P4: Task Scheduling in Edge Server Systems with limited Solar Energy and Infinite Batteries

Group members-

1. Achintya Gupta- 210101005
2. Kannan Rustagi- 210101054
3. Kshitij Maurya- 210101059

Introduction to the Problem:

The problem states that we have multiple machines, let us say N and we have a certain number of fixed time slots in a day, say T. Certain amount of power will be generated in each machine(m) in each time slot(t), say S[m][t] and certain number of tasks will arrive at in each machine(m) in each time slot(t), say D[m][t]. S[m][t] and D[m][t] values are known beforehand.

Constraints and description:

- The amount of power consumed in a server in a time slot is equal to the cube of the tasks done on the server.
- S[i][j] <= S where S is fixed. But infinite battery power can be stored.
- Tasks can be migrated to a different machine in a given time slot at no cost but not across time slots.
- Energy cannot be transferred across machines but can be stored for use in later time slots.
- Compute capacity is infinite.

Objective:

To maximize the total number of task executions for the entire day(across all time slots) by all the edge servers.

Solution Approach:

It is a common observation that the cubic function grows very quickly. Hence, 3 tasks scheduled on 3 different machines will save a lot of energy as compared to scheduling them in a single machine. Hence, this saved energy can then be used in later time slots. Moreover, the cost of migration of tasks in a time slot is 0, hence, tasks can be migrated without considering which machine they originally came from.

Similarly, even if more tasks arrive in a particular time slot, it may not be wise to schedule all the tasks in the given time slot, as reducing even a single task executed can lead to many more tasks being executed in a later time slot.

This demands an equitable distribution of tasks both across machines and across time slots, thereby, trying to be energy efficient, despite the objective being to maximize the number of tasks. It is best to not be greedy.

First, to distribute tasks equitably across machines, we thought to assign an equal number of tasks to each machine (or as much as they could if they had insufficient energy). However, this raised problems as there was a possibility of not scheduling some tasks to maximize the tasks executed at a later stage.

We resorted to distributing one task at a time. Despite the seemingly bad complexity, it has a surprisingly closed-bound. It works well, with most of the tasks always being scheduled.

1. **Data Structures Used:**
    - 2D Arrays: The core data structure used for storing the state of the system at any iteration are 2D arrays. These arrays store information such as energy usage (`Prev_Pref_Used[m][t]`, `Curr_Extra_Pref_Used[m][t]`), task scheduling decisions (`D_Sched[m][t]`), and energy availability (`Prev_Suff_Left[m][t]`, `Pref_S[m][t]`). Each element in these arrays represents the state of a machine at a specific time slot.

    - Sets and Pairs: Sets and pairs are utilized to efficiently manage and track available energy and scheduling decisions. Pairs of integers (`pair<int, int>`) represent machine-time slot combinations, facilitating iteration over all possible scheduling options. Sets (`set<pair<int, pair<int, int>>>`) are employed to store and update information about available energy and scheduling decisions. To schedule tasks we iterate over this set which basically contains the set of machines and slots where it is possible for more tasks to be scheduled at any iteration. The set is ordered by the time and then by the energy available in the machine at that slot since it is advantageous to start scheduling tasks from the minimum energy available machine first.

2. **Algorithm:**
    - Calculating Available Energy
        i. The algorithm calculates the available energy for task scheduling in each time slot and machine.
        ii. It considers the power generation data (stored in the `S` array) to determine the energy available at each machine for each time slot.
        iii. By tracking the total energy used till the previous iteration (stored in the `Prev_Pref_Used` array), the algorithm ensures that the available energy reflects the current state of energy consumption.
        iv. During each iteration, the algorithm updates arrays tracking energy usage (`Prev_Pref_Used`, `Curr_Extra_Pref_Used`) based on task scheduling decisions. It calculates and accumulates the extra energy used for scheduling additional

tasks and updates the total energy usage for each machine at each time slot.

