HRF: A Resource Allocation Scheme for Moldable Jobs

Song Wu, Qiong Tuo, Hai Jin, Chuxiong Yan, Qizheng Weng
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
wusong@mail.hust.edu.cn

ABSTRACT

Moldable jobs, which allow the number of allocated processors to be adjusted before running in clusters, have attracted increasing concern in parallel job scheduling research. Compared with traditional rigid jobs where the number of allocated processors is fixed, moldable jobs are more flexible and therefore have more potential for improving their average turnaround time (a crucial metric to describe performance of jobs in a cluster). Average turnaround time of moldable jobs depends greatly on resource allocation schemes. Unfortunately, existing schemes do not perform well in reducing average turnaround time, either because they only consider a single job's turnaround time instead of the average turnaround time of all jobs, or because they just aim at fairness between short and long jobs instead of their average turnaround time. In this paper, we investigate how resource allocation affects the average turnaround time of moldable jobs in clusters, and propose a scheme named HRF (*highest revenue first*), which allocates processors according to the highest revenue of shortening runtime. In our simulations, experimental results show that HRF can reduce average turnaround time up to 71% when compared with state-of-the-art schemes.

Categories and Subject Descriptors

C.m [**Computer Systems Organization**]: Miscellaneous; D.2.8 [**Metrics**]: performance measures

General Terms

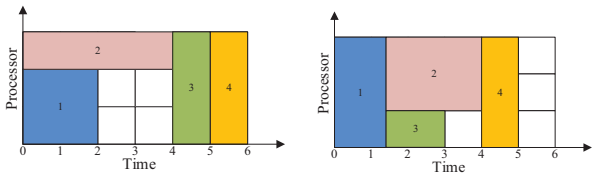
Algorithms, Design, Performance

Keywords

Moldable job, Scheduling, Resource allocation

1. INTRODUCTION

Job scheduler is one of the most important components in clusters, which determines the job performance (e.g., turnaround time). In rigid job scheduling, users specify a fixed number of processors for a job. However, most parallel jobs can run on an unfixed number of processors [1]. Moldable jobs allow a scheduler to adjust such numbers before scheduling. Because of the flexibility of moldable jobs, the scheduler can decide the number of processors allocated to jobs to reduce their waiting time or runtime, thus improving performance by reducing the average turnaround time of the jobs. Figure 1 shows an example: the runtime of Job 1 and 2 is reduced and the waiting time of Job 3 and 4 is reduced (and thus average turnaround time of jobs is reduced) when adopting a flexible resource allocation in moldable job scheduling.



(a) rigid job scheduling (b) moldable job scheduling

Figure 1: Moldable job scheduling can improve performance of average turnaround time when compared with rigid job scheduling.

Rigid job scheduling is traditional scheduling whereby is only necessary to consider the scheduling order of jobs. The simplest scheduling policy of rigid jobs is based on *first come first served* (FCFS), whose principle is to queue jobs in order of submission, and the queue head job will be served when spare processors can satisfy its requirement. However, FCFS has proved its poor performance due to generating resource fragmentation [2, 3]. To overcome this problem, several backfilling policies [4, 5, 6] have been proposed and implemented in production schedulers. Backfilling allows later arriving jobs in the queue to move forward to use the fragmentation as long as they do not delay preceding jobs. Since backfilling fills fragmentation in clusters with short jobs, the resource utilization is greatly improved, and the average turnaround time is shortened.

Different from rigid job scheduling, moldable job scheduling policies not only decide the scheduling order of queued jobs (job selection scheme), but also decide how many processors are allocated to those jobs (resource allocation scheme). There have already been some researches on moldable job scheduling summarized in [1], but their resource allocation schemes do not perform well in reducing the average turnaround time of jobs, because they either only consider a sin-

CF'15 May 18-21 2015, Ischia, Italy

http://dx.doi.org/10.1145/2742854.2742870.

gle job's turnaround time instead of the average turnaround time of all jobs, or they just aim at fairness between short and long jobs instead of their average turnaround time. Therefore, our motivation is to design a resource allocation scheme for moldable jobs, aiming to reduce their average turnaround time.

In this paper, we study average turnaround time in an example with different processor allocation strategies, which has inspired us to reduce the runtime of moldable jobs with limited processors. We propose an indicator named *Revenue per Processor* (RP), which expresses the revenue of shortening runtime with every allocated processor. Based on this indicator, we design a resource allocation scheme named *highest revenue first* (HRF), which always allocates processors to the job with the highest RP, in order to reduce the average turnaround time of jobs in clusters.

We conduct simulation experiments by fulfilling scheduling policies integrating HRF and other state-of-the-art resource allocation schemes, and observe the average turnaround time of jobs. The results show that the policy integrating HRF can achieve an improvement of up to 71% in average turnaround time.

This paper is organized as follows. In section 2, we introduce the background and related work. Section 3 introduces two kinds of schemes in moldable job scheduling and our motivation. We propose the indicator RP in section 4 and propose our resource allocation scheme HRF based on RP in section 5. In section 6 we conduct experiments with scheduling policies integrating HRF. We talk about future work in section 7 and conclude our paper in section 8.

2. BACKGROUND AND RELATED WORK

In this section, we discuss current related research on parallel job scheduling, and compare different kinds of parallel jobs. Aiming at moldable job scheduling, we give a brief description of the Downey model used to compute job runtime with different numbers of processors.

2.1 Parallel Job Scheduling

In traditional clusters, rigid job scheduling specifies a fixed number of processors before job submission, which causes a resource fragmentation problem [2, 3]. Though backfilling [2, 4, 7, 8] is adopted to alleviate this problem, rigid job scheduling still performs poorly in a busy shared-cluster facility [9].

Compared with rigid jobs, moldable jobs are flexible in selecting the number of processors and thus have potential to improve performance. Cirne et al. [9] propose a greedy scheme to allocate processors to each job. Srinivasan et al. [10] find the performance of small jobs under a greedy scheme decreases to a large extent, and consequently proposes a fair share scheme. Krishnamoorthy et al. [1] find previous approaches are not very robust under different load and scalability of jobs, and propose an enhanced approach to make scheduling more robust. Gerald et al. [11] adopt an iterative method in processors allocation, which is similar to our work. But they do not consider how the parameter (they call overbooking) affects the scheduling performance.

Malleable jobs are even more flexible than moldable jobs because of parallelism variance when jobs are running. Utrera et al. [12] propose a VM-based FCFS-malleable algorithm, which aims to improve system utilization and shorten waiting time. He et al. [13] introduce a two-level framework for malleable job scheduling. Algorithms A-GREEDY and A-STEAL are used to schedule jobs. However, Sun et al. [14] finds that A-GREEDY algorithm suffers from instability. These authors propose A-CONTROL algorithm based on control-theoretic properties to solve this problem.

With the development of parallelism, moldable job scheduling becomes more and more practical and is fulfilled in an HPC management system owned by IBM [15]. Malleable jobs are more flexible than moldable jobs, but they require support, libraries and runtime systems [12], which are very costly in practice. Therefore in this paper, we study the scheduling of moldable jobs among the above-mentioned parallel jobs.

2.2 Downey Model

Different number of processors can be allocated to moldable jobs, which are correlated to different job runtime. In moldable job scheduling, job runtime information can help decide how many processors are allocated to jobs. Downey studies the runtime speedup of moldable jobs with different numbers of allocated processors and builds a model for them [10, 16]. This model has been widely accepted and applied in many works on moldable jobs [1, 10].

Downey explicitly defines speedup function $S(n) = \frac{L}{T(n)}$, where $S(n)$ is speedup of a moldable job with n processors, L is sequential runtime (the job running on one processor), and $T(n)$ is runtime of the job running on n processors. The Downey model characterizes a job with two parameters: 1) σ indicates how close the speedup function is to linear (the lower σ is, the more linear the speedup curve); 2) A is used to measure the maximum speedup a job can achieve. If we know A, σ and L of a job, we can use $T(n) = \frac{L}{S(n)}$ to obtain the runtime of the job running on n processors. Due of space, the equation for $S(n)$ dependent on A and σ can be found in [16].

3. MOTIVATION

In a cluster, jobs are submitted by users and queued in order until there are enough resources for them to run. Moldable job scheduling comprises two kinds of schemes when scheduling jobs. The first is the resource allocation scheme, as shown in Figure 2, which decides the number of processors allocated to every queued job. Another is the job selection scheme, which decides which job is scheduled when resources are available. In this section, we will introduce these two kinds of schemes in detail, and indicate which scheme we are concerned with in this paper.

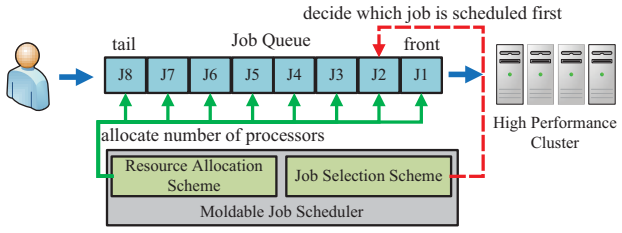


Figure 2: Moldable job scheduling policy can be divided into two kinds of schemes: resource allocation scheme and job selection scheme.

3.1 Resource Allocation Scheme

Moldable jobs are more flexible than rigid jobs because their number of processors is moldable before they are scheduled, which may potentially improve performance. While users typically have no authority to intervene in their jobs after the jobs are submitted, it is the scheduler's responsibility to allocate processors to jobs. Therefore, moldable job scheduling needs a resource allocation scheme which can efficiently allocate processors to queued jobs.

There are two occasions on which a scheme makes a resource allocation decision, submit-time and schedule-time. If the decision is made at submit-time, the number of allocated processors is unchangeable after jobs are queued (e.g., *Submit-time Greedy* and *Submit-time Fair* in [1]). Compared with submit-time, delaying the decision of resource allocation to schedule-time is more flexible (e.g., *Schedule-time Fair* in [1]), allowing a scheduler to make decisions more efficiently based on the current status of the job queue.

When deciding how many processors are allocated to moldable jobs, *Submit-time Greedy* adopts a straightforward approach, choosing the number of processors resulting in the shortest turnaround time (it is almost equivalent to the shortest runtime because the decision is made at submit-time). Typically, job runtime decreases with more allocated processors, hence this approach may allocate as many processors as these jobs can achieve. *Submit-time Fair* and *Schedule-time Fair* modify *Submit-time Greedy* by adding a restriction to the number of processors of jobs. That is, the maximum number of processors that can be allocated to each job is restricted by its weight, which refers to the sequential runtime of the job. These two fair share schemes are concerned with how to fairly allocate processors according to jobs' runtime, which may give fair opportunities to both short and long jobs. Therefore, they are able to improve scheduling performance because they can avoid resources being overly occupied by long jobs.

