# Apache Hadoop YARN Placement-Aware Scheduling

Di Jin (dijin@andrew.cmu.edu) HingOn Miu (hmiu@andrew.cmu.edu)
*Carnegie Mellon University*

## Introduction

Apache YARN is one of many open-source implementations of MapReduce, with improvement in scalability, efficiency and the capacity of supporting heterogeneity compared with Hadoop 1.x. An important issue for such improvement is the 2-level scheduler design with different scheduling strategies. In default, YARN supports three types of schedulers: FIFS, fair sharing, Dominant Resource Fairness. Although these types of schedulers can deal with most jobs encountered, they are not al-ways efficient, nor are they suitable for scheduling some specified cases. In this project, we explore fields such as scheduling utility, queuing latency through extensive experiments.

## 1. Degree of Freedom

We have 5 scheduling policies: the NONE scheduler allocates machines randomly to jobs in FIFO order, the HETERGEN scheduler also allocates machines to jobs in FIFO order but it tries allocating preferred resources before allocating non-preferred resources, the SJF scheduler handles jobs in their duration order and also tries to allocate preferred resources first, the HARD scheduler picks the highest utility job and only allocates its preferred resources, and the SOFT scheduler picks the highest utility job and tries allocating preferred resources before allocating non-preferred resources.

All above policies have little degree of freedom in deciding the job execution order because there is only one job can be executed in any time, either the earliest job, shortest job, or the highest utility job. The HARD and SOFT schedulers adapt the greedy approach to maximize total utility such that at the time a job starts running, there is no job that has arrived, has not started, and has higher expected utility, given the available re-sources. The NONE and HETERGEN schedulers al-ways find the earliest arrived waiting job such that at the time a job starts running, there is no job that has arrived, has not started, and arrives earlier, given the available resources. The SJF scheduler always find the shortest duration job such that at the time a job starts running, there is no job that has arrived, has not started, and is “shorter”, given the available resources. In order words, all above schedulers are allowed to delay scheduling decision, but there is only precisely one job can be handled at any time.

## 2. Description of Code

### 2.1. Header

Data Structures:

***int free_machines***

Global counter that keeps track the number of available machines.

***int mode***

Global variable that keeps track the mode (NONE, HARD, SOFT), which sets default by make and overrides by config file.

***vector <vector<int32_t>> free_racks***

Global 2D vector that keeps track of the free machines of each rack.

***vector <vector<int32_t>> used_racks***

Global 2D vector that keeps track of the used machines of each rack.

***unordered_map<int32_t, time_t> free_times***

Global hash table that maps used machine to its expected free time.

***set<Job_S *> jobs***

Global set of all waiting jobs.

***mutex mtx***

Global mutex to ensure synchronization on resource allocation.

```
struct Job_S {

    JobID jobId;
    job_t::type jobType;
    int32_t k;
    int32_t priority;
    double duration;
    double slowDuration;
    double chosen_duration;
    time_t added_time;
    int32_t *machines;
};
```

Dynamically allocated structure of job info.

### 2.2. Scheduling Algorithm

**Initialization**:

Our code reads the scheduling policy and rack information from config file, and initializes mode as the scheduling policy, free_machines as the total number of machines, free_racks as the indices of available machines and used_racks as empty 2D vector.

**AddJob**:

Our code first acquires mutex locks. It records the added_time of the new job and adds the new job to jobs set. It then calls TryToAllocate to see if resource allocation is possible and releases the mutex locks.

**FreeResources**:

Our code first acquires the mutex lock. Then, it places the freed machines from used_racks to free_racks. Also, free_machines count is updated. Next, it calls TryToAllocate to see if resource allocation is possible. Lastly, it releases the mutex lock.

**TryToAllocate**:

First, our code tries to remove all time-out jobs from jobs set. The timeout jobs are those jobs that have condition of (maxT - duration) < (waited time in jobs set). So, this can avoid wasting resources to run jobs that will be killed anyways.

Then, depending on the scheduling policy, our scheduler picks the earliest added job if the mode is NONE or HETERGEN, picks the shortest duration job if mode is SJF, or picks the highest utility job if the mode is HARD or SOFT. For NONE scheduler, machines are randomly picked from free_racks.

For HETERGEN scheduler, our code first checks if preferred resources allocation is possible for the earliest job. If preferred resources allocation is possible (GPU job wants machines from first rack and MPI wants machines from same rack), those resources are allocated. If preferred resources allocation is not possible, we check the condition (the minimum expected waiting time + duration) < (slowDuration). If true, our code chooses to wait for next free rack. If not, our code picks machines randomly across different racks.

For SJF scheduler, our code first checks if preferred resources allocation is possible for the shortest duration job, and repeats above process if preferred resources allocation is not possible.

