N-Step Search Based Greedy Scheduler

Ke Wu, *Electrical and Computer Engineering*
Linquan Chen, *Electrical and Computer Engineering*

Abstract

In this paper, we implement four different scheduling policies: soft, step-0, hard and none, and use the none policy as baseline. We restrict ourselves to greedy policy, and implement the soft policy using N-step search algorithm which will choose to delay allocating resources to a job, even though it could have been executed when it could achieve higher utility. Based on the soft policy, we reduce the search step to zero, and implement the step-0 policy, which can highlight the advantages of the N-step search. As for hard policy, although it uses the same N-step search policy as soft policy, it only allocates preferred resources, which results in wasting lots of free resources and reducing the utility.

Keywords: scheduling, greedy policy, N-step search

1. Problem Space

In our implementation of scheduler, there are two entry functions, namely *AddJob()* and *FreeResources()*. The scheduling problem can be described as given a resource set *R* and a pending job list *L*, how to allocate resources for jobs to achieve the shortest completion time and the maximum utility. Scheduling in *AddJob()* is just add a new pending job to the job list *L* and then try to solve the above problem. Scheduling in *FreeResources()* is just add more resources to the resource set *R* and then try to solve the above problem.

No matter which scheduling strategy is used, it must satisfy the greedy policy described in handout. The expected utility for a pending job can be calculated as

$$E(U) = U_{max} - E\left(T_{completion}\right) - (T_{current} - T_{arrive})$$

For a given resource set *R* and a pending job list *L*, the runnable jobs (jobs in *L* whose requirement can be satisfied by *R*) are known. If the scheduler decides to allocate some resources *r* for one runnable job *J*, that job must be the one that has the highest expected utility. After the allocation, the resource set becomes $R' = R - \{r\}$ and the pending job list becomes $L' = L - \{J\}$. If the scheduler decides to continue scheduling, this becomes the similar problem as the previous one. Therefore, for a given *R* and *L*, the scheduler must follow a fixed order $<J_1, J_2, \dots, J_n>$ to schedule jobs and J_n is the last job that scheduler is able to schedule with current resource set *R*.

However, a non-aggressive scheduler may not want to exhaust resources to schedule as many jobs as possible in one scheduling round. It may choose to schedule only a small number of jobs. But the order that job is scheduled must still be followed.

Finally, our problem can be described as

In one round of scheduling *S*, given a resource set *R* and a pending job list *L*, choose a prefix $<J_1, \dots, J_k>$ of $<J_1, \dots, J_n>$ to schedule jobs so that

$$max \sum_{S} \sum_{1 \le i \le k} E(U_{J_i})$$

where J_n is the last job that scheduler might be able to schedule with R. The choice of k gives the scheduler some freedoms to achieve a better performance.

2. Degrees of Freedom

In phase 1, our scheduler must follow the FIFO policy to schedule jobs. In phase 3, our scheduler must follow the greedy policy but it has the freedom to choose the value of k to determine how many jobs to schedule in each round. If it determines to schedule a job, it will try to schedule it with its preferred configuration. In other words, our scheduler will not schedule a job with non-preferred resources if the preferred resources are available, even though it may not achieve the highest utility.

3. Data Structure and Algorithms

3.1. Data Structure

Each virtual machine is represented as an object of *MyMachine* class. The states of all machines are stored in a vector. Each element in the vector represents a rack, and the size of the vector is the number of racks. Every element of the vector is also a vector in which every element represents a virtual machine, and the size of it is the number of machines in that rack.

Each job is represented as an object of *MyJob* class. Pending jobs and running jobs are stored separately. All pending jobs are stored in a list. All running jobs are stored in a priority queue. The running jobs in priority queue are sorted based on their completion time.

Besides, a class called Cluster is used for scheduling algorithm implementation-convenience. An object of

Cluster is just a deep copy of the current state (i.e. a snapshot) of the scheduler (vector of virtual machines, list of pending jobs and priority queue of running jobs).

3.2. Scheduling Algorithms

Assume no future job will come (i.e. *AddJob()* will not be invoked), then the next time of scheduling is the time when a running job is finished and releases resources. Because the expected completion time for each running job is known, the scheduler actually knows when the next time of scheduling will happen.

