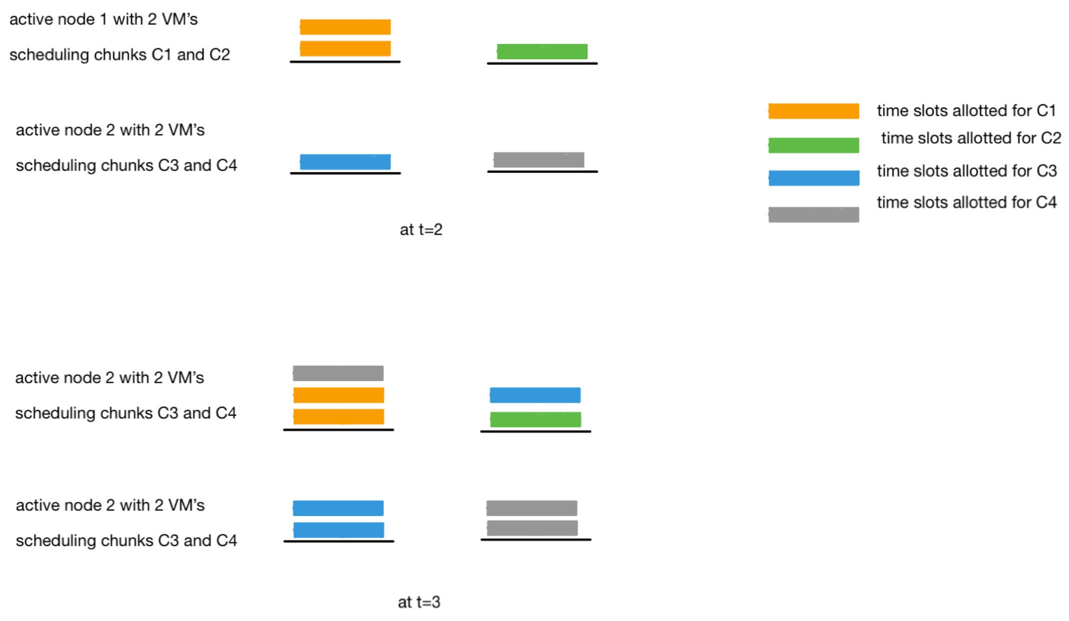
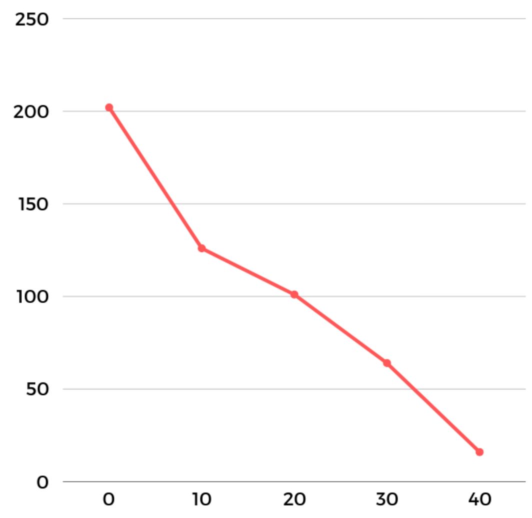

active node 1 with 2 VM's
scheduling chunks C1 and C2
active node 2 with 2 VM's
scheduling chunks C3 and C4
at t=2
time slots allotted for C1
time slots allotted for C2
time slots allotted for C3
time slots allotted for C4
active node 2 with 2 VM's
scheduling chunks C3 and C4
active node 2 with 2 VM's
scheduling chunks C3 and C4
at t=3

Figure 3: Visualisation of how number active nodes are minimised by reusing previous active nodes and scheduling new nodes.



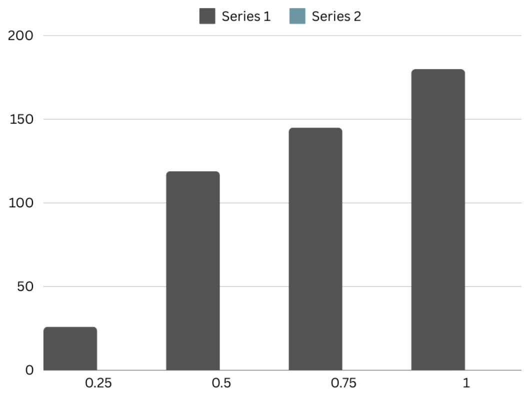


Figure 4: line chart represents effect of the time difference between d1 and d2 on the total number of nodes and the bar graph represents effect of the ratio of d1- to d2-type of jobs on the number of nodes needed.

5.2 Assumptions

1. For a given job, the data chunks it requires are all distinct.
2. All data chunks are placed before scheduling tasks; therefore, the time taken for getting data chunks is not considered.
3. We assume that there is no node failure.

5.3 References

1. M. Xu, S. Alamro, T. Lan and S. Subramaniam, "CRED: Cloud Right-Sizing with Execution Deadlines and Data Locality," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 12, pp. 3389-3400, 1 Dec. 2017, doi: 10.1109/TPDS.2017.2726071.(link to reference)