For HARD and SOFT, to pick highest utility job, our code first traverses all jobs to compute utility of each job and free_machines count is used to skip those jobs that can-not be fulfilled at the moment. Our code first checks if preferred resources allocation is possible for each job. If preferred resources allocation is possible, the utility is computed with maxT - duration - (waited time in jobs set). If preferred resources allocation is not possible, HARD scheduler always chooses to wait for free rack while SOFT scheduler computes the minimum expected waiting time for the next available rack. If (the mini-mum expected waiting time + duration) < (slowDura-tion), SOFT scheduler chooses to wait for next free rack. If not, SOFT scheduler computes utility as maxT - slowDuration - (waited time in jobs set) and picks machines randomly across different racks.

Once the earliest job, the shortest job or the highest utility job is picked (depending on the mode), our code allocates the free machines and moves them from free_racks to used_racks. Then, the expected freed time of those machines are stored in free_times such that they can be used to compute the minimum expected waiting time for next free rack. Also, free_machines count is updated. The handled job is then removed from jobs set. Finally, our code checks if the jobs set is empty, and if not, it picks the new earliest job or highest utility job to handle.

## 3. Non-preferred Resources Allocation & Resources Allocation Delay

Obviously, NONE scheduler does not have a sense of preferred resources allocation. The HARD scheduler always allocates preferred resources. So, when only non-preferred resources are available, the HARD scheduler chooses to wait for preferred resources to be freed. On the other hand, when preferred resources for a job are not available, the HETERGEN, SJF, and SOFT schedulers evaluate the expected waiting time for its preferred resources to decide whether they should wait for preferred resources or not.

These 3 schedulers compare (the minimum expected waiting time for preferred resources + fast duration time) with (slow duration time). If former is less than latter, it means that this job should wait for its preferred resources. If not, non-preferred resources allocation is decided.

| Scheduling Policy (Mean Completion Time) | NONE | HETERGEN | SJF |
|---|---|---|---|
| traceGPU-micr | 122.319 | 97.874 | 97.70 |
| traceMPI-min | 188.855 | 147.258 | 146.33 |
| traceMPI-c2x1-rho0.xx-batc | 659.6147 | 431.45166 | 417.34616 |

**Table 1. Three Scheduling Policies**

We see the performance of NONE scheduler from above chart that the sense of preferred resources allocation is very important in improving running time as the jobs do not have to run in their slowDuration most of the time. We see a rather similar performance improvement for HETERGEN scheduler and SJF scheduler, since they both adopt the same strategy in delaying resources allocation. The difference between their performances should explain whether job arrival order or job duration order is more essential in improving completion time. With above results, we see job duration order is more essential.

| HARD | traceCombined-c2x6-rho0.70 | traceCombined-c2x4-rho0.80 |
|---|---|---|
| Mean Completion Time | 378.688 | 529.429 |
| Total Utility | 28745.916 | 38893.116 |

**Table 2. Hard Scheduler Performance**

| SOFT | traceCombined-c2x6-rho0.7 | traceCombined-c2x4-rho0.8 |
|---|---|---|
| Mean Completion Tim | 379.80 | 396.01 |
| Total Utilit | 28706.95 | 46631.29 |

**Table 3. Soft Scheduler Performance**

For HARD and SOFT schedulers, we see a huge improvement in traceCombined-c2x4-rho0.80, but essentially no difference in traceCombined-c2x6-rho0.70. This behaviour is expected because preferred resources are available most of the time for MPI jobs in trace-Combined-c2x6-rho0.70, and so there is little influence of deciding to allocate non-preferred resources. For traceCombined-c2x4-rho0.80, the MPI jobs are added more closely together and so their preferred resources are not usually available.

Therefore, it is important for scheduler to be capable to decide whether to allocate non-preferred resources smartly. If the scheduler decides non-preferred resources allocation for a MPI job, it means that its completion time can be shortened and so it maximizes the overall utility.

## 4. Job Starvation

For the HARD and the SOFT scheduler, the low utility jobs are starved as more new jobs are added. This starvation behaviour is expected since utility of each job is inversely correlated to its waiting time in jobs set, and so those new jobs are more likely to have higher utility. Those low utility jobs eventually are more likely to be killed by timeout errors. For the SJF scheduler, the long duration jobs are starved as more new short duration jobs are added. This starvation behaviour also is expected since job duration of each job is inversely correlated to its waiting time in jobs set, and so those long duration jobs eventually are more likely to be killed by timeout errors.

The NONE and HETERGEN schedulers do not suffer from job starvation because each job is ran by its arrival order and so no job is unfairly starved. However, we see that the NONE scheduler performs much worse than the utility based schedulers. So, we believe that utility based job starvation is good for overall completion time and utility since it allows higher utility jobs to be completed more efficiently with preferred resources.

| NONE | traceCombined-c2x6-rho0.7 | traceCombined-c2x4-rho0.8 |
|---|---|---|
| Mean Completion Tim | 519.20 | 887.16 |
| Total Utilit | 23827.83 | 18144.56 |

**Table 4. None Scheduler Performance**

## 5. Weaknesses

Our utility based scheduler is greedy so that it always picks the highest utility job to maximize the overall utility. However, the greedy approach does not always give the most optimal solution. If the trace is known to scheduler prior to running the jobs, the scheduler can easily compute the optimal configuration of running the jobs. Since this is unrealistic, greedy approach is a good strategy to get near-optimal solution.

