CS528 :- High Performance Computing
Winter Semester 2023-24

Placement and Scheduling Data Intensive Jobs in Edge-Cloud System

A COMPREHENSIVE REPORT

This report introduces a comprehensive approach to addressing the challenge of Placement and Scheduling Data Intensive Jobs in Edge-Cloud Systems. It provides an overview of our strategy to optimise job allocation and scheduling within these complex environments, aiming to maximise efficiency and resource utilisation.

Jethwani Siddhant Pratap
210101053

Vatsal Jain
210101110

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 $\mathbf{d_j}$ and is required to access a set $\mathbf{C_j}$ of equal-sized chunks, we can assume each job j has $\mathbf{|C_j|}$ number of tasks accessing one data chunk each and can be run in parallel on the same machine or different machine.

The chunks are stored in a distributed file system on the cloud. Each node is capable of hosting up to **B** data chunks and is equipped with **S** virtual machines which implies each node is able to simultaneously process S jobs. Let $\mathbf{C} = \cup \mathbf{C_j}$ be the set of all data chunks available at the central storage server, before processing the request we need to bring the data chunk to the physical machine.

Replication of data chunks in different machines is allowed and data chunk needs to be placed only once at the beginning (before any job starts execution, once any one job starts the execution, we are not allowed to change the data placement) and during run time we can not 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.

Completing a job j before the deadline d_j is equivalent to processing all the required chunks c of C_j before the deadline. Only one VM can access a data chunk in a given time slot of the same physical machine. The problem aims to minimise the **total number of active nodes ($\mathbf{N_a}$)** to process all the jobs. The active node means the node stores at least one data chunk and is processed by at least one job. The number of active machines is always lower than the total number of machines which is $\mathbf{N_a < N}$.

NP-Completeness of the Problem

- **Scheduling Constraints -** Involves scheduling jobs with specific deadlines on physical machines with constraints like virtual machine capacity and chunk processing capability.
- **Data Chunk Allocation -** Allocating data chunks to machines before job execution adds complexity, akin to problems like graph colouring and bin packing.
- **Deadline Constraints -** Ensuring job completion before deadlines adds a temporal dimension, crucial for feasibility and optimality.
- **Replication of Data chunks -** Allowing replication across machines complicates decisions on chunk placement to minimise completion time.
- **Non-deterministic Polynomial Time Complexity** Verification is polynomial, but finding an optimal solution may require exponential time due to the vast solution space.
- **Combinatorial Nature -** Decisions on job and chunk allocation, considering various constraints, contribute to the problem's combinatorial complexity.
- **Trade-off between Resource Utilisation and Job Completion -** Balancing resource utilisation while meeting deadlines poses a classic trade-off, contributing to NP-completeness.

Proposed Heuristic Greedy Algorithm

- **Input Data Handling -** The algorithm begins by taking input data from the user via terminal, including the total number of chunks, total number of jobs, chunk assignments to each job, deadlines for each job, and machine constraints such as the maximum number of chunks per machine (b) and the number of simultaneous jobs a machine can process (s).

- **Binary Search for Minimum Machines -** A binary search is applied on the number of machines to determine the minimum number of machines required for scheduling. The search space ranges from 1 (minimum) to the total number of chunks assigned to different machines (maximum).

- **Check Scheduling Function -** For a particular number of machines, the **`checkScheduling`** function is employed to determine if a valid assignment of chunks to machines exists, satisfying all deadlines and constraints. The function utilises several data structures :-
  - **`assignedChunksForMachine`** – Stores the assignment of chunks to machines for a specific number of machines. (`vector<set<long long int>>`)
  - **`assignedAt`** – Tracks the chunks being executed by each machine at a particular timestamp. (`vector<vector<set<long long int>>>`)
  - **`busyVirtualMachines`** – Records the number of busy virtual machines at each timestamp. (`vector<vector<long long int>>`)

- **`chunksPlacementForMachine`** – Stores scheduling details including timestamps, machines, jobs, and chunks. (`vector<vector<long long int>>`)
- **`finishTimesForMachine`** – Tracks the finish times of jobs when executed with a specific number of machines.
  (`vector<pair<long long int,long long int>`)
- **`chunksPlacement`** – This structure stores {timestamp, machine, job, chunk} vectors for scheduling with a minimum number of machines.
  (`vector<vector<long long int>>`)
- **`assignedChunks`** – It holds the final assignment of chunks to machines for optimal scheduling.
  (`vector<set<long long int>>`)
- **`finishTimes`** – This structure records the finishing times of all jobs for evaluation.
  (`vector<pair<long long int, long long int>>`)

- **Scheduling Procedure -** Our proposed scheduling procedure is as follows :-

  - The algorithm iterates over all possible timestamps from **`t=1`** to **`maxDeadline`** (deadline of the job that finishes last)
  - For each unsatisfied job, the algorithm attempts to assign its unassigned chunks to available machines within the specified deadline.
  - Jobs are prioritised based on their deadlines, with earlier deadlines receiving higher priority.
  - Chunks are assigned to machines based on the availability of idle virtual machines and the timestamp constraints.

    - If a job is satisfied (all chunks assigned and executed before its deadline), the algorithm proceeds to the next unsatisfied job.

- **Output Generation -** The required output of the program is generated as follows :-

    - If all jobs are satisfied at some point during the scheduling process, the algorithm returns true, indicating the existence of a valid scheduling solution.
    - The algorithm stores the scheduling details including chunk assignments to machines and finish times of jobs.
    - A '`schedule.txt`' file is generated containing detailed scheduling information for further analysis.

- **Main Function Execution -** The main function executes as follows :-

    - The main function orchestrates the entire scheduling process, including input data acquisition, binary search for minimum machines, scheduling verification, and output generation.
    - If a valid scheduling solution is found, the algorithm outputs the minimum number of machines required and generates a '`schedule.txt`' file with scheduling details.

Strategies behind our Greedy Algorithm

The greedy algorithm's prioritisation of jobs based on deadlines, coupled with iterative chunk allocation and resource utilisation optimization, is expected to lead to efficient scheduling. The following strategies have been used :-

- **Deadline-oriented Scheduling** - The algorithm prioritises jobs based on their deadlines, aiming to complete those with earlier deadlines first. By focusing on meeting deadlines, it aims to minimise the risk of missing important job completions.

- **Chunk Allocation Optimization** - The algorithm optimises the allocation of data chunks to machines by iteratively assigning chunks to available machines, ensuring that each chunk is processed efficiently within the specified deadline. This approach helps in maximising resource utilisation and minimising idle time.

- **Iterative Improvement** - The algorithm iterates through each job and attempts to schedule its chunks on available machines, refining the scheduling decisions iteratively. This iterative process allows the algorithm to make local decisions that contribute to a globally optimal solution.

- **Resource Utilisation Consideration** - By considering factors such as the maximum number of chunks per machine and the number of simultaneous jobs a machine can handle, the algorithm aims to balance resource utilisation across machines. This helps in avoiding resource bottlenecks and maximising overall system throughput.

- **Dynamic Adaptation to Constraints** - The algorithm dynamically adapts to machine constraints and job requirements during the scheduling process. It explores various scheduling options based on the current state of available resources, making decisions that lead to efficient resource allocation.

- **Incremental Deadline Achievement** - By scheduling jobs incrementally based on their deadlines, the algorithm gradually works towards achieving all deadlines while minimising the total number of active nodes. This incremental approach helps in managing computational complexity and improving scheduling efficiency.

- **Greedy Strategy** - The greedy strategy of prioritising jobs with earlier deadlines and iteratively assigning chunks to machines based on current availability aims to achieve a locally optimal solution at each step. Although not guaranteed to find the globally optimal solution, this strategy often leads to satisfactory results in practice.

Time & Space Complexity Analysis

Time Complexity

- **Sorting Deadlines** - Sorting the deadlines takes *O(JlogJ)* time, where J is the number of jobs.

- **Binary Search** - The binary search for the minimum number of machines takes *O(log C)* time, where C is the sum of chunks across all the jobs.

- **Check Scheduling Function** - The overall complexity of this function is discussed as follows :-

    1. **Outer loop** :- The outer loop iterates over all the deadlines, ranging from 1 to the max deadline. In the worst case, this loop runs *O(D)* times, where D is the maximum deadline.
    2. **Inner Loop** :- Within each iteration, these are nested over all jobs, machines, and timestamps, with a maximum combined complexity of *O(J.C.D.log(C))*.

- **Total Time Complexity** - Considering all operations, the overall time complexity is *O(J log J + logC . J . C . D)*. Since J, C and D are parts of the input size, and all other operations within the algorithm are polynomial in terms of these inputs, the overall time complexity of this algorithm is polynomial.

Space Complexity

- **Data Structures**

    1. **chunksPlacement** - Requires space proportional to the number of valid scheduling entries, which could be up to *O(J.C.D)*, where J is the numbers of jobs, C is the sum of chunks over all machines and D is the maximum deadline for any job.
    2. **assignedChunks** - Stores the final assignment of chunks to machines, with space proportional to the number of machines M, this requiring a maximum of *O(M.n)* space, where n is the average number of chunks assigned to a machine.
    3. **finishTimes** - Stores the finish times for all jobs, requiring space proportional to the number of jobs J, this requiring *O(J)* space.
    4. Additional data structures like queues, sets and vectors used in the checkScheduling function contribute to the space complexity.

- **Total Space complexity** - Considering all the data structures and auxiliary space, the overall space complexity is *O(J.C.D + M*n + J)*. Since J,C, and D are part of the input size, and all other auxiliary data structures require polynomial space, the overall space complexity is polynomial.

Conclusion

In summary, this project tackles the complex task of scheduling jobs on a cloud platform with strict deadlines and limited resources. We've devised a heuristic algorithm that relies on clever strategies and utilises optimised data structures to efficiently handle scheduling. Our approach aims to minimise the number of active nodes while ensuring all job requirements are met.

The strength of our algorithm lies in its ability to swiftly find a nearly optimal solution within a reasonable timeframe, making it applicable to real-world cloud computing scenarios. Through thorough testing and analysis, we've confirmed the algorithm's performance and established its polynomial time complexity, indicating scalability for large-scale scheduling tasks.

Moreover, our project sheds light on the challenges of resource allocation and scheduling in cloud environments, highlighting the delicate balance between resource usage, job completion times, and algorithmic intricacies. By documenting the algorithm's design, implementation, and performance metrics, we've created a valuable resource for both researchers and practitioners in the cloud computing and optimization fields.

Overall, this project contributes to advancing the development of efficient scheduling algorithms for cloud environments, offering potential applications across various industries where resource allocation and optimization are key factors in achieving success.