- Determining Extra Energy Needed:
    i. The algorithm computes the extra energy needed for scheduling additional tasks in each time slot and machine.
    ii. It calculates the difference between the energy required for the current number of scheduled tasks and the energy required for scheduling one additional task.
    iii. This calculation is based on the constraint that the amount of power consumed in a server in a time slot is equal to the cube of the tasks done on the server.
- Energy Constraints and Optimization:
    i. The algorithm considers energy constraints to ensure that scheduling additional tasks does not reduce energy below 0 in any later time slots.
    ii. To ensure that the energy being used to schedule extra tasks in the current time slot doesn't lead to a scenario wherein in future slots, there isn't sufficient energy left to schedule the tasks that were previously allotted to that machine, it considers the remaining energy in subsequent time slots(stored in the `Prev_Suff_Left` array).
    iii. By optimizing task scheduling based on available energy and considering future energy needs, the algorithm aims to maximize task execution without risking energy shortages.
- Dynamic Adjustment of Scheduling Strategy:
    i. The algorithm dynamically adjusts its task scheduling strategy based on available energy and task requirements.
    ii. It prioritizes scheduling tasks in time slots where there's sufficient available energy, while also considering the potential impact on future energy availability.

Pseudo Code

```
1. Create a vector D_New of size T to store the total number
   of tasks in each time slot.
2. Create 2D vectors to store various information related to
   task scheduling.
3. Calculate the Pref_S matrix which stores the cumulative
   sum of energy received by each machine at each time slot.
4. Initialize Prev_Suff_Left matrix with the Pref_S matrix.
5. Create a list Machine_Available with pairs representing
   available machines and time slots.
```

6. Enter a loop until Machine_Available is not empty.
    a. Clear the set Current_Machines.
    b. Clear the list Machine_Available_Next.
    c. Update Prev_Suff_Left matrix.
    d. Iterate over Machine_Available.
        i. Calculate available energy for the current machine and time slot.
        ii. Add the current machine and time slot with available energy to the Current_Machines set.
    e. Iterate over Current_Machines.
        i. Try to schedule tasks based on available energy.
        ii. Update relevant matrices and lists.
    f. Update Machine_Available to Machine_Available_Next.

Time Complexity and Bound Analysis

The primary contributing factor to the high time complexity of the solution approach are the nested for loops, which iterate over machines and assign tasks one at a time. If there are M machines and T time slots, with each slot receiving a maximum of S units of energy.

The innermost loop, which updates the suffix matrix, has a time complexity of O(MT). Meanwhile, the loop that iterates over each slot of each machine has a worst-case time complexity of O(MTlog(MT)) because a 'set' data structure is used in the loop. Using this information we can say that each iteration of the outer loop takes O(MTlog(MT)) in the worst case scenario.

Now we need to calculate exactly how many times would the outermost loop run. Using a worst case analysis approach for it as well, we see that a single machine will at most receive S*T units of energy over all time slots. The outer loop would run the maximum number of times if all this energy is being used in one particular slot of that machine. Since we are assigning one task to a slot at a time, this would result in $(ST)^{1/3}$ iterations of the outer loop in the worst case as the energy utilized in a particular slot is equal to cube of the number of tasks scheduled in the slot. There will be no more iterations than this as if we are not able to schedule a task in a particular slot of a machine, it is clear that we will not be able to do so in the future iterations as well. We deal with this by using the `Machine_Available_Next` vector to find the slot in which we may be able to schedule a new task. This results in our outer loop terminating once it is not possible to schedule a new task in any slot of a machine.

Using the information above we can conclude that the solution approach provided above takes **$O((ST)^{1/3}MTlog(MT))$** time in the worst case.

Observations and Analysis of Different Kinds of Test Cases

1. Medium Load:

```
Tasks received at each slot at each machine:

3 4 1 2 4 0 3 8 8 4
2 3 3 0 7 4 7 2 4 0
4 6 4 2 1 6 1 4 2 4
3 2 0 4 4 8 0 1 4 4
3 4 8 0 7 3 1 1 3 8
5 4 3 6 1 4 3 1 0 0
1 2 2 7 1 0 0 6 3 6
0 4 4 3 2 1 6 7 2 1
1 0 8 3 3 7 4 1 0 0
4 5 6 2 4 1 1 8 2 1

Tasks scheduled at each slot at each machine:

2 4 5 3 4 3 3 4 3 3
3 3 4 3 4 4 2 4 3 3
2 3 4 3 3 4 3 5 3 3
2 4 4 3 3 4 2 4 3 3
2 3 4 3 3 3 3 3 3 3
3 3 4 2 3 3 2 4 2 2
3 4 3 3 3 2 3 3 3 3
3 4 3 3 4 4 3 3 3 3
3 2 3 3 4 4 3 5 3 3
3 4 5 3 3 3 2 4 2 2
```