We realized that job starvation is a big issue. An easy way to fix this is to prioritize jobs with utility getting close to 0 so that they can finish execution before they get timeout errors. However, this sacrifices the overall completion time and utility.

## 6. Designed Trace

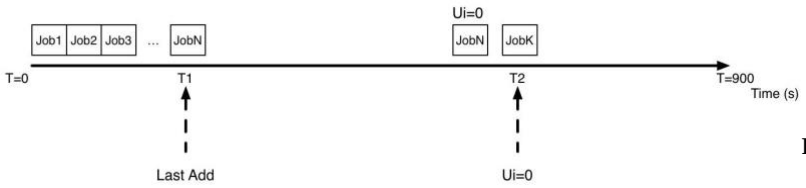


**Fig. 1 Designed Trace Workflow**

The idea of the designed trace is illustrated above. Within the time period of [0, T1], job1, job2, ...jobN are added. All of these jobs have long duration. Within the period of [T1, T2], no more jobs are added. We design our trace in the way that shortly before time T2, some of those N jobs in jobs set will fail to have sufficient time to complete execution such that its utility reaches 0.

In other words, even if those jobs run, they will still get timeout errors. For schedulers that do not check this condition, those job will run. Then, at time T2, a new job jobK is added. So, those old jobs will block the re-sources that can be used to execute the new job jobK. However, as our scheduler checks and removes all those jobs from jobs set if waited_time + execution_time > Max_T, those old jobs will be removed so that the new job jobK will not be blocked, thus improving the overall completion time and utility. The experiment results are shown as follows. The highlighted job result in both cases is for the newly added job, and we do see a huge performance difference when those already-timeout waiting jobs are discarded smartly.

```
0-4-0-315.0-345 : F     T:  333.79      U:  866.21
0-4-0-315.0-345 : F     T:  364.06      U:  835.94
0-4-0-315.0-345 : F     T:  414.66      U:  785.34
0-4-0-315.0-345 : F     T:  426.66      U:  773.34
0-4-0-315.0-345 : F     T:  637.46      U:  562.54
0-4-0-315.0-345 : F     T:  682.75      U:  517.25
0-4-0-315.0-345 : F     T:  725.34      U:  474.66
0-4-0-315.0-345 : F     T:  959.07      U:  240.93
0-4-0-315.0-345 : F     T: 1023.90      U:  176.10
0-4-0-315.0-345 : F     T: 1033.82      U:  166.18
0-4-0-67.0-78   : F     T:   83.41      U: 1116.59
0-4-0-315.0-345 : K     T: 1192.44      U:    7.56
0-4-0-315.0-345 : K     T: 1192.61      U:    7.39
0-4-0-315.0-345 : K     T: 1192.18      U:    7.82
0-4-0-315.0-345 : K     T: 1192.39      U:    7.61
0-4-0-315.0-345 : K     T: 1193.80      U:    6.20
0-4-0-315.0-345 : K     T: 1194.38      U:    5.62
mean completion time (E[T]) = 814.277470588
total utility = 6557.283
```

**Fig. 2 Scheduler removed already-timeout waiting jobs**

```
0-4-0-315.0-345 : F     T:  347.80      U:  852.20
0-4-0-315.0-345 : F     T:  374.35      U:  825.65
0-4-0-315.0-345 : F     T:  388.06      U:  811.94
0-4-0-315.0-345 : F     T:  419.83      U:  780.16
0-4-0-315.0-345 : F     T:  662.25      U:  537.75
0-4-0-315.0-345 : F     T:  677.84      U:  522.16
0-4-0-315.0-345 : F     T:  690.60      U:  509.40
0-4-0-315.0-345 : F     T:  966.64      U:  233.36
0-4-0-315.0-345 : F     T: 1003.55      U:  196.45
0-4-0-315.0-345 : F     T: 1012.90      U:  187.10
0-4-0-315.0-345 : K     T: 1194.56      U:    5.44
0-4-0-315.0-345 : K     T: 1193.28      U:    6.72
0-4-0-315.0-345 : K     T: 1192.17      U:    7.83
0-4-0-315.0-345 : K     T: 1193.10      U:    6.90
0-4-0-315.0-345 : K     T: 1193.36      U:    6.64
0-4-0-315.0-345 : K     T: 1193.08      U:    6.92
0-4-0-67.0-78   : F     T:  335.49      U:  864.51
mean completion time (E[T]) = 825.816882353
total utility = 6361.113
```

**Fig. 3 New job is blocked by soon-timeout waiting jobs**

The designed trace is included in the report folder, and it is under the "evidence/phase3/designed _trace/" directory. The file "results_designedTrace_ good.analysis" corresponds to Fig. 2; file "results_ designedTrace_bad_analysis" corresponds to what is illustrated in Fig. 3.

## 7. Reference


1. Vavilapalli, Vinod Kumar, et al. "Apache Hadoop yarn: Yet another resource negotiator." Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013.