3.2 Job Selection Scheme

After deciding the number of processors, queued jobs wait to be scheduled to clusters. Whenever resources in a cluster can satisfy any job's requirement, the scheduler selects a job from the queue and schedules it to the cluster. Depending on which job is scheduled, the moldable job scheduler has different job selection schemes.

The most straightforward scheme is FCFS, which queues jobs in order of submission and only schedules jobs at the head of the queue. FCFS is a fair scheme respecting the submission order of jobs, but it is likely to cause resource fragmentation and thus delay jobs requiring many processors. To avoid such fragmentation, a scheme named SJF (*shortest job first*) [17] always schedules the shortest job (i.e., the job with the shortest estimated runtime) first in a queue. SJF can reduce fragmentation in clusters, but may also cause unfair problems (i.e., starve long jobs in the waiting queue). Backfilling is another scheme designed to reduce fragmentation. There are many studies on this [2, 4, 5, 6, 7, 18]. The two best known backfilling strategies are conservative backfilling [4] and aggressive backfilling [6]. In conservative backfilling, jobs are allowed to jump ahead if they do not delay other jobs. In aggressive backfilling, jobs are allowed to jump ahead if they do not delay the job at the head of a queue. Backfilling can significantly improve performance by reducing resource fragmentation while guaranteeing fair opportunities for jobs being scheduled.

3.3 Our Objective

Turnaround time is the time taken to accomplish a job, which is directly perceived by users, and average turnaround time is one of the most crucial metrics to describe the performance of all jobs in a cluster. Reducing average turnaround time has already been studied extensively in job selection schemes of rigid job scheduling policies. Because the number of allocated processors to rigid jobs is fixed, rigid job scheduling does not consider resource allocation schemes. However, as moldable job scheduling has been proposed in recent years, studies on resource allocation schemes for moldable jobs are relatively few. Some state-of-the-art schemes are introduced above (*Submit-time Greedy*, *Submit-time Fair*, and *Schedule-time Fair*), but they do not perform well in reducing the average turnaround time of jobs. For example, *Submit-time Greedy* only considers single job performance (allocates as many processors as possible to each current considering job), ignoring the effect on later jobs (prolongs their waiting time), hence resulting in poor performance. *Submit-time Fair* and *Schedule-time Fair* attempt to fairly allocate processors in terms of the weight (sequential runtime) of jobs. These two fair schemes mainly aim at guaranteeing the fairness of resource allocation to short jobs and long jobs, instead of reducing turnaround time of all the jobs.

In this paper, our objective is to design a resource allocation scheme for moldable jobs, which attempts to reduce the average turnaround time of all jobs (the performance we are concerned with in this paper).

4. BASIC IDEA

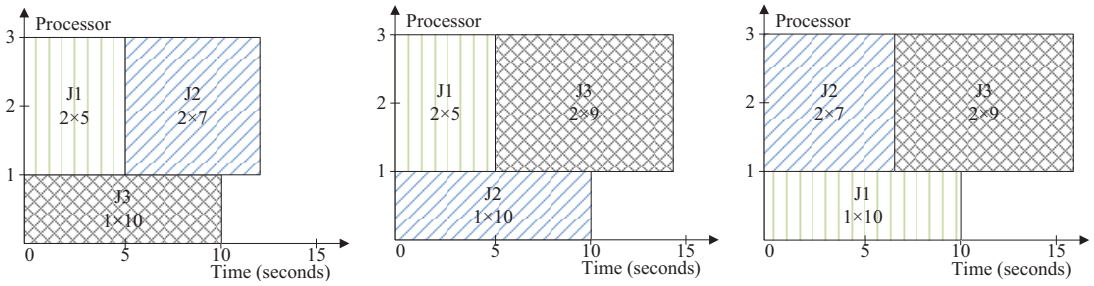
In this section, we introduce our basic idea before designing our resource allocation scheme. First, we study an example to explore how a resource allocation scheme affects the average turnaround time of moldable jobs, and offer a suggestion to allocate processors to achieve higher shortening runtime. Then, we propose an indicator named RP, which can clearly express the shortening runtime when we allocate an extra processor.

4.1 An Example

In moldable job scheduling, an adjustable number of processors can be allocated to moldable jobs, therefore leading to a different job runtime. Here, we study an example to explore how the average turnaround time of moldable jobs is affected by different processor allocation strategies. Consider a cluster with three processors and three jobs (*J1*, *J2*, and *J3*) to be scheduled. These jobs arrive at 0 seconds and can be scheduled with one processor or two processors, and their runtime, when with different numbers of processors, is shown in Table 1. For simplicity, we assume there are five processors (not necessarily equal to the total number of processors in the cluster) that can be allocated to these three jobs (we allocate one processor to one job, and two processors to two jobs each). Then, there could be three allocation strategies as shown in Figure 3.

Table 1: Runtime of jobs when one or two processors are allocated to jobs.

Job name	*Runtime* (1 *processor*)	*Runtime* (2 *processors*)
J1	10 *seconds*	5 *seconds*
J2	10 *seconds*	7 *seconds*
J3	10 *seconds*	9 *seconds*



(a) Strategy A (b) Strategy B (c) Strategy C

Figure 3: There are three possible processor allocation strategies in our example. Among them, Strategy A achieves the shortest average turnaround time.

In Strategy A (Figure 3(a)), one processor is allocated to *J3*, while two processors each are allocated to *J1* and *J2*. The average turnaround time of this strategy is 9 seconds. When adopting Strategy B and Strategy C (Figure 3(b) and 3(c)), the average turnaround times are 9.67 seconds and 11 seconds, respectively. With the same number of processors (five processors) allocated to moldable jobs (*J1*, *J2*, and *J3*), different allocation strategies can result in different average turnaround times. Among these strategies, Strategy A performs the best. Next, we analyze why Strategy A performs the best and offer a suggestion on achieving the shortest average turnaround time of moldable jobs when allocating processors.

As we know, the turnaround time of a job is composed of its waiting time and runtime. If we reduce the runtime of a job, this also means we reduce the waiting time of jobs waiting for the resources occupied by this job. For example, in Strategy B, *J3*'s waiting time is dependent on *J1*'s runtime (5 seconds), and thus its turnaround time is 14 seconds. However, in Strategy C, *J3*'s waiting time is prolonged because *J2*'s runtime is longer than *J1*'s. Therefore, with a limited number of processors, our suggestion reduces the runtime of jobs as far as possible, which can actually reduce their turnaround time.

Then, our question is, how can we reduce the runtime of moldable jobs? In moldable job scheduling, job runtime is shortened when one extra processor is allocated to the job. In our example, there are five processors to allocate to three different jobs. If an extra processor is allocated to *J1*, *J2*, and *J3*, which have occupied one processor already, the shortening runtime is 5, 3, and 1 second, respectively. Strategy A, allocating two processors to *J1* and *J2*, achieves the highest shortening runtime (8 seconds in total), and thus performs the best among all the three strategies. Therefore, in moldable job scheduling, our suggestion can be translated as allocating processors to jobs with higher shortening runtime can reduce the turnaround time of jobs.

4.2 Revenue per Processor

In the previous section, we suggest allocating processors to moldable jobs with higher shortening runtime, so as to reduce the average turnaround time of jobs. Now, we propose an indicator to express a job's shortening runtime when one extra processor is allocated to it, and analyze the characteristic of this indicator.

We assume the runtime of job J_i on n processors is $T_{J_i}(n)$. As long as we allocate one extra processor, it will achieve $T_{J_i}(n) - T_{J_i}(n+1)$ runtime shortened. In fact, the shortening runtime of a job can be regarded as the revenue in resource allocation. Thus, we call the shortening runtime with an extra allocated processor *Revenue per Processor* (RP). If we take RP as the indicator, it can express how much revenue of shortening runtime of a job can be achieved before an extra processor is allocated.

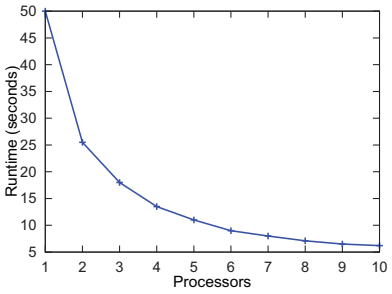


Figure 4: An example job in which the runtime and RP both decrease with more allocated processors.

Figure 4 shows the correlation between job runtime and the number of processors of an example job, in which the runtime decreases monotonously when 1 to 10 processors are allocated to the job. This means that RP is always greater than 0 with more processors allocated to the job. Meanwhile, RP decreases monotonously when we increase the number of processors (e.g., RP is about 25 when 1 processor is allocated to the job, while it is about 2 when 5 processors are allocated to the job). This means that RP of a job is highest when one processor is allocated to the job. These two characteristics allow us to conclude that, if we hope to achieve the highest shortening runtime when we allocate processors to moldable jobs, we can first allocate one processor to each job, and then consider allocating processors one after another to the job with the highest RP.

5. HIGHEST REVENUE FIRST ALLOCATION SCHEME

In the previous section, we suggest allocating processors to achieve the highest shortening runtime, in order to reduce average turnaround time. We propose an indicator named RP to express the shortening runtime of every job when an extra processor is allocated to it. In our resource allocation scheme, we want to design an algorithm which can effectively allocate processors to the jobs achieving the highest shortening runtime based on RP.

When there is one job or multiple jobs to be scheduled in a waiting queue, we should first initialize the number of

allocated processors of each job with one processor, and then prepare a certain budget of processors and allocate them to these moldable jobs. We define the processor budget as the number of processors to be allocated to queued jobs. Basically, the processor budget can be flexible (greater or less than the total number of processors in a cluster). If we set a higher budget, queued jobs are likely to obtain more processors with less runtime, while a lower budget means jobs may occupy fewer resources with longer runtime. We set a parameter α to express the processor budget's percentage of the total number of processors in a cluster. Specifically, α can be greater or less than 1(e.g., α in the example in section 4.1 is $\frac{5}{3}$, the total number of processors is 3, so the processor budget is $\frac{5}{3} * 3 = 5$. When α is greater than 1, queued jobs are likely to obtain more processors with less runtime), and it is typically set to 1 in existing schemes. Meanwhile, an apparent restriction on the number of processors of each job is that it cannot surpass the total number of processors in a cluster. If we use the parameter *threshold* to denote the maximum percentage of processors in a cluster that a job can use, then $threshold \leq 1$.

In each allocation turn, it allocates processors in descending order of RP of considering jobs (all queued jobs are considering jobs at the beginning). Whenever the number of processors allocated to a job reaches its threshold, we remove this job from considering jobs in this turn, and the allocation will continue until there are no unallocated processors left in the processor budget or no considering jobs left. In this way, processors are allocated one after another to the moldable job with the highest shortening runtime. Every time the state of the waiting queue changes (a job joins the queue or leaves the queue), we need to start a new allocation for queued jobs by invoking our algorithm. Specifically, cluster administrators set the values of α and *threshold* according to cluster's load (in our experiments, we will analyze how these two values affect scheduling performance under different loads).

Algorithm 1 Highest Revenue First Algorithm

Require: The runtime function of job J_i on x processors is $T_{J_i}(x)$. There are n jobs in the queue and the total number of processors in the cluster is m. The percentage of the processor budget in an allocation turn is α. The maximum percentage of processors in the cluster occupied by a job is *threshold*.

Ensure: The number of processors x_i of jobs

1: initialize considering jobs with all jobs J_1 to J_n
2: **for** considering jobs **do**
3: $x_i \leftarrow 1$
4: **end for**
5: **for** $k \leftarrow 1$ to $\alpha * m$ **do**
6: find the job J_i of considering jobs with the highest $RP_{J_i}(x_i) = T_{J_i}(x_i) - T_{J_i}(x_i + 1)$
7: **if** $x_i < threshold * m$ **then**
8: $x_i \leftarrow x_i + 1$
9: **else**
10: remove J_i from considering jobs and redo this loop
11: **end if**
12: **end for**

The algorithm is shown in Algorithm.1. In Line 1, we initialize the considering jobs with all queued jobs. Then, in Lines 2 to 4, we initialize all considering jobs with one allocated processor. In Line 5, we iterate a loop for $\alpha * m$ times (m is the number of processors in the cluster), which is to allocate $\alpha * m$ processors (processor budget) for all considering jobs. Lines 6 to 11 present that in each loop, we find the job with maximum RP and allocate one processor of all $\alpha * m$ processors. If the number of processors of any considering job has reached its threshold($threshold * m$), the job will be removed from the considering jobs. Because we can store the greatest RP of all n jobs in a big heap, whose seeking complexity is $log_2 n$, the overall computing complexity of this algorithm is $\alpha * m * log_2 n$. After invoking Algorithm 1, we will acquire a list of x_i indicating the number of processors allocated to every job.

This algorithm always finds the job with the highest RP and allocates an extra processor to it. By integrating this algorithm as our resource allocation scheme, we call our scheme *highest revenue first* (HRF), which allocates processors to achieve higher shortening runtime, thus reducing average turnaround time in moldable job scheduling.

6. EXPERIMENTS

In this section, we fulfil the scheduling policies of moldable jobs integrating different resource allocation schemes (HRF and other state-of-the-art schemes), and conduct simulated experiments on them. First, we adjust the key parameters in our resource allocation algorithm to show how they affect scheduling performance. Then, we compare the performance of the polices adopting HRF with other state-of-the-art schemes.

6.1 Experiment Setup

6.1.1 Environment

Our experiments are carried out on a server equipped with Intel Xeon E5-2670 CPU, and the OS is RHEL 6.2. We use a variable to denote the number of spare processors (128 in total) in our simulated cluster. When the number of spare processors can satisfy a job, we subtract the number of allocated processors from the variable to simulate the execution of the job.

6.1.2 Traces

Cirne01 [19] is a moldable job model that can synthesize moldable job traces. In Downey mode, synthetic traces are composed of job records including the key fields that we are concerned with (1) job name, (2) submission time, (3) L, (4) A, (5) σ. With these fields, we can easily work out the runtime of jobs on any number of processors. We use Cirne01 adopting the pattern ANL (Argonne National Laboratory SP2) to generate 200 moldable jobs in our experiments. We simulate jobs with different submission rate at about 2000, 1000 and 600 seconds per job, which represent light, middle, and heavy load, respectively.

6.1.3 Comparative Scheduling Policies

In our experiments, we fulfill our proposed resource allocation scheme HRF. In addition, as a comparative, we fulfill state-of-the-art schemes denoted as SbmGrdy (*Submit-time Greedy*), SbmFair (*Submit-time Fair*), and SchFair (*Schedule-time Fair*) proposed in [1], respectively. Specifically, HRF is a scheme to decide the number of processors at schedule-time. By combining these schemes with different job se-

Table 2: The scheduling policies in our evaluation.

Scheme	*Highest Revenue First*	*Submit − time Greedy*	*Submit − time Fair*	*Schedule − time Fair*
FCFS	*HRFFcfs*	*SbmGrdyFcfs*	*SbmFairFcfs*	*SchFairFcfs*
Backfilling	*HRFBf*	*SbmGrdyBf*	*SbmFairBf*	*SchFairBf*

lection schemes (we choose FCFS denoted as Fcfs and aggressive backfilling as Bf in this experiment), we have eight moldable job scheduling policies as listed in Table 2. We will compare scheduling performance of these policies in the following evaluations.

6.1.4 Performance Metrics

In a cluster, there are several metrics to express scheduling performance, such as *average turnaround time* (ATT), *average bounded slowdown* (ABS), and *utilization* (U) as proposed in [20] in their evaluation. However, ABS and U are not suitable to evaluate the performance of moldable jobs for the following two reasons.

The expression of slowdown of a job in ABS is $\frac{turnaround\ time}{runtime}$. If we optimize the resource allocation of this job, its runtime and turnaround time may both decrease (which means the performance of this job is improved), but its slowdown may increase if the decreasing ratio of runtime is less than that of turnaround time. Thus, ABS is not suitable to reflect the performance of moldable jobs.

The expression of another metric U of a cluster is

$$\frac{\sum_{i=1}^{m} parallelism_i \times runtime_i}{N \times (LastJobLeavingTime - FirstJobArrivingTime)}$$

where m refers to total number of jobs and N refers to total number of processors in the cluster. As with ABS, U may decrease though the runtime of all jobs decreases when resource allocation is optimized. Therefore, U is also not suitable to be a performance metric for moldable jobs.

Turnaround time (also known as response time) is the time between a job being submitted and when it finishes, which is directly perceived by job users. In this paper, our resource allocation scheme aims to reduce the *average turnaround time* of all moldable jobs. Therefore, we apply ATT in our experiments to evaluate the performance of different scheduling policies. Turnaround time of a job is composed of job waiting time and job runtime. To further understand how different scheduling policies or key parameters in our resource allocation algorithm affect ATT, we also use *average waiting time* (denoted as AWT) and *average runtime* (denoted as ART) in our evaluations.

6.2 Evaluation Results with Adjusting Parameters

There are two key parameters in Algorithm.1 adopted by HRF, α, and *threshold*. They determine how many processors are allocated in an allocation turn (processor budget) and how many processors a job can maximally use, respectively. Now, we try to adjust these two parameters and observe how they affect the performance of jobs when adopting HRFFcfs and HRFBf. The experiments are carried out under different loads of jobs.

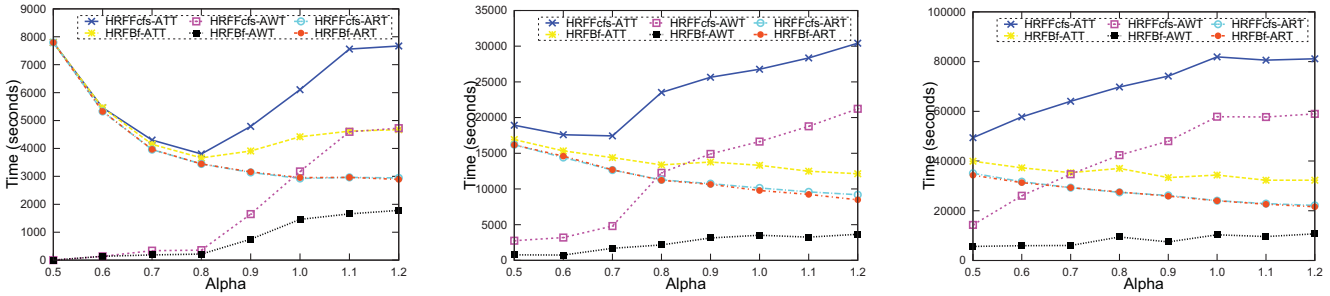
First, we keep *threshold* at 0.9 and adjust the value of α in the range from 0.5 to 1.2. The results presented in Figure 5 show that, for both HRFFcfs and HRFBf, AWT increases while ART decreases with increasing α. This is possibly because greater α will allocate more processors to moldable jobs, thus reducing runtime of jobs. However, allocating more processors to preceding jobs will delay the time for subsequent jobs to obtain resources, thus increasing the waiting time of jobs. The trend of ATT of HRFFcfs in Figure 5(a) and 5(b) decreases at the beginning and then increases, where the shortest ATT happens when α is 0.8 and 0.7. In Figure 5(c), HRFFcfs monotonously increases with greater α. That is to say, the best choice of α may depend on cluster load. Typically, when adopting the FCFS scheme, smaller α can achieve better performance in heavier load clusters. The trend of ATT of HRFBf decreases at the beginning and then increases in Figure 5(a), while monotonously decreasing with greater α in Figure 5(b) and 5(c). This result demonstrates that when adopting a backfilling scheme, greater α is more suitable in heavier load clusters.

Second, we keep α at 1 and adjust the value of *threshold* in the range from 0.1 to 1, and the results are presented in Figure 6. Similar to the results of adjusting α, for both HRFFcfs and HRFBf, AWT increases while ART decreases with increasing *threshold*. This is because lower *threshold* can reduce waiting time by saving resources taken by each job, but prolongs runtime because of a restricted number of allocated processors. Comparing the results under different load in Figure 6, we find that ATT is very fluctuant at the beginning (when $threshold < 0.3$) and then becomes stable (when $threshold > 0.3$). The reason could be that, ART is very high and AWT is very low when at small *threshold*, which makes ATT (the sum of AWT and ART) deviate from stable significantly, especially in the scenario shown in Figure 6(a). Based on these results, we would suggest setting the parameter *threshold* at a relatively larger level (larger than 0.3) to achieve stable performance of jobs.

6.3 Evaluation Results with Comparative Scheduling Policies

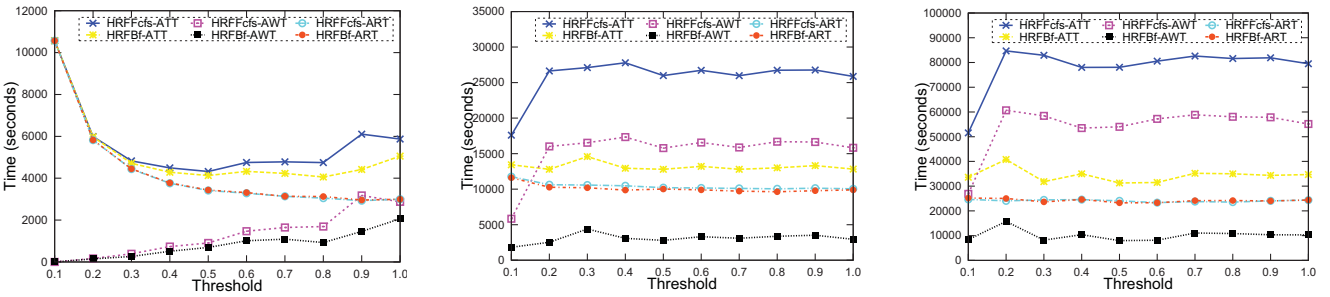
In this section, we compare the performance of scheduling policies listed in Table 2. The experiments above tell us that the performance of scheduling policies is significantly affected by using different parameters (α and *threshold*) in the resource allocation algorithm. To explore whether HRF performs better in most cases, we conduct our evaluations on these scheduling policies with different combinations of parameters. There are 56 combinations of parameters in all (we set α from 0.5 to 1.2 and *threshold* from 0.4 to 1, thus the number of combinations is the product of 8 and 7). Then, we treat the mean performance of any scheduling policy under specialized load as its combined performance, and we get the results denoted as combined-ATT, combined-AWT, and combined-ART, respectively (e.g., there are 56 results of ATT of HRFFcfs under light load, then we use the mean value of these results as combined-ATT of HRFFcfs under light load). The performance results of our comparative policies are presented in Figure 7.

For all the cases with different load of jobs, the relative better or worse performance of any two scheduling policies present in a similar manner. Taking Figure 7(b) as an exam-



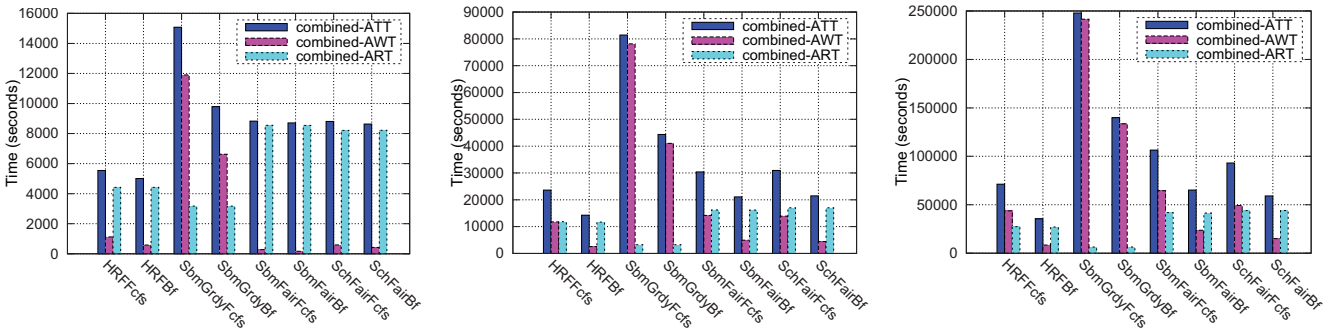
(a) light load of jobs (b) middle load of jobs (c) heavy load of jobs

Figure 5: The effect of α on ATT, AWT, and ART of jobs with light, middle, and heavy load of jobs when $threshold = 0.9$. The results show that the optimal α for performance is dependent on the load of jobs.



(a) light load of jobs (b) middle load of jobs (c) heavy load of jobs

Figure 6: The effect of *threshold* on ATT, AWT, and ART of jobs with light, middle, and heavy load of jobs when $\alpha = 1$. The results show that the performance is not stable until $threshold > 0.3$.



(a) light load of jobs (b) middle load of jobs (c) heavy load of jobs

Figure 7: Comparing combined-ATT, combined-AWT, and combined-ART of different scheduling policies with light, middle, and heavy load of jobs, HRFBf always performs best.

ple, combined-ATT of SbmGrdyFcfs is the longest, which is 81,444 seconds, 2.63 times SchFairFcfs and 3.45 times HRFFcfs. Though SbmGrdyFcfs uses the shortest combined-ART among all scheduling policies adopting FCFS, it performs the poorest in combined-ATT. This is because it adopts a greedy manner when allocating processors to jobs, and every job occupies as many resources as possible. The greedy manner ignores the limitation of the processor budget and thus leads improper resource allocation to jobs. SbmFairFcfs is the policy adopting a fair manner, which takes account of job weights (the sequential runtime of jobs) as the reference for allocating processors, using only 30,400 seconds in combined-ATT. SchFairFcfs is the variant of SbmFairFcfs whose difference is performing allocation at schedule-time, and its ATT is 30,935 seconds. The result of SchFairFcfs is slightly worse than SbmFairFcfs in Figure 7(b) but better in Figure 7(a) and Figure 7(c). The reason for SchFairFcfs performing better than SbmFairFcfs in more cases is probably because it can cover the variation of queued jobs between submit-time and schedule-time. HRFFcfs achieves even better result, with combined-ATT of 23,617 seconds, which is only 29%, 77.7%, and 73.3% of combined-ATT of SbmGrdyFcfs, SbmFairFcfs and SchFairFcfs, respectively. HRFFcfs performs better than policies adopting state-of-the-art resource allocation schemes at combined-ATT because it always allocates processors to the job which will achieve the highest RP at schedule-time.

The result in Figure 7(b) also reveals that, when adopting the same resource allocation scheme, policies adopting backfilling perform better than policies adopting FCFS scheme. For example, combined-ATT of HRFBf is 13309 seconds, which is 50.3% better than HRFFcfs (the reduction of combined-ATT is typically contributed by combined-AWT). The reason is that although moldable jobs are more flexible than rigid jobs, they also cause resource fragmentation. Backfilling can fill some fragmentation (especially effective in reducing waiting time), which helps moldable job scheduling policies achieve better performance.

Comparing results in Figure 7(a), 7(b), and 7(c) with different load of jobs, we find that the heavier the load of jobs, the longer combined-ATT is. This can be explained because heavier load means that fewer resources can be allocated to jobs, which directly leads to longer runtime. Meanwhile, more jobs aggravate congestion, and also bring about longer waiting time of jobs (hence eventually longer turnaround time). However, although under different load, the relative relationship of performance between different policies is similar. For example, HRFFcfs and HRFBf perform the best compared with state-of-the-art scheduling policies adopting FCFS and backfilling, respectively. That is to say, scheduling policy adopting our resource allocation scheme can significantly improve performance of moldable jobs in the cluster under both light and heavy load.

6.4 Summary

In the above experiments, we explore how *threshold* and α in our resource allocation algorithm affect the performance of scheduling, and offer some suggestions on how to set them in practical scheduling. Then, we compare the *average turnaround time* under different scheduling policies. Results show that the scheduling policy integrating HRF with backfilling can always achieve the shortest *average turnaround time* among all policies (up to 71% improvement compared with other policies).

7. FUTURE WORK

Moldable job scheduling requires the accurate prediction of the runtime of jobs with different allocated processors. However, to the best of our knowledge, such prediction is absent. A possible solution is making predictions based on the runtime traces of jobs. We aim to study how to design the prediction and how the accuracy of prediction affects scheduling performance. We offer a suggestion on how to set the parameters (i.e., *threshold* and α) of HRF under different load of jobs. By far, the setting of parameters requires the intervention of administrators. We will make our algorithm adaptive to the load of jobs automatically in further study.

8. CONCLUSION

Moldable jobs are the sort of parallel jobs that can adjust their number of allocated processors before running in clusters. Because of their flexibility, moldable jobs have more potential to improve the performance of jobs.

In this work, we study how *average turnaround time* is affected by resource allocation, and propose to allocate processors to jobs with the highest revenue of shortening runtime. We propose an indicator RP to express such revenue. Then we devise a resource allocation scheme named HRF deciding the number of processors for moldable jobs based on RP, which is proved to outperform scheduling policies adopting other state-of-the-art resource allocation schemes in our simulations.

9. ACKNOWLEDGEMENTS

The research is supported by National Science Foundation of China under grants No.61232008, National 863 Hi-Tech Research & Development Program under grants No.2014AA01A302 and No.2015AA011402, and Research Fund for the Doctoral Program of MOE under grant No.20110142130005.

10. REFERENCES

[1] S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling technology for moldable scheduling of parallel jobs. In *Proceedings of CLUSTER*, pages 92–99. IEEE, 2003.

[2] D. Tsafrir, Y. Etsion, and D. G Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *TPDS*, 18(6):789–803, 2007.

[3] W. Tang, D. Ren, Z. Lan, and N. Desai. Toward balanced and sustainable job scheduling for production supercomputers. *PARCO*, 39(12):753–768, 2013.

[4] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *TPDS*, 12(6):529–543, 2001.

[5] D. Talby and D. G Feitelson. Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. In *Proceedings of IPDPS*, pages 513–517. IEEE, 1999.

[6] D. A. Lifka. The anl/ibm sp scheduling system. In *Proceedings of workshops on Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer, 1995.

[7] B. G. Lawson, E. Smirni, and D. Puiu. Self-adapting backfilling scheduling for parallel systems. In *Proceedings of ICPP*, pages 583–592. IEEE, 2002.

[8] A. A. Chandio, K. Bilal, N. Tziritas, Z. Yu, Q. Jiang, S. U. Khan, and C. Xu. A comparative study on resource allocation and energy efficient job scheduling strategies in large-scale parallel computing systems. In *Proceedings of CLUSTER*, pages 1349–1367. IEEE, 2014.

[9] W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. *JPDC*, 62(10):1571–1601, 2002.

[10] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Effective selection of partition sizes for moldable scheduling of parallel jobs. In *Proceedings of HiPC*, pages 174–183. Springer, 2002.

[11] G. Sabin, M. Lang, and P. Sadayappan. Moldable parallel job scheduling using job efficiency: an iterative approach. In *Proceedings of workshops on Job Scheduling Strategies for Parallel Processing*, pages 94–114. Springer, 2007.

[12] G. Utrera, S. Tabik, J. Corbalan, and J. Labarta. A job scheduling approach for multi-core clusters based on virtual malleability. In *Proceedings of EuroPar*, pages 191–203. Springer, 2012.

[13] Y. He, W. Hsu, and C. E. Leiserson. Provably efficient two-level adaptive scheduling. In *Proceedings of workshops on Job Scheduling Strategies for Parallel Processing*, pages 1–32. Springer, 2007.

[14] H. Sun, Y. Cao, and W. Hsu. Efficient adaptive scheduling of multiprocessors with stable parallelism feedback. *TPDS*, 22(4):594–607, 2011.

[15] Load sharing facility, `http://www-03.ibm.com/systems/platformcomputing/products/lsf`.

[16] A. B. Downey. A model for speedup of parallel programs. Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley, Jan 1997.

[17] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of SIGMETRICS*, pages 104–113. ACM, 1988.

[18] Y. Yuan, G. Yang, Y. Wu, and W. Zheng. Pv-easy: a strict fairness guaranteed and prediction enabled scheduler in parallel job scheduling. In *Proceedings of HPDC*, pages 240–251. ACM, 2010.

[19] W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. In *Proceedings of workshops on Workload Characterization*, pages 140–148. IEEE, 2001.

[20] J. Chen, B. B. Zhou, C. Wang, P. Lu, P. Wang, and A. Y. Zomaya. Throughput enhancement through selective time sharing and dynamic grouping. In *Proceedings of IPDPS*, pages 1183–1192. IEEE, 2013.