As discussed in section 1, the scheduler should choose a k in each round of scheduling to obtain total utility as much as possible. Assume the scheduler chooses a value of k and allocates resources for J_1-J_k and removes these jobs from L and adds them to running job priority queue Q. After this round of scheduling, the state of the scheduler becomes R', L' and Q'. If the no-new-pending-job-comes assumption holds, the scheduler knows that the next time of scheduling will happen in exactly the time when the running job at the head of Q' is completed and free resources. At that time, the state of the scheduler can be described as R'', L'' and Q'' which can be calculated by scheduler. So after the scheduler chooses a value for k, it can calculate and predict the state of the scheduler in the next time of scheduling. This is the basis of our scheduling algorithm. In the next time of scheduling, the scheduler should choose a new value for k which is exactly the same problem the scheduler tries to solve in the first place.

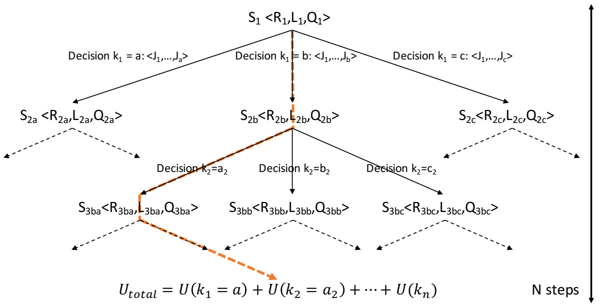
S_1 <R_1,L_1,Q_1>
Decision k_1 = a: <J_1,…,J_a>
Decision k_1 = b: <J_1,…,J_b>
Decision k_1 = c: <J_1,…,J_c>
S_{2a} <R_{2a},L_{2a},Q_{2a}>
S_{2b}<R_{2b},L_{2b},Q_{2b}>
S_{2c}<R_{2c},L_{2c},Q_{2c}>
Decision k_2=a_2
Decision k_2=b_2
Decision k_2=c_2
S_{3ba}<R_{3ba},L_{3ba},Q_{3ba}>
S_{3bb}<R_{3bb},L_{3bb},Q_{3bb}>
S_{3bc}<R_{3bc},L_{3bc},Q_{3bc}>
$U_{total} = U(k_1 = a) + U(k_2 = a_2) + \cdots + U(k_n)$
N steps

Figure 1. Search tree when choosing k_1

To achieve the maximum utility, the scheduler should choose k values <k_1, k_2, ... k_m> for each time of scheduling <S_1, S_2, ... S_m> so that the total obtained utility is maximized. As shown in the figure 1, when the scheduler choosing a value for k_1, the scheduler could actually simulate what will happen in the future and calculate the final utility it will get. Ideally, the scheduler could simulate all possible combination of <k_1, k_2, ..., k_m> and calculate their total utility. Based on the calculation, the scheduler knows how to choose k_1 to maximize the final utility. This can be implemented as a DFS algorithm. With n pending jobs and m running jobs, the time complexity of the search algorithm is O(n^m) for each round of scheduling. To make sure the decisions can be done in a reasonable time, we limit the maximum steps of searching. When the searching algorithm reaches the maximum step, the scheduler just schedules as many jobs as possible with current resources and calculate total utility obtained so far.

Compared with a "dumb" scheduler that schedules as many jobs as possible as long as some resources are available. Our scheduler tries to predict what will happen within n steps of future then chooses a best decision.

4. Experimental Analysis

We used four different combined trace files to test four different scheduling policies, which are soft, step-0, hard and none. The none policy is FIFO random scheduler. The soft and hard policies are both based on N-step search. The step-0 policy is similar to soft policy, except that instead of performing further search, it just acts as a "dumb" scheduler that schedules as many jobs as possible as long as some resources are available. The experimental results are shown at Table 1.

In our experiment, we use the none policy as our baseline. The utility is pretty low when using none policy, and other three policies get higher utility at each experiment. The soft policy is our default and best policy. There are several reasons why the soft policy succeeds.

As we can see from the results, the soft policy can achieve higher utility than the step-0 policy, because in soft policy when we get better allocated strategy after searching, we will choose to delay allocating resources to a runnable job, even though it could have been executed. For example, although there are enough resources for two pending jobs, we will still search the situations that starving one or two jobs to see if it is possible to achieve higher utility in the future. Therefore, we may choose to wait for the specific running job finishes, and then allocate two jobs to the preferred resources. Although we waste some utility to wait, the total utility we get from this allocation strategy will be higher.

Besides, the soft policy also can achieve higher utility than the hard policy. Although hard policy uses the same N-step search algorithm as soft policy, it only chooses to allocate preferred resources. Therefore, hard policy will delay allocating resources even there are enough free resources and will get high utility to allocate the job, which will waste a lot of resources and reduce the total utility. Especially when we have limited GPU resources and get many GPU jobs, the hard policy will get lower utility. For example, we get 47 jobs at Exp3 and 27 of them are GPU jobs, therefore lots of GPU jobs are blocked and be killed at last.

Table 1. Average utility for different policies

	Exp1[a]	Exp2[b]	Exp3[c]	Exp4[d]
soft	800.50	741.13	935.17	757.89
step-0[e]		636.15	919.59	
hard	741.19	724.73	700.84	683.29
none		358.45		

[a] traceMPI-c2x6-rho0.70,traceGPU-c2x4-rho0.80

[b] traceMPI-c2x4-rho0.80,traceGPU-c2x4-rho0.80

[c] traceMPI-c2x4-rho00.80,traceGPU-c2x4-rho00.80

[d] traceMPI-c2x4-rho0.advcc10.90,traceGPU-c2x4-rho0. advcc10.90

[e] soft policy without further search

5. Weaknesses and Improvements

5.1. Non-preferred Resources

In our scheduler, we do only allocate preferred resources to a job when there are preferred and no-preferred free resources. For example, currently, there are four free GPU machines in the GPU rack and two free no-GPU machines in each other rack, which can be displayed as [4, 2, 2, 2]. After finishing the search process, the highest utility job is a MPI job which requires four machines, therefore, we decided to allocate the job to the GPU rack. However, the next highest utility job is a GPU job which also requires for four machines. Thus, we can only allocate the GPU job to no-GPU racks. As we know, it needs more than 100-200 seconds to run a GPU job on non-preferred machines than preferred machines. However, it only takes more than 30-40 seconds to run a MPI job on different racks than the same rack. Therefore, if we choose to allocate the MPI job to two different racks (still the highest utility job) firstly and then allocate the GPU job to the GPU rack, this will improve the total utility of these two jobs.

The reason that the scheduler has this weakness is we don't consider the next potential job when calculate the utility. Consider on this problem, we can improve the scheduler as following:

When there are enough machines and the job has the highest utility whenever running on preferred resources and non-preferred resources, we need to calculate the current total utility in these two situations plus the next potential pending job.

5.2. Job Starves

Another weakness in our scheduler is that there is a high possibility to make job starve and we have no idea to guarantee that no job starves. There are several reasons as following:

Firstly, because we are restricted to greedy policy that must allocate the highest utility job at first. Therefore, the lowest utility job need to wait until all other pending jobs begin to run, which causes the lowest utility job gets lower utility and be killed at last.

Secondly, our policy even chooses to make some jobs starve (dealy) in some specific situations. For example, when there are non-preferred resources for a job, we will try to analysis the running jobs. If it is better to wait one job to finish and run the pending job, we will starve the potential job first.

5.3. Optimization of search algorithm

As mentioned in section 3, our scheduler must limit the maximum searching steps due to the time complexity. Currently we set the maximum searching steps to 7. However, some optimization can be used to improve the simple DFS algorithm. For example, branch-and-bound algorithm is a possible improvement. If the scheduler finds the upper bound of the utility in a search branch can not exceed the highest utility found so far, the scheduler could just discard the branch without searching.

5.4. Estimated running time

Because our algorithm need to search and simulate what will happen in the future, it heavily depends on the estimated running time of each job. If the difference between the real running time and the estimated time is large, our scheduler might perform badly. Actually, we observed about 10% difference between the two time during the experiments. A possible improvement is to add a compensation factor for the estimated time so that it is more approximate to the real time. However, we don't add it to our code because we should not make assumption about the trace.

6. Deigned Trace

Our designed trace contains 10 MPI jobs and 7 GPU jobs, which will run nearly 900 seconds. For MPI jobs, all jobs require for same machine resource, and for GPU jobs, most of them require for two machines and only one or two jobs requires for three or four machines. We design this trace mostly in order to test the following conditions:

When there are enough resources (likely non-preferred resources) for two pending jobs, a well designed scheduler may choose to not to schedule the two jobs to the non-preferred resources immediately. Because there is a running job will finish and free several machines which will make preferred resources for the two pending jobs.

In our scheduler, although there are enough resources for two pending jobs, we will still search the situations that starving one or two jobs. Therefore, we will choose to wait for the specific running job finishes, and then allocate two jobs to the preferred resources. Although we waste some utility to wait, the total utility we get from this allocation strategy will be higher.