Placement and Scheduling Data Intensive Jobs in Edge-Cloud System

Aasneh Prasad, Sanjana Siri Kolisetty, Swagat Bhupendra Sathwara
210101001, 210101093, 210101102
Department of Computer Science and Engineering, IIT Guwahati

Abstract

Efficient allocation of resources is important for meeting performance objectives while minimizing costs. The objective of the problem of job scheduling in a cloud infrastructure consisting of homogeneous physical machines is to minimize the total number of active nodes required to process a set of jobs with given deadlines, each accessing a specific set of data chunks.

The proposed algorithms focus on optimizing the allocation of jobs to virtual machines (VMs) on physical nodes with respect to the constraints. By strategically placing data chunks and scheduling jobs, the algorithms aim to maximize resource utilization and meet job deadlines, all while minimizing the number of active nodes.

This report presents several algorithmic solutions. Through experimental evaluation and analysis, the performance of these algorithms is assessed.

Keywords: Job, Chunk, Node, Virtual Machine, Deadline

1 Introduction

1.1 Problem Statement

Detailed breakdown of the problem statement is given below:

- **Cloud Environment:** In the cloud, each physical machine, also known as **Node** contains S virtual machines. The cloud has a central storage server which stores the set C which contains all the data chunks. Each node can replicate these data chunks and can host upto B data chunks.
- **Job Characteristics:** A set of J jobs arrive at time $t = 0$. Each job J_i has to process the set of chunks C_i. A job is said to be completed when all the chunks in it's chunk set are processed. The processing time of each chunk in every job is one unit of time. Each job J_i has a deadline d_i associated with it. The job J_i has to be completed before it's deadline i.e. d_i. In other words, all the chunks in the chunk set C_i have to be processed before the deadline d_i for all jobs J_i.
- **Constraints:** Several constraints have to be taken care of while formulating a solution for the problem.
  - **Deadline:** Each job J_i has to be completed before it's deadline d_i.
  - **Replication of Chunks:** Replication of data chunks in a node is allowed and once a data chunk is processed by a node, it can't be replaced by another data chunk.
  - **Data Chunk Bound:** Each node can only host up to B distinct data chunks. Once these B fill up, it can't replicate any other data chunk.
  - **Access of Data Chunk in VM:** Only one virtual machine can access a data chunk in a given time slot of the same physical machine.
- **Objective:** The objective is to minimize the total number of active nodes. A node is said to be active if it stores at least one data chunk and it is processed by at least one job.

1.2 Repository

The repository of the algorithms implemented can be found at : Repository

2 Main Algorithm

2.1 Algorithm

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.

We first formulate this problem as an *Integer Linear Programming Probem.* 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 as well as no two Vm's of the same node must access the same data chunk at a given time. Therefore, we can formulate the job scheduling problem in terms of required processing time for each chunk.

Denote $D^* = \bigcup_j \{d_j\}$ to be the set of D distinct deadlines. Without loss of generality, we assume the deadlines in D^* are ordered, so that $d_i^* < d_j^*$ for all i,j and $d_i^*, d_j^* \in D^*$. We now formulate the job scheduling problem with respect to the variable $f_{c,n,i}$ which is defined as the number of time slots on node n that are scheduled to process chunk c before the i^{th} smallest deadline d_i^* . We define $F_{c,i}$ as the minimum number of required time slots for chunk c before deadline d_i^*. Let $p_{c,n}$ be a binary chunk placement variable that is 1 if a chunk c is hosted by node n and 0 otherwise. Similarly, we use $u_n = 1$ to denote that node n is active and $u_n = 0$ otherwise. Let $q_{i,t,c,n}$ be a binary chunk placement variable that is 1 if a chunk c is hosted by node n and is processed on Vm i at time slot t.

- **Deadline Constraint**: $\sum_{n=1}^{\tilde{N}} f_{c,n,i} \geq \sum_{j:d_j<d_t} T_j = F_{c,i}, \forall c, d_t$
- **Date Locality Constraint**: Job i can be scheduled on node n only if the node is active and its required data chunks are available locally.
  $f_{c,n,i} = 0 \quad \text{if} \;\; (p_{c,n} = 0 \;\text{ or }\; u_n = 0), \; \forall c, n, d_t.$
  $\sum_{i=1}^{S} q_{i,t,c,n} \leq B \; \forall t, c, n.$
- **Virtual Machine Access Constraint**: For a specific type of chunk c
  $\sum_{i=1}^{S} q_{i,t,c,n} \leq 1 \; \forall t, n.$
- **Storage Constraint**: $\sum_{c \in C} p_{c,n} \leq B \cdot u_n, \quad \forall n, \; C = \bigcup_j \{C_j\}.$
- **Computational Constraint**:$\sum_{c:p_{c,n}>0} f_{c,n,i} \leq d_i^* \cdot S \cdot u_n, \quad \forall n, d_i^*.$

Objective: Our proposed optimization problem aims to minimize the total number of active nodes to process all jobs, under the above constraints.

- minimize $\quad \hat{N} = \sum_{n=1}^{N} u_n$
  s.t. all above constraints are satisfied.

Consider the chunk set C_i for $d_i^\uparrow$ with size C_i. We sort all chunks in descending order based on the number of required time slots for $d_i^\uparrow$ and record the order in an array, $R_{d_i^\uparrow}$. The chunk recorded in the head of $R_{d_i^\uparrow}$ has the largest number of required time slots for $d_i^\uparrow$.

Overview of the Algorithm: The algorithm can be divided into three major functions: **CRED-M**, **CRED-S**, **SCHEDULE**.

- **CRED-M:** In CRED-M, we iterate through the distinct deadlines in D^*. For each deadline d_i, we call CRED-S. We receive the nodes created by CRED-S and their schedules from it. Now, we iterate through all active nodes that have been created so far and schedule any chunks belonging to the deadlines greater than the current deadline i.e chunks such that their job J_j has deadline d_j ¿ d_i on these nodes.
- **CRED-S:** CRED-S receives a deadline d and a chunk set C that contains the chunks that haven't been processed completely and the number of timeslots they require to be processed and belong to the jobs whose deadline is d. We can break the algorithm into two cases here:
  - $B < S$: In this case, all of the S virtual machines in a node can never function together in a timeslot. This is because the atmost B chunks can be processed at once. So, the remaining $S - B$ virtual machines will be idle.
    In this case, we can schedule the first B chunks in $R_{d_i^\uparrow}$ as they require the highest number of time slots. Let $\mathcal{H}_b$ be the set of B chunks at the head of $R_{d_i^\uparrow}$. Then, $\sum_{c \in \mathcal{H}_b} F_{c,i}$ denotes the number of required time slots for the B chunks at the head of $R_{d_i^\uparrow}$. This sum is greater than or equal to $S * d$ which is the total number of timeslots available in a node. Thus, we try to schedule $\mathcal{H}_b$ by calling **SCHEDULE** and the scheduled chunks are removed from C.

    We continue this till C is empty and then return the nodes and their schedules to CRED-M.

  - $B >= S$**:** In this case, all virtual machines can function together in a timeslot. Thus, it is crucial that we choose the chunks in such a way that the availability of the virtual machines and data chunks is utilized well.

    Let $\mathcal{H}_{b,d}$ denote the the first set of B contiguous chunks, from the tail of the $R_{d_i}^{\uparrow}$, with the total number of required time slots larger than or equal to $S * d$. When $\sum_{c \in \mathcal{H}_{b,d}} F_{c,i} > S * d$, we place $\mathcal{H}_{b,d}$ into a node and call **SCHEDULE** for timeslots scheduling. The condition $\sum_{c \in \mathcal{H}_{b,d}} F_{c,i} > S * d$ guarantees that $\mathcal{H}_{b,d}$ exists. By choosing $\mathcal{H}_{b,d}$, we can schedule $S * d$ time slots in each node. Choosing $\mathcal{H}_{b,d}$ and calling **SCHEDULE** guarantees that we can take care of as many chunks as possible while scheduling $S * d$ 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 $\sum_{c \in \mathcal{H}_{b,d}} F_{c,i} <= S * d$, we can place any B chunks into one node and remove all of them.

    We continue this till C is empty and then return the nodes and their schedules to CRED-M.

- **SCHEDULE:** Schedule receives a node N, a chunk set C and deadline d. The task is to schedule as many chunks as possible in N till $t = d$.

  In this function, we iterate through all the chunks and check two conditions: does N have a copy of the current chunk?, is the number of data chunks at N less than B?. If any one of them is true, we go through each timeslot from $t = 0$ to $t = d$ and for each timeslot, we check if the current chunk is being accessed by any of the virtual machines. If not and if we find a virtual machine idle, we schedule this chunk there and decrement the time slots required by this chunk to completely process by one.

