Scalable Hierarchical Scheduling for Malleable Parallel Jobs on Multiprocessor-based Systems

Yangjie Cao[1], Hongyang Sun[2], Depei Qian[3], and Weiguo Wu[4]

[1]School of Software Engineering, Zhengzhou University, China.
[2]School of Computer Engineering, Nanyang Technological University, Singapore
[3]School of Computer Science and Engineering, Beihang University, China
[4]School of Electronic and Information Engineering, Xi'an Jiaotong University, China
caoyj@zzu.edu.cn, sunh0007@ntu.edu.sg, depeiq@buaa.edu.cn, wgwu@mail.xjtu.edu.cn

Abstract: The proliferation of multi-core and multiprocessor-based computer systems has led to explosive development of parallel applications and hence the need for efficient schedulers. In this paper, we study hierarchical scheduling for malleable parallel jobs on multiprocessor-based systems, which appears in many distributed and multilayered computing environments. We propose a hierarchical scheduling algorithm, named AC-DS, that consists of a feedback-driven adaptive scheduler, a desire aggregation scheme and an efficient resource allocation policy. From theoretical perspective, we show that AC-DS has scalable performance regardless of the number of hierarchical levels. In particular, we prove that AC-DS achieves $O(1)$-competitiveness with respect to the overall completion time of the jobs, or the makespan. A detailed malleable job model is developed to experimentally evaluate the effectiveness of the proposed scheduling algorithm. The results verify the scalability of AC-DS and demonstrate that AC-DS outperforms other strategies for a wide range of parallel workloads.

Keywords: Hierarchical scheduling; Feedback-driven adaptive scheduling; Malleable parallel jobs; Multiprocessors

1 Introduction

Multi-core and multiprocessor-based computers are increasingly used to support a wider range of parallel and distributed computing environments, such as multi-clusters, grid and more recently the cloud computing infrastructures. In these environments, the productivity and performance gains largely depend on the effective exploitation of application parallelism across the available computing resources. Due to the increasing scale and the dynamic nature of these modern computing platforms, there is a need for more efficient scheduling strategies in order to effectively allocate the available computing resources to the parallel applications [1, 2].

Since most multi-clusters, grid and cloud computing platforms have been built on top of the existing local resource management systems, a hierarchy of schedulers has been naturally established [3, 4, 5]. A typical scheduling prototype in grid platforms is shown in Fig. 1. At the highest level there is a grid-level scheduler responsible for the management of the overall resources but it lacks the detailed knowledge about the local scheduling environment, where the parallel jobs will be eventually executed. To understand the scheduling complexity in such a hierarchical structure, one should note that the process involves several phases of scheduling at different levels, including determining available computing resources, determining task requirements, invoking a scheduler to determines how many processors are allocated to each task, and monitoring task execution.

To date, much work has been reported in the two-level scheduling paradigm [6, 7, 8, 9, 10, 11], where a global scheduler focuses on efficiently allocating computing resources to a partition and a local scheduler focuses on effectively using the allocated resources to schedule the ready tasks in each partition. Although these studies address certain important aspects of hierarchical scheduling, such as fairness and efficiency of resource allocation, little attention has been paid to the scalability of the scheduling algorithms with increasing hierarchical complexity in large-scale systems, such as today's grid or cloud computing platforms. In this paper, we present a more general hierarchical and adaptive scheduling model, where the structure of the system and the number of hierarchical levels are not restricted. Each job is submitted into the system from a leaf node in the hierarchy, and each intermediate node in the hierarchy contains either a group of jobs or an aggregation of job groups. The objective is to design hierarchical scheduling algorithms that allocate the available processors

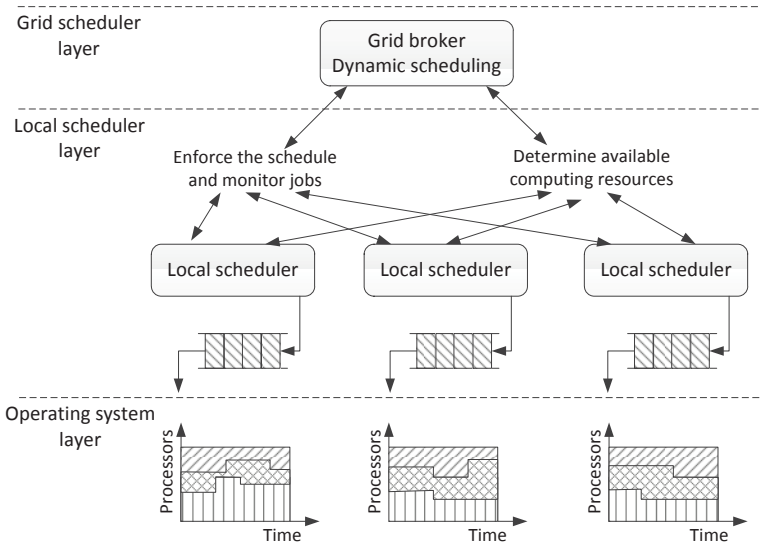


Figure 1: A typical scheduling prototype in hierarchical grid platforms.

from the root of the hierarchy down to the jobs in the leaf nodes through all intermediate levels to minimize the overall completion time, or the makespan.

The tree-structured view of a system is a useful and general notion since its succinctly describes many different system architectures and helps us focus on the important similarities among different systems. For instance, a virtual tree-structured approach is recently used to model and evaluate the performance of grid computing infrastructures [12, 13, 14]. While the machines in real systems often exhibit different forms of heterogeneity, we restrict our attention to the ones that consist of homogeneous processors or cores. Due to advancement of virtualization technologies, we believe that the differences in these individual processors or cores can be effectively resolved, and are therefore hidden from the application interface, which makes our model a reasonable representation of many typical hierarchical systems.

To formalize the scheduling problem in our study, we adopt the online and non-clairvoyant scheduling model, which requires the algorithm to operate in an online manner, that is, to make irrevocable scheduling decisions without any knowledge about the jobs' future characteristics, such as their release times, processor requirements and remaining work. Compared to the previous work, we present a general hierarchical scheduling paradigm for multiprocessor-based systems, which provides a flexible approach to hierarchically allocating computing resources for a set of parallel applications. In addition, to verify the effectiveness of different hierarchical scheduling algorithms, we develop a malleable parallel job model using a generic set of internal parallelism variations, which proves to be an effective tool to evaluate the performance of adaptive scheduling algorithms. The main contributions of our paper are summarized as follows:

- Integrating a feedback-driven adaptive scheduler, a desire aggregation scheme and a resource allocation policy, we propose a hierarchical scheduling algorithm AC-DS, which scalably achieves $O(1)$-competitiveness with respect to the makespan regardless of the number of hierarchical levels. In addition, we provide a generalized analysis framework for any hierarchical scheduling algorithm with certain performance guarantees.

- Based on a new malleable parallel job model, we conduct experiments to evaluate and compare three hierarchical scheduling algorithms. The results demonstrate that AC-DS achieves better and more stable performance than the other strategies, and in general feedback-driven algorithms outperform the simple strategy based on equal resource partitioning for a wide range of malleable workloads.

The rest of this paper is organized as follows. In Section 2, we formally define the hierarchical scheduling model. Section 3 describes the hierarchical algorithm AC-DS and analyzes its performance with respect to makespan. In Section 4, a malleable job model is first presented followed by the experimental evaluations that compare different scheduling strategies. In Section 5, we review some related work. Finally, Section 6 concludes the paper and discusses some future directions.

2 Hierarchical Scheduling Model

This section formally describes the hierarchical scheduling model. A set of n parallel jobs, denoted by $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$, arrive over time in an online manner. The jobs are assumed to be malleable, that is, they can be executed with a variable number of processors during runtime [15]. (See Section 5 for a detailed classification of parallel jobs.) Moreover, the parallelism or the number of ready threads of a job can also change during their executions. At any time t, suppose the parallelism of a job J_i is $h_i(t)$ and the job is allocated $a_i(t)$ processors, the execution rate $\Gamma_i(t)$ for the job, that is, the amount of work done per unit of time, is given by $\Gamma_i(t) = \min\{a_i(t), h_i(t)\}$. For each job $J_i \in \mathcal{J}$, let r_i denote its release time. We also define w_i and l_i to be its total work and total span, which are two important parameters representing the time to execute the job with one processor and infinite number of processors, respectively.

The hierarchical system is organized as a tree structure with a single root and an arbitrary number of levels, denoted by K. Each job $J_i \in \mathcal{J}$ is released into the system at time r_i from one of the leaf nodes at the bottom of the hierarchy. The problem is to design an online scheduling algorithm that allocates a total number of P processors from the root of the tree down to the available jobs through all intermediate levels without any knowledge of the future job arrivals. The objective is to minimize the overall completion time of the jobs, or the makespan. Note that when $K = 2$, i.e., there are two levels, the problem is reduced to the classical makespan scheduling problem for malleable parallel jobs [7, 9, 16, 17]. Hence, our hierarchical model represents a more general setting for the multiprocessor scheduling problem.

While the tree structure reflects the characteristics of the system hierarchy, it is assumed to be fixed and hence cannot be altered by the online algorithm during the executions of the jobs. Moreover, we require an online algorithm to be non-clairvoyant, that is, it must make all scheduling decisions without knowing the characteristics of a job, such as its remaining work and span, as well as the future parallelism variations. Finally, the scheduling decisions at each intermediate node can only be made with direct feedbacks from its immediate children, without knowing the decisions at the other nodes. Therefore, the processors need to be allocated in a distributed manner. These additional challenges make the hierarchical scheduling problem significantly more complex than many classical scheduling problems, where only centralized decisions need to be made on a single root node with relatively flat system structures.

We evaluate the performance of our online scheduling algorithms using both theoretical analysis and simulation study. Theoretically, the performance is bounded using competitive analysis [18], which compares an online algorithm with the optimal offline scheduler which knows complete information of the jobs in advance. Suppose the makespan of an online algorithm for a job set $\mathcal{J}$ is $M(\mathcal{J})$ and the makespan of the optimal offline algorithm for the same job set is $M^*(\mathcal{J})$. Then the online algorithm is said to be c-competitive if $M(\mathcal{J}) \leq c \cdot M^*(\mathcal{J})$ holds for any job set $\mathcal{J}$. For the purpose of simulation, we develop a novel malleable parallel job model by augmenting an existing moldable job model [19] with a set of generic parallelism variations. We use the new model to drive the simulations and to evaluate the performance and scalability of our scheduling algorithms.

3 A Scalable Hierarchical Scheduling Algorithm

In this section, we present a hierarchical scheduling algorithm, which consists of an adaptive feedback-driven scheduler at the bottom level, a desire aggregation scheme at the intermediate levels, and a dynamic resource allocation policy for allocating the processor resources. We show that the algorithm achieves scalable performance with respect to the makespan, and we generalize its analysis framework to other scheduling algorithms that satisfy certain properties.

3.1 A Feedback-Driven Scheduler

We first present a bottom-level scheduler that interacts directly with the jobs and provides feedbacks to the higher level. For this purpose, we apply a feedback-driven scheduler, called A-Control (or AC for short) [9, 17], which predicts the processor requirements or desires for each job periodically after a pre-defined interval of time, commonly known as scheduling quantum. The processor desires are then provided to the higher-level scheduler for the readjustment of the jobs' processor allocations in the next quantum.

Specifically, AC calculates the processor desire for a job in the next quantum based on the information collected in the current quantum, namely, the job's average parallelism. Suppose in any scheduling quantum q, which starts at time t_q and lasts for L amount of time, job J_i completes $w_i(q)$ amount of work and reduces its span by $l_i(q)$. Due to the time-varying characteristic of the job's parallelism, the two parameters can be obtained by $w_i(q) = \int_{t_q}^{t_q+L} \Gamma_i(t)dt$, and $l_i(q) = \int_{t_q}^{t_q+L} \Gamma_i(t)/h_i(t)dt$ [9], where $\Gamma_i(t)$ and $h_i(t)$ denote the execution rate and the parallelism of job J_i at time t, respectively. Then the average parallelism of the job during this quantum

is given by $A_i(q) = w_i(q)/l_i(q)$, which is a well-known approach for calculating the average parallelism of a job. AC then directly utilizes this average parallelism as the job's processor desire for the next quantum. That is, the processor desire of the job for quantum $q+1$ is set to be

$$d_i(q+1) = A_i(q) \ . \tag{1}$$

The rationale behind this simple desire-calculation strategy is as follows: Although the average parallelism of a job could change over time, its current parallelism is likely to be still representative of the job's resource requirement for the near future under reasonable assumptions about the quantum length and the job's parallelism variation.[1] The initial processor desire for the job in the first scheduling quantum when it is just submitted into the system is simply set to be 1. For the ease of analysis, we say that job J_i is satisfied in quantum q if the number of processors $a_i(q)$ actually allocated to the job is at least its processor desire, i.e., $a_i(q) \geq d_i(q)$. Otherwise, the job is said to be deprived if $a_i(q) < d_i(q)$.

3.2 A Desire Aggregation Scheme

We now present a desire aggregation scheme for any intermediate node that takes the processor desires from the lower levels and provides a feedback to the higher level. The scheme, called Desire-Sum (or DS for short), collects the desires from the immediate children of the node, sums them up as its own desire and reports the sum to the parent node.

Formally, suppose an intermediate node n_i^k at level k has m immediate children at level $k-1$ in the system hierarchy. At the end of each quantum q, the m children report to node n_i^k their processor desires for quantum $q+1$, which are denoted as $\{d_1^{k-1}(q+1), d_2^{k-1}(q+1), \cdots, d_m^{k-1}(q+1)\}$. Then, node n_i^k calculates the aggregate desire $d_i^k(q+1)$ for quantum $q+1$ as follows:

$$d_i^k(q+1) = \sum_{j=1}^{m} d_j^{k-1}(q+1) \ . \tag{2}$$

For the hierarchical scheduling problem, all levels may not share the same quantum length. For instance, compared to the lower levels, a higher level may need a substantially longer quantum in order to reduce the overhead in the reallocation of resources. In this paper, we make the reasonable assumption that the quantum length at a particular level can only be an integral multiple of that at the immediate lower level. Suppose the quantum at level k has not expired when the nodes at level $k-1$ report their desires, then these desires will be discarded, and only the most recent ones from the lower levels are considered when the quantum at level k does expire. This is a reasonable strategy because as the number of levels increases, it is not likely that the execution status of the jobs in the distant past is still relevant to predict the future processor requirements.

3.3 A Processor Allocation Policy

Finally, to allocate processors to the nodes at each level including the jobs at the bottom level, we apply a fair and efficient policy, called Dynamic EQui-partitioning (or DEQ for short) [20]. DEQ allocates the processors received at any node to its immediate children based on their processor desires. Generally speaking, it attempts to give a fair share of processors to each child, but for efficiency it does not allocate more processors to a child than what the child desires. For ease of analysis, we allow fractional processor allocation as in [7, 8, 9, 10, 11]. This can be considered as time-sharing a processor among several concurrently running jobs.

Suppose the quantum for level k expires at the end of quantum q, and the node n_i^k receives a_i^k processors from the higher level at the beginning of quantum $q+1$. Let $N = \{n_1^{k-1}, n_2^{k-1}, \cdots, n_m^{k-1}\}$ denote the set of m children of node n_i^k. The processors are then allocated to the children nodes in N as described in Algorithm 1.[2]

As can be seen from the pseudocode, the algorithm allocates the processors by first satisfying the children with small processor desires in a recursive manner, and then it gives an equal share to the remaining children with large desires. In Line 3, the algorithm considers those children whose processor desires are not more than the current equal share $a_i^k / |N|$, and they will be satisfied (Lines 8-10). Then, the policy is recursively invoked by excluding the jobs already satisfied and the processors already allocated. As the new equal share may be increased, the process above will be repeated until all jobs are satisfied (Lines 1-2) or no more job can be satisfied. In the latter case, each of the remaining children will get the latest equal share (Lines 4-7).

[1] In [9, 17], the processor desire is set to be a linear combination of the average parallelism and the desire in the previous quantum, where the weight on the previous desire is determined by a user-defined convergence rate. For simplicity and performance, we set the convergence rate to be zero in this paper, so that the fastest convergence can be achieved.

[2] For convenience, we drop the quantum index $q+1$ for the processor desires and processor allocations.

Algorithm 1 DEQ(N, a_i^k)

```
1: if N = ∅ then
2:    return
3: S = {n_j^{k-1} ∈ N | d_j^{k-1} ≤ a_i^k / |N|}
4: if S = ∅ then
5:    for each n_j^{k-1} ∈ N do
6:       a_j^{k-1} = a_i^k / |N|
7:    return
8: else
9:    for each n_j^{k-1} ∈ S do
10:      a_j^{k-1} = d_j^{k-1}
11:   DEQ(N\S, a_i^k − Σ_{n_i^{k-1} ∈ S} a_j^{k-1})
```

In a straightforward implementation of the algorithm, each iteration scans all remaining jobs and compares their processor desires with the current equal share. In the worst case, only one job will be satisfied and hence the algorithm will be invoked m times. The time complexity of the algorithm is therefore $O(m^2)$.

Note that the DEQ policy is applied to all levels, including the jobs at the bottom level. At any particular level, it is only executed when the quantum for this level expires. The processors allocated to a node (or job) will stay with the node (or job) till the beginning of the next quantum when DEQ is invoked again. However, the lower levels may have smaller quantum lengths. Hence, the processors could be reallocated among the nodes (or jobs) at the lower levels more frequently than at higher levels.

3.4 Performance Analysis

Combining the adaptive feedback-driven scheduler AC, the desire aggregation scheme DS, and the processor allocation policy DES, we obtain a hierarchical scheduling algorithm, which we call AC-DS. In this section, we provide the performance analysis of the AC-DS algorithm when there is negligible cost for processor reallocation and all levels share the same quantum length. Specifically, we show that the competitive ratio achieved by AC-DS in this case is scalable, that is, it does not increase with the number of levels in the hierarchy. Experimental studies are performed for the more general cases with different quantum lengths and reallocation costs in the next section.

Before analyzing the performance of AC-DS, we first define two relevant concepts for the jobs. For any job $J_i \in \mathcal{J}$, we define t_i^S to be its total satisfied time, that is, the overall execution time of the job whenever the job is satisfied, and define a_i^T to be the job's total processor allocation, that is, the aggregate processor allocation the job receives throughout its execution. For convenience, we assume that the quantum length L is normalized to 1. Therefore, we can formally expressed the total satisfied time and the total processor allocation as $t_i^S = \sum_{q \in Q} [J_i \in S(q)]$ and $a_i^T = \sum_{q \in Q} a_i(q)$, where Q denotes the set of all scheduling quanta, $S(q)$ denotes the set of satisfied jobs in quantum q, and $[x]$ returns 1 if the proposition x is true and 0 otherwise.

We now introduce an important parameter, which is called the transition factor and is denoted by $c \geq 1$. This parameter indicates the maximum ratio on the average parallelism of any job over two adjacent quanta [9, 17]. Specifically, let $A_i(q)$ and $A_i(q+1)$ denote the average parallelism of job J_i in quantum q and $q+1$, respectively. Then the average parallelism of the job should satisfy $\frac{1}{c} \leq \frac{A_i(q)}{A_i(q+1)} \leq c$ for any quantum q.

The following lemma, which was formally proven in [9, 17], gives the bounds for the total satisfied time t_i^S and total processor allocation a_i^T of any job J_i scheduled under AC in terms of the job's transition factor, work, and span.

Lemma 1 *For any job $J_i \in \mathcal{J}$ scheduled by the AC scheduler, its total satisfied time t_i^S and total processor allocation a_i^T are given by*

$$t_i^S \leq (c+1) \cdot l_i \ , \tag{3}$$

$$a_i^T \leq (c+1) \cdot w_i \ , \tag{4}$$

where w_i and l_i denote the work and the span of job J_i, respectively, and c denotes the transition factor of the job. □

Note that the bounds shown in Lemma 1 hold for any job scheduled by AC, regardless of the desire aggregation scheme and the processor allocation policy used at the higher levels. We will now rely on these two bounds to show the makespan performance of AC-DS.

Theorem 1 *For the hierarchical scheduling problem with the same quantum length in all levels, the makespan $M(\mathcal{J})$ for a job set $\mathcal{J}$ scheduled by the AC-DS algorithm satisfies*

$$M(\mathcal{J}) \leq 2(c+1) \cdot M^*(\mathcal{J}) \ , \tag{5}$$

where $M^*(\mathcal{J})$ denotes the makespan of the job set scheduled by the optimal offline algorithm.

Proof. The performance is obtained by bounding the total satisfied time and the total deprived time of the last completed job in the job set, separately.

Let $J_k \in \mathcal{J}$ denote the last completed job in job set $\mathcal{J}$ scheduled by AC-DS. Then, the makespan is the same as the completion time of J_k, which includes its release time r_k, total satisfied time t_k^S, and total deprived time t_k^D. The total satisfied time of J_k, according to Inequality (3), is given by $t_k^S \leq (c+1) \cdot l_k$. When J_k is deprived, since all levels share the same quantum length, according to the desire aggregation scheme DS and the processor allocation policy DEQ, all the ancestors of J_k, including the root node, in the hierarchy are also deprived. This is because if any ancestor of J_k is satisfied, it could have satisfied all of its descendants, including J_k, and this property holds regardless of the number of levels. Hence, all P processors must be allocated to the jobs in this case due to the deprivation. Based on Inequality (4), the total deprived time of job J_k is therefore bounded by $t_k^D \leq \frac{\sum_{i=1}^{n} a_i^T}{P} \leq (c+1) \cdot \frac{\sum_{i=1}^{n} w_i}{P}$.

The makespan of the jobs, which is the completion time of J_k, is then given by $M(\mathcal{J}) = r_k + t_k^S + t_k^D \leq (c+1) \cdot (l_k + r_k) + (c+1) \cdot \frac{\sum_{i=1}^{n} w_i}{P}$. Since the optimal offline algorithm takes at least the span l_k time to complete job J_k and hence the whole job set after the release of J_k, so we have $M^*(\mathcal{J}) \geq l_k + r_k$. Also, we have $M^*(\mathcal{J}) \geq \frac{\sum_{i=1}^{n} w_i}{P}$, since this is the time needed to complete all work of the jobs even when all processors are efficiently utilized without any waste [7]. Based on these two lower bounds, the theorem is directly implied. □

Under the reasonable assumption that the jobs have smooth parallelism variations, that is, their transition factor c can be considered as a constant, Theorem 1 shows that the hierarchical scheduling algorithm AC-DS achieves $O(1)$-competitiveness in terms of the makespan of the jobs. Moreover, Theorems 1 also suggests that the competitive ratio of AC-DS does not increase with the number of levels in the hierarchy. Hence, the algorithm is scalable and can be used to schedule malleable parallel jobs in any hierarchical system with the same performance guarantee. The reason of such scalability comes from the nice properties of the algorithm's three components, namely, the performance guarantee of the AC scheduler in terms of each individual job, the effectiveness of the DS scheme for aggregating the processor desires at the intermediate nodes, and the efficiency of the DEQ policy for allocating the processors throughout the hierarchy.

Corollary 1 *The scheduling algorithm AC-DS scalably achieves $O(1)$-competitiveness with respect to the makespan regardless of the number of hierarchical levels.* □

3.5 Generalized Analysis Framework

From the analysis of the AC-DS algorithm, we can observe that its competitive ratio is mainly determined by the properties of the AC scheduler at the job level, while the DS algorithm and the DEQ algorithm are designed to maintain the performance in the presence of scheduling hierarchies. Based on this observation, we generalize its analysis in this section to other scheduling algorithms that can offer similar guarantees in terms of the running time and the processor allocations.

Let X-DS denote any hierarchical scheduling algorithm that uses scheduler X to calculate processor desires for each job and uses DS and DEQ for aggregating desires and allocating processors, respectively. As with the analysis of AC-DS, define t_i^S to be the total satisfied time and define a_i^T to be the total processor allocation for any job $J_i \in \mathcal{J}$ scheduled by X-DS. Moreover, the two parameters can be bounded in terms of the job's work and span as follows:

$$t_i^S \leq \alpha \cdot l_i \ , \tag{6}$$

$$a_i^T \leq \beta \cdot w_i \ . \tag{7}$$

By following the analysis given in the previous section, we can easily bound the performance of the X-DS algorithm, which is stated in the corollary below.

Corollary 2 *The scheduling algorithm X-DS achieves $(\alpha + \beta)$-competitiveness with respect to the makespan regardless of the number of hierarchical levels, provided that Inequalities (6) and (7) are satisfied for each job.* □

For instance, we can apply Corollary 2 to the algorithm AG-DS, which uses another feedback-driven scheduler, called A-Greedy (or AG for short) [16], at the bottom level. Like AC, AG is also a quantum-based scheduler, but employs a multiplicative increase and decrease strategy that either doubles or halves the processor desire for a job in the next quantum depending on the job's processor utilization in the current quantum. It was shown in [16] that the total satisfied time and the total processor allocation of any sufficiently large job under AG can be bounded by a constant factor in terms of the job's span and work respectively, i.e., $\alpha = \beta = O(1)$. Hence, Corollary 2 implies that AG-DS also achieves constant competitiveness with respect to makespan.

Despite achieving $O(1)$-competitiveness, the processor desires predicted by the AG scheduler are, however, less stable compared to that of AC, even for jobs with constant parallelism profile [9, 17]. This will inevitably affect the performance of the algorithm by introducing extra scheduling overhead. In the next section, we will evaluate the practical performances of these algorithms in the hierarchical scheduling environment under the more general setting with variable quantum lengths and reallocation costs over different levels.

4 Simulations

In this section, we empirically evaluate the performance of our hierarchical scheduling algorithm presented in the previous section. We first present a malleable parallel job model derived from a traditional moldable job model augmented with a generic set of internal parallelism variations. We then build a hierarchical scheduling framework based on the new model and conduct a set of simulations to evaluate the scalability, