Greedy based Genetic Algorithm and its evaluation

Solution for Scheduling Jobs across Geo-Distributed Data Centers

Jianxing Qin (519030910270, qdelta@sjtu.edu.cn), Zhuoren Liang (519030910231, Liang_ZR@sjtu.edu.cn), Yixin Zhang (519030910273, yixin2001@163.com)

Department of Computer Science,
Shanghai Jiao Tong University, Shanghai, China

Abstract. To solve the geo-distributed data center scheduling problem, we first extracted useful information from the requirements to make a formalization. We designed the $\boldsymbol{cost-function}$ to calculate the processing time and evaluate the performance. With the evaluation, we proved it is an NP-complete problem. Then we design an algorithm with greedy method to optimize the performance. The algorithm is tested on the toy data. Then we generated data of various characteristics to examine the performance. Lastly, we discussed the result and the application of the algorithm.

Keywords: Geo-distributed Data Center Scheduling, NP-Complete Problem, Genetic Algorithm, Greedy Approach, Directed Acyclic Graph Algorithm.

1 Introduction

1.1 Background

It has become increasingly common for large scale data to be formed and processed in a geographically distributed pattern. Especially in the field of big data processing, there has been more international corporations working in this field. Processing among several data centers has its advantage of concurrency and the variety and sufficiency of data. We can predict that the researching field have a promising future. However, the problem is restricted by the distribution and capacity of data centers and transferring bandwidth. Therefore, it is greatly significant to construct a model and then forward an feasible algorithm to keep a balance between efficiency and fairness of the jobs.

1.2 Our Work

In order to solve the problem, we did the following work, which is shown in 1.

Firstly, in order to depict the dependency of the tasks of every job, we transfer each job into a Directed Acyclic Graph(DAG). In this step, the problem can then be reduced to a DAG planning problem: How to match DAGs with the data centers and slots.

When we are matching computational resources to DAG models, we will reach Max-Min fairness by allocating the computational resources to the jobs fairly. To improve the performance of the algorithm, we adopt the greedy method. As for the running time, we can apply the genetic algorithm. We will use the given data and generate some data to verify the correctness and efficiency of the algorithm and then visualize the result.

2 Fundamental Assumptions

To simplify the problem, we make the following basic assumptions, each of which is properly justified:

- The calculation of the data slots is absolutely correct and plausible.
- The server of the network has no processing capacity limit and will not crash during the process of computing the data.
- The bandwidth limitations are for each connection, and data can be transferred though a path with multiple data centers.

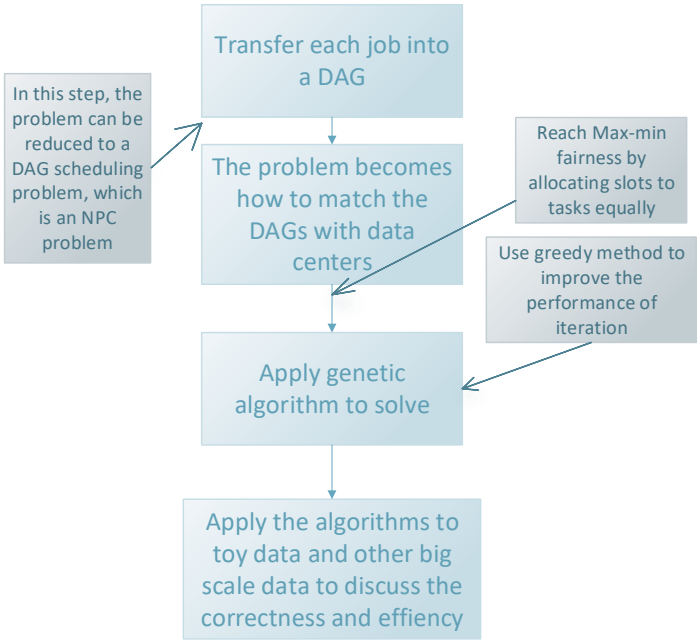

Fig. 1. The Flow Structure of Our Work

3 Notations and Formalization

3.1 Notations

We list all the symbols we use in the following table:

Table 1. Notations

Symbol	Definition
n	number of jobs
m	number of date centers
U	number of all tasks
V	number of all slots
$J_1, J_2, \dots J_n$	jobs to be done
$C_1, C_2, \dots C_m$	data centers
D_k	process task set of the k-th data center
n_i	number of tasks of the i-th job
m_i	number of slots of the i-th data center
s_{ij}	the j-th slot of the data center C_i
q_{ij}	the task queue of s_{ij}
t_{rs}	the r-th job's s-th task to be calculated in a computation slot
t'_{rs}	the r-th job's s-th task together with its transferring time, a node in the DAG of $J_r$placed in a certain q_{ij} queue

3.2 Formalization

Firstly, we use the Directed Acyclic Graph(DAG) model to describe the dependency of tasks in one job. Since we combine the transferring time in the calculating time, t'_{ij} will be the processing and transferring workload for one task. Let J_i represent the DAG of the i-th job. The tasks in the graph are vertices $t'_{i1}, t'_{i2}, \dots, t'_{in_i}$ where n_i is the task number of the i-th job. If there is a directed edge from t'_{ir} to t'_{is}, then the processing of t'_{is} relies on the result of $t'_{ir}(1 \le r, s \le n_i)$.

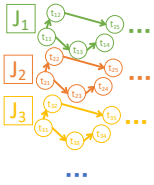

Fig. 2. Jobs and Corresponding DAG

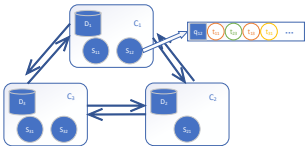

Fig. 3. A Possible Arrangement

Secondly, the problem is to find an possible arrangement of assigning t'_{ij} to corresponding slots in C_i, let the arrangement satisfy the dependency requirement. For every arrangement of q_{ij}, we are going to let it qualify the restrictions generated by the **cost-function** 1, which is actually a simulator to evaluate the maximum finish-time of all jobs.

Algorithm 1: COST-FUNCTION

Input: the arrangement of every q_{ij}
Output: the processing time of all the jobs

$T_{current} \leftarrow 0$;
priority queue $q_{time} \leftarrow \emptyset$;
$completed \leftarrow \emptyset$;
while *True* **do**
  $m \leftarrow pop(q_{time})$;
  $completed \leftarrow completed \cup \{m\}$;
  forall q_{ij} **do**
    $t \leftarrow first(q_{ij})$;
    if *all dependents of* t *are completed* **then**
      $pop(q_{ij})$;
      $push(q_{time}, T_{current} + finish_time(t))$;
  if $q_{time} = \emptyset$ **then**
    break;
if *all* q_{ij} *are empty* **then**
  return $T_{current}$;
else
  return ∞;

Intuitively, the simulator maintains a priority queue of executing tasks and their finishing time, each time pop a task with the minimum finishing time and try to push other tasks in the task queues.

To evaluate the finishing time, we sum up the current time, the transferring time and the execution time. The current time and the execution time is known while the transferring time need to be evaluate since our bandwidths are going to be modified according to our assumptions.

In our assumption, the data can be transferred though a path with multiple data centers, which means the bandwidth will be the maximum path bandwidth between the source and sink, with the path bandwidth be the minimum direct bandwidth of each edge.

Using Floyd-Warshall algorithm 2 we can obtain the final bandwidth between data centers.

Then the transferring time will be the amount of transferring data divided by the bandwidth between corresponding data centers.

4 NP-Complete Proof

4.1 Qualitative Definition

According to our formalization of the problem, we can conclude that the problem is an NP-Complete Problem.

Algorithm 2: Floyd-Warshall Algorithm

Input: the original bandwidth *band*
Output: the final bandwidth *band*

```
for k ← 1, ..., size(band) do
    for i ← 1, ..., size(band) do
        for j ← 1, ..., size(band) do
            band[i][j] ← max(band[i][j], min(band[i][k], band[k][j]));
return band;
```

4.2 Certificate and Certifier

We define the certificate and certifier as follow:

Certificate: An arrangement of all the tasks to corresponding slots, all the processed data to corresponding transferring links. The arranging order in each slots.

Certifier: run the **cost-function** algorithm 1 to get a result.

4.3 Proof

***Step* 1** First we prove that the problem is a NP Problem. The running process of our algorithm is to push and pop elements in the queue. The data of the tasks will only enter the queue q_{ij} for once to be processed. Since there is sufficient links, the processed data will be transferred at most once. We can see that for each data element, it will be operated by the queue for at most twice. The comparison between bandwidth and data will take linear time. So we can conclude that we can check the result in polynomial time. The problem is NP.

***Step* 2** Since there are more than 1 jobs, for one job, the execution of it is a DAG solution problem. According to the documents we have looked through(can be seen in the References [1]), DAG is a NP-Complete Problem. Apparently, multiple job DAG scheduling is more complicated than single job DAG scheduling. So we can conclude that

$$DAG - SCHEDULING \leq_p MULTI - SCHEDULING\ ACROSS\ DATA\ CENTERS$$

5 Algorithm Design

To find the solution of the problem, we designed a genetic algorithm, reaching the max-min fairness when assigning slots to jobs. And use greedy method to improve the performance.

5.1 Max-min Fairness implementation

To give a intuition of our max-min fairness, we take the utilization of each computational slot as resources, which means we will assign tasks fairly to every slot, resulting in a fair distribution of assigned tasks among data centers according to the computing power.

Assuming that the numbers of tasks in queues of all slots are $k_1, k_2, \cdots, k_V \in \mathbb{N}$, we reach the max-min fairness by satisfying following constraints:

$$\begin{cases} \sum_{i=1}^{V} k_i = U \\ \max_{1 \leq i \leq V}\{k_i\} - \min_{1 \leq i \leq V}\{k_i\} \leq 1 \end{cases}$$

It is trivial that multi-set $K = \{k_1, k_2, \cdots, k_V\}$ will have at most 2 different elements, so we can design a Fair-Dispatch algorithm 3

Algorithm 3: Fair-Dispatch

Input: the number of tasks U and number of slots V
Output: f, fn, c, cn where K have fn multiple f and cn multiple c

$f \leftarrow \lfloor U/V \rfloor$;
$c \leftarrow \lceil U/V \rceil$;
$cn \leftarrow U - V \times f$;
$fn \leftarrow V - cn$;
return f, fn, c, cn;

5.2 Genetic Algorithm to Solve the Slots Arrangement Problem

Since this is an NP-Complete problem, while iterating all possible parameters can result in a optimal solution, the cost is usually unacceptable. So we design a genetic algorithm to get a sub-optimal solution.

The idea of Genetic Algorithm(GA) is that we directly operate on the objects in the group, using probability methods to optimize the results to obtain a better performance. [2] Genetic algorithms are randomized search algorithms that have been developed in an effort to imitate the mechanics of natural selection and natural genetics. It usually works on string structures, which requires us to transfer the graph problem into a sequence aligning problem. It works on parameter set, not themselves. It works on a population of strings, not focusing on a single point. The GA works on the payoff of the data, not caring about derivatives. GA also uses probabilistic methods to generate new individuals, not deterministic methods.

About the steps of GA, we can describe generally as follows. At first, the coding to be used must be defined. Then using a random process, an initial population of strings is generated. Next, we generate some offsprings randomly. Using the selection method, strings with higher fitness value have bigger probability of contributing offsprings to the next generation. Crossover is a process in which members of the last population are mated at random in the mating pool to hopefully keep the advantage of both their parents. Mutation is the occasional random alteration of the value of a string position.

For example, the set of all the permutations is the population, we randomly change the order of the tasks in the queue, which is the mutating in the genetic algorithm. After we get some new permutations, we keep some fragments in them and exchange the rest, which can be considered as crossover. We set a rule of fitness, eliminate the individual permutations that are not fit enough. We call this process of mutation, crossover and selection as a generation. Repeating for many generations is the process of natural selection. After a number of generations, we can get the whole group more suitable to the environment, in other words, less processing time.

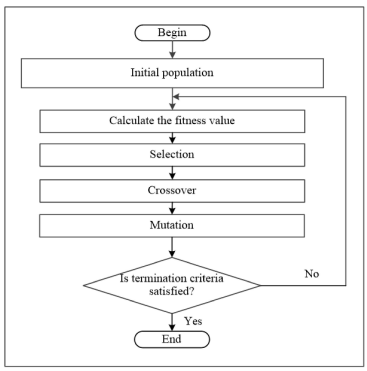

Fig. 4. the basic steps of genetic algorithm

What's worth mentioning is how we encode the model parameters and how the crossover and mutation are implemented.

- ***Encoding*** According to our formalization, the only variable in this model is the task queues of each slot. We come up with a encoding that uses a permutation of tasks and a permutation of sizes, which split the task permutation according to the sizes to generate all task queues.
  For example, if we have 5 tasks and 3 slots with the task permutation $[3, 1, 2, 4, 5]$ and the size permutation $[2, 1, 2]$, then the resulting tasks queues will be $[3, 1], [2], [4, 5]$.
- ***Crossover and mutation*** According to this article [3], intuitively, the information in the permutations is the order of elements, so the crossover have to combine orders of two permutations. It can be done by maintaining part of the first permutation while reordering the other part as it appears in the second permutation.
  For example, we have two permutations $[2, 3, 4, 1, 5], [3, 1, 4, 5, 2]$ and we want to maintain the segment between the second and the third element:

$$\begin{aligned} &[2, 3, 4, 1, 5], [3, 1, 4, 5, 2] \\ \rightarrow &[-, 3, 4, -, -], [-, 1, 4, -, -] \\ \rightarrow &[1, 3, 4, 5, 2], [2, 1, 4, 3, 5] \end{aligned}$$

  For mutation, similar to the implementation of genetic algorithm on binary strings, which is a random bit flip, we do random swaps to adjacent elements to implement it.

5.3 Greedy Method in Optimization

While our algorithm can converge to a sub-optimal solution, it may be relatively slow when the dependencies of tasks are complicated or the number of slots is significantly limited. In this situation many tasks will be in each task queue, and we observed that many tasks must wait for the dependent task assigned to the approximately the tail of the queue, and the slots are out of work. So if we advance the depended tasks to the head, the arrangement has a greater possibility to be an optimal solution. So we give bigger chance to the nodes closer to the source nodes in arrangement.

We first do the topological sort in each DAG, then in the time scheduling tasks, tasks will be sort in the same order as the topological sort. We guarantee that in the same task queue each task will appear after its dependents. In intuition this method works and we will test it later.

6 Performance and Results

6.1 Running Result of algorithm

We test our algorithm on the toy data to check the correctness and efficiency. And it turns out to be an efficient algorithm with approximately the optimal solution, as the running result shown in figure 5.

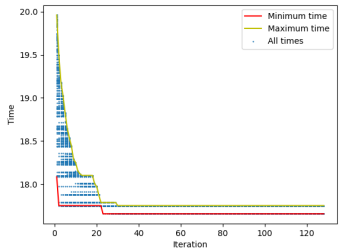

Fig. 5. the running result on the toy data

6.2 Discussion of the Result

We can see that in the running result of the toy data, the computation result(finishing time) will converge to about 17.65 in several iterations. This is due to the toy data is not complicated, and there is few tasks.

What's more, the number of tasks is less than the number of slots. If we use another group of data, then the result will be different.

We found no evident change after applying greedy method, for in the toy data, the number of slots is bigger than that of tasks, so the greedy method can not have much optimization.

To test whether our greedy method works. We designed an set of data consisting of just one single data center and two slots. But the job is relatively complex with a complicated DAG structure, as shown in 6. According to the running result, as shown in the figure 7 and 8 , we can know the method indeed improve the performance significantly in this situation.

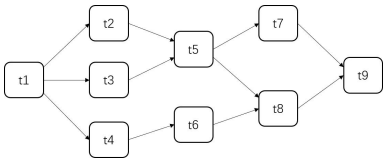

Fig. 6. the Structure of the Job in Generated Data, from the Project Description

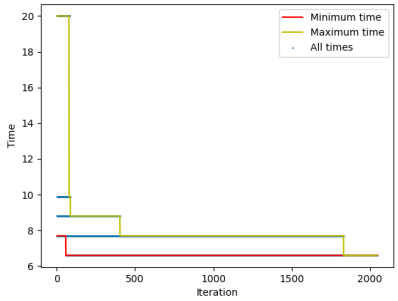

Fig. 7. the running result on the generated data without greedy

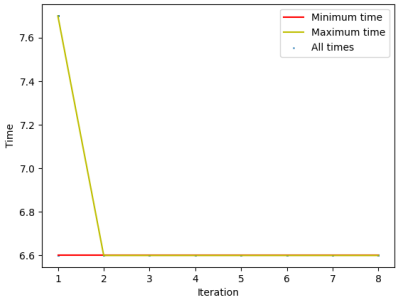

Fig. 8. the running result on the generated data with greedy

7 Conclusion and Discussion

Based on the results of the code, we can have the following conclusion. 888 From the visual diagrams of our result, we can see that the genetic algorithm actually saved great time and computational resources compared with brute force method. What's more, the greedy approach obviously enhanced the performance of the algorithm. We can see that the algorithm successfully solved the model and get a satisfying result.

The algorithm we have designed can be applied to data transferring across distant data centers. What's more, networks containing many vertices and flow capacities can also fit in this model.

The model we constructed is similar to the descriptions in the requirement document. However, the model still far from being applied to the real situation. The genetic algorithm may perform badly on large scale of data. The bandwidth restriction may be more strict and there may be power and network failure causing some links unavailable. What's more, we may not evaluate the tasks and their execution time accurately due to the unstable performance of the computing slots. The model and algorithm still have a lot of work to do to be perfected.

Acknowledgements

In the process of completing the project, we had many thoughts.

To be honest, the project is a great challenge for all of us. From the beginning, the meaning of the requirement and background information was difficult for us to understand. What's more, it is tough for us to formalize a practical problem to certain calculable mathematical models. Even at the end of the project, the model still has some flaws that can be improved.

However, we actually did plenty of work and learned a lot from the process. The project indeed require us to utilize what we have learned in the lessons to solve problems with practical backgrounds. And our programming skills and documentation skills are also polished. In the process of building the model, we have to modify it according to the execution result of the code. Witnessing the process of our model approaching the final result is a great sense of achievement.

In the end of the project, we want to give thanks to Prof.Gao and Prof.Wang for designing the course and teaching us with such useful knowledge. Thanks to two conscientious TAs for guiding us no matter the labs or the project. Thanks to our corporation and participation which give all of us wisdom and strength.

References

1. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. **31**(4) (December 1999) 406–471
2. Roetzel, W., Luo, X., Chen, D.: Chapter 6 - optimal design of heat exchanger networks. In Roetzel, W., Luo, X., Chen, D., eds.: Design and Operation of Heat Exchangers and their Networks. Academic Press (2020) 231–317
3. Whitley, D., Yoo, N.W.: Modeling simple genetic algorithms for permutation problems. Volume 3 of Foundations of Genetic Algorithms. Elsevier (1995) 163–184