Total tasks received: 317
Total tasks completed: 317

2. Light Load

```
Tasks received at each slot at each machine:

0 1 4 1 3 3 4 4 1 2
3 0 1 0 0 1 3 4 2 1
4 3 4 0 2 3 2 1 0 4
3 1 3 1 2 3 2 2 4 1
4 1 1 2 4 1 3 4 0 1
1 4 3 4 2 1 3 2 3 3
0 3 2 4 1 4 2 1 3 4
2 2 3 2 1 4 2 4 4 2
4 2 1 4 2 1 3 1 1 2
3 4 0 4 4 1 0 4 0 3

Tasks scheduled at each slot at each machine:

3 2 2 2 2 2 2 3 2 3
3 2 2 2 2 2 2 2 1 2
2 1 3 3 3 3 3 3 2 3
2 2 2 2 2 2 2 3 2 2
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 3 3 2 2
2 3 2 2 2 2 2 3 2 2
3 2 2 2 2 2 2 2 1 2
3 3 3 2 2 3 3 3 2 3
2 2 2 3 2 2 3 3 2 2
```

Total tasks received: 224
Total tasks completed: 224

3. Occasionally Heavy Load

```
Tasks received at each slot at each machine:

2 3 2 2 2 5 2 2 1 2
5 2 3 2 3 1 1 4 3 2
1 2 3 4 2 1 12 3 1 4
2 1 2 0 10 4 9 1 3 2
2 2 2 1 2 3 5 1 9 10
3 1 2 3 12 6 4 2 1 1
2 1 3 2 3 5 5 2 10 3
4 11 1 10 1 3 5 3 1 2
3 2 11 3 4 4 2 5 3 5
3 3 3 3 1 10 3 2 1 1

Tasks scheduled at each slot at each machine:

3 2 3 3 4 4 5 3 3 3
3 3 3 3 4 4 4 3 3 3
3 3 3 3 4 4 6 2 3 3
3 3 4 3 4 4 4 3 3 4
2 3 3 3 4 4 5 2 4 3
3 3 3 3 4 5 4 3 4 4
2 2 3 3 4 4 6 2 3 3
3 3 3 3 4 4 5 2 3 3
3 3 3 3 4 4 4 3 4 3
2 3 4 3 4 5 4 2 3 3
```

Total tasks received: 337
Total tasks completed: 336

4. Very Heavy load

```
Tasks received at each slot at each machine:

4 6 8 5 1 2 2 9 7 5
11 2 10 10 4 1 9 3 4 1
3 5 4 1 4 1 2 4 10 6
3 2 5 2 6 2 11 10 3 4
3 4 8 5 4 5 6 5 5 2
5 5 4 2 1 8 1 7 5 2
3 11 1 4 2 5 6 1 4 9
7 8 2 6 11 2 3 5 5 1
5 1 4 4 5 8 5 1 6 4
3 5 3 5 10 9 3 2 7 5

Tasks scheduled at each slot at each machine:

4 4 4 5 4 3 4 3 4 4
2 4 4 4 3 4 4 5 4 3
4 3 3 4 3 3 3 3 3 3
3 3 2 2 4 4 5 4 3 4
4 3 4 4 3 3 3 5 4 4
3 4 5 4 4 4 4 4 4 3
2 4 3 4 4 4 4 4 4 3
3 4 4 5 4 4 4 4 3 3
4 3 4 4 4 3 3 4 3 4
2 4 3 4 4 4 4 5 4 4
```

Total tasks received: 470
Total tasks completed: 366

Conclusion:

In all the above examples, it is visible how the load is distributed uniformly among both dimensions, i.e, an almost uniform number of tasks are scheduled in any machine across all time slots, and also, a uniform number of tasks are executed in all machines across a single time slot. This was our main aim while designing the algorithm, and this fact allows the majority of tasks to be executed in all cases, thus achieving the goal.