  Later, we remove the chunks that have been completely processed from C and return the remaining.

2.2 Pseudocode

```
Algorithm 1 CRED-S
Input: $d \leftarrow$ deadline, $C \leftarrow$ Chunk set used till deadline $d$ with number of time slots required for each type of chunk.
if $B < S$ then
    while (C is not empty) do
        Sort chunks in decreasing order of required time slots Select first $B$ chunks (or less) and schedule by calling Schedule function
else
    while (C is not empty) do
        Sort chunks in decreasing order of required time slots.
        if (time slots required for first $B$ chunks $> S \cdot deadline$) then
            Take $B$ chunks from tail of set with required time slots at least $S \cdot$ deadline Call Schedule function for these chunks.
        else
            Select first $B$ chunks (or less) and schedule by calling Schedule function.
Output: Number of active nodes required
```

```
Algorithm 2 Schedule
Input: $N^*$, Chunk set $C$ which will be scheduled in this node, $d \leftarrow$ deadline.
Sort chunk set $C$ with smallest time required first.
for (each chunk in C) do
    if chunk is already present in node or there is space to add new chunk then
        for $t = 0 \rightarrow deadline$ do
            Schedule this chunk at all time frames when no VM is using this chunk and update remaining chunks left to schedule in $C$.
Output: Scheduling jobs on $N^*$.
```

Algorithm 3 CRED-M

```
for each unique value ($d_i^*$) of deadline $D$ do
    Create a Job set, consisting of jobs to be scheduled with deadline ($d_i^*$).
    Call CRED-S and schedule these jobs.
    for each active node ($N^*$) formed till now do
        for $d = d^*_(i + 1) \to$ largest deadline do
            Find further job set with deadline $d$
            Call function Schedule with this job set to schedule all possible jobs on node $N^*$.
Output: Number of active nodes ($\tilde{N}$) required to schedule all jobs before deadline.
```

2.3 Complexity Analysis

2.3.1 Time Complexity:

We iterate through the set $(J_j, C_i))$, with Jobs corresponding to Chunk set C_i with lesser time slot requirement coming first. For every Iteration (for each tuple (J_j, C_i)) of the ***Schedule*** Algorithm:

- Try to schedule the $(J_j, C_i))$ in one of the time slot of the Vm's. $\to O(d_j^* \cdot S)$

For every Iteration of the ***CRED-S*** Algorithm:

- Sort the Chunk Set C in descending order of it's total requirement of time slots. $\to O(|C| \cdot \log |C|)$
- Select B or lesser chunks C accordingly to be scheduled. $\to O(|C| + B)$
- Try to *schedule* maximum amount of the remaining chunks from *Chunk Set*. $\to O(|C| \cdot d_j^* \cdot S)$

For every Iteration (deadline d_i) of the ***CRED-M*** Algorithm:

- Calculate the Total *Chunk Set* corresponding Jobs with deadline d_i. $\to O(|J_{di}| \cdot |C_{max}|)$
- Call ***CRED-S***. $\to O(|J| \cdot |C| \cdot \max(\log |C|, d_j^* \cdot S))$
- Iterate through each active node and try to *schedule* each of the Total *Chunk Set* corresponding Jobs with deadline $d_j \geq d_i$. $\to O(|J| \cdot |C| \cdot \max(\log |C|, (\max_j d_j^*) \cdot S))$

Hence **Overall Time Complexity** is $O(|D| \cdot |J| \cdot |C| \cdot \max(\log |C|, (\max_j d_j^*) \cdot S))$.

2.3.2 Space Complexity:

Overall we need to store the schedule of each active node upto d_{max} and the chunk set requirement of each Job. This will require $O(d_{max} \cdot S \cdot N + |J| \cdot |C_{max}|)$.

3 First Fit Algorithm

3.1 Algorithm

The *First Fit algorithm* is a classic heuristic approach used in various optimization and allocation problems, especially in the context of resource allocation. Its general principle is simple: when faced with a new item to allocate, it is assigned to the first available location (or container) where it fits.
Initially, all physical machines (nodes) are considered inactive, i.e., they do not store any data chunk and are not processing any job.
Iterate through each job in the give order. For each job, determine the set of data chunks it requires (C). Go through all the current existing nodes and try to allocate (i.e., the node has enough capacity and VMs to process the job) as many chunks we can. If there still exists requirement of more nodes, then allocate the required number of nodes.
The output of the algorithm is the total number of active nodes required to process all the jobs while meeting their deadlines.

3.2 Pseudocode

Algorithm 4 `Schedule_ffa`

```
Input: $C$, $d$ $N^* \leftarrow 0$
for $k = 1$ to $N$ do
    Allocate maximum number of chunks possible to physical machine $N_k$ such that data locality and deadline constraints are not violated.
    Adjust the Chunk set $C$ accordingly.
while $C > 0$ do
    Create a new Node $N^+$.
    Allocate maximum number of chunks possible to a the new physical machine ($N^+$) such that data locality and deadline constraints are not violated.
    Adjust the Chunk set $C$ accordingly.
    $N^*$ += 1
Output: $N^*$
```

Algorithm 5 `First Fit Algorithm`

```
Input: $J$, $B$, $S$ $\tilde{N} \leftarrow 0$
for $k = 1$ to $m$ do
    $C$ = Chunk set of the Current Job ($J_i$)
    $d$ = Deadline of Current Job ($J_i$)
    $\tilde{N}$ += Schedule_ffa(C,d)
Output: $\tilde{N}$
```

3.3 Complexity Analysis

3.3.1 Time Complexity:

For Schedule Algorithm, we must go through the nodes and check for each time-slot in it's S Vm's and try to schedule it $\rightarrow O(|C| \cdot d \cdot S)$.
For every Iteration of the First Fit Algorithm:

- Calculate the chunk set C. $\rightarrow O(|C_i|)$
- Call *Schedule_ffa(C,d)*. $\rightarrow O(|C| \cdot d \cdot S)$

Hence **Overall Time Complexity** is $O(N \cdot |C_{max}| \cdot d \cdot S)$.

3.3.2 Space Complexity:

Overall we need to store the schedule of each active node upto d_{max} and the chunk set requirement of each Job. This will require $O(d_{max} \cdot S \cdot N + |J| \cdot |C_{max}|)$.

4 Greedy Algorithm

4.1 Algorithm

The main idea of the algorithm is to schedule those jobs first which require the minimum number of nodes to be created apart from the already existing nodes. This objective helps in maximizing the utility of the nodes created as we are looking for jobs that mostly use the exisiting nodes to process their chunks and it also helps in minimizing the number of active nodes as we are constantly looking for the unscheduled jobs that require the least number of nodes to be created.

Overview of the Algorithm:

- **Main Function:** In the main function, **ExtraNodes** is called on the unscheduled jobs and it returns the list of the number of nodes that have to be created to schedule each job before it's deadline on the existing nodes and the new nodes created. Then, we choose the job with the minimum number of the nodes to be created and schedule it.
  This process continues till all the jobs are scheduled.

- **ExtraNodes:** In this function, the list of unscheduled jobs is taken as input. Then, each job is scheduled on the existing nodes and if some chunks remain unscheduled, then new nodes are created accordingly to schedule them before the deadline. It is important to note that these new nodes are not actually created but simulated to find out the number of new nodes to be created to schedule this job. The existing nodes and their schedules remain intact and no change takes place. This is achieved by creating copies of the existing nodes and simulating this process on them. We store the number of extra nodes to be created for each job and return it to the main function.
- **Schedule:** Once the job to be scheduled is selected by the **Main** function after calling **ExtraNodes**, it is scheduled by calling **Schedule**.
  In this function, we iterate through each chunk and try to schedule it first on the existing nodes and if it not possible, we create a new node and schedule it. Now this new node is added to the list of existing nodes.

4.2 Pseudocode

Algorithm 6 `Greedy Algorithm`

```
Input: $J \leftarrow$ list of jobs containing the chunk set $C_i$, processing time $p_i$ and deadline $d_i$ for each job $J_i$
while ($J$ has unscheduled jobs) do
    Call ExtraNodes to get the number of new nodes to be created to schedule each unscheduled job
    Select the job with the least number of new nodes to be to be created to schedule it and call Schedule
Output: Active nodes and their schedules
```

Algorithm 7 `ExtraNodes`

```
Input: $J \leftarrow$ list of unscheduled jobs containing the chunk set $C_i$, processing time $p_i$ and deadline $d_i$ for each job $J_i$
for $i = 1$ to $|J|$ do
    $C$ = Chunk set of the Current Job ($J_i$)
        $d$ = Deadline of Current Job ($J_i$)
        $\tilde{EN}_i$ = Schedule(C,d)
        /* $\tilde{EN}_i$: the number of new nodes created to schedule a job */
Output: $\tilde{EN}$
```

Algorithm 8 `Schedule`

```
Input: $C \leftarrow$ Chunk set, $d \leftarrow$ of the job to be scheduled
for $i = 1$ to $|C|$ do
    Allot possible timeslots for the chunk $C_i$ on the exisiting nodes
    while ($C_i$ is not processed completely) do
        Create a new node
        Allot timeslots for the chunks $C_i$ in this node and decrement the time required to process this chunk. Add the newly created node to the list of exisiting nodes
Output: $\tilde{EN}$
```

4.3 Complexity Analysis

4.3.1 Time Complexity

In every iteration of **Greedy Algorithm**:

- **ExtraNodes** is called. In it's every iteration, for each job, for each chunk, the slots in each virtual machine in all nodes before the job's deadline are checked by calling **Schedule**. Let d^* denote the maximum deadline. Then, the complexity of **ExtraNodes** would be $O(|J||C||N||S * d^*|)$.
- Once the output of **ExtraNodes** is received, the list of jobs with the number of extra nodes to be created to schedule them is sorted in the increasing order of the number of extra nodes. The sorting complexity would be $O(|J| \log |J|)$.
- In **Schedule**, the slots of all virtual machines in all nodes before the job's deadline are checked for each chunk of the job. Thus, the complexity of **Schedule** would be $O(|C||N||S * d^*|)$.
- Combining everything, the complexity of a single iteration would be $O(|J| * \max(|C||N||S * d^*|, \log |J|))$.

Thus, the total time complexity of the algorithm would be $O(|J|^2 * \max(|C||N||S * d^*|, \log |J|))$.

4.3.2 Space Complexity

Overall we need to store the schedule of each active node upto d_{max} and the chunk set requirement of each Job. This will require $O(d_{max} \cdot S \cdot N + |J| \cdot |C_{max}|)$.

5 Earliest Deadline First (EDF) Algorithm

5.1 Algorithm Description

The Earliest Deadline First (EDF) algorithm is a scheduling heuristic that prioritizes jobs based on their deadlines. It aims to minimize the maximum lateness of jobs and ensure timely completion of tasks.
In EDF scheduling, jobs are selected or sorted based on their deadlines, with the earliest deadline being given the highest priority. This means that jobs with closer deadlines are scheduled first to ensure they are completed on time. The algorithm iterate according to shortest deadline and allocate resources for that job and adds a node if require and moves to next job .
Overall, the Earliest Deadline First algorithm is effective for scenarios where job deadlines are critical, and tasks need to be prioritized dynamically based on their urgency to ensure timely execution and efficient resource utilization

5.2 Pseudocode

Algorithm 9 *Schedule_sda*

```
Input: $C$, $d$ $N^* \leftarrow 0$
for $k = 1$ to $N$ do
    Allocate maximum number of chunks possible to physical machine $N_k$ such that data locality and deadline constraints are not violated.
    Adjust the Chunk set $C$ accordingly.
while $C > 0$ do
    Create a new Node $N^+$ Allocate maximum number of chunks possible to a the new physical machine ($N^+$) such that data locality and deadline constraints are not violated.
    Adjust the Chunk set $C$ accordingly.
    $N^*$ += 1
Output: $N^*$
```

Algorithm 10 Earliest Deadline First Algorithm

```
Input: $J$, $B$, $S$ $\tilde{N} \leftarrow 0$
Sort job set with shortest deadline first
for $k = 1$ to $m$ do
    $C$ = Chunk set of the Current Job ($J_i$)
    $d$ = Deadline of Current Job ($J_i$)
    $\tilde{N}$ += Schedule_sda(C,d)
Output: $\tilde{N}$
```

5.3 Complexity Analysis

5.3.1 Time Complexity

The sorting step to arrange jobs based on deadlines requires a time complexity of $O(|J| \log |J|)$. The subsequent steps are similar to those in the First Fit Algorithm, resulting in an overall time complexity of $O(|J| \log |J| + N \cdot |C_{max}| \cdot d_{max} \cdot S)$.

5.3.2 Space Complexity

The space complexity remains $O(d_{max} \cdot S \cdot N + |J| \cdot |C_{max}|)$, similar to the First Fit Algorithm.

6 Performance

In this section, we evaluate the performance of all the algorithms by creating a schedule for a testcase and then comparing them.

Let's say, $S = 3$, $B = 2$ and there are 4 types of Chunks. So, the entire chunk set would be $C = C_0, C_1, C_2, C_3$ We have to schedule 4 jobs. Description of each job is given below:

1. J_0: Processing time = 6, Deadline = 3, Chunk set = C_0, C_2
2. J_1: Processing time = 3, Deadline = 4, Chunk set = C_0
3. J_2: Processing time = 5, Deadline = 5, Chunk set = C_0, C_1, C_2
4. J_3: Processing time = 5, Deadline = 5, Chunk set = C_0, C_2

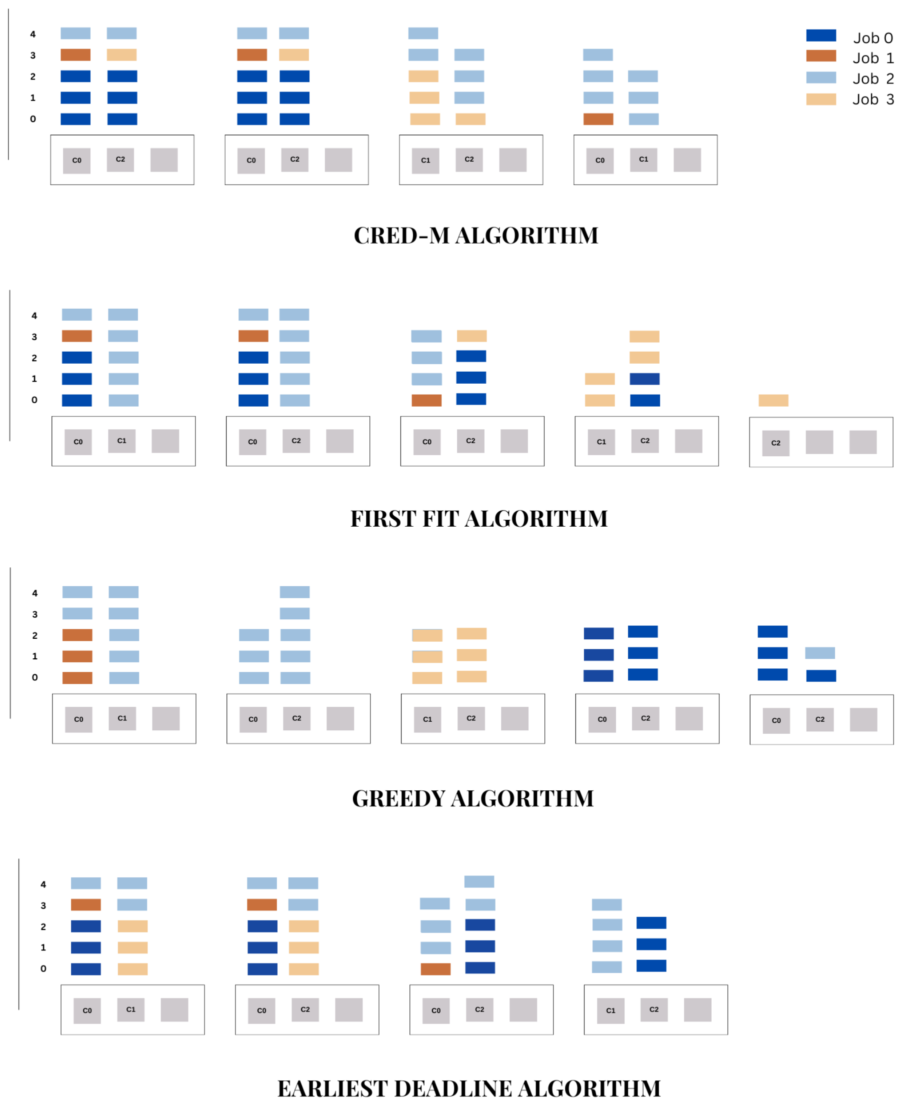

Figure 1: Schedules of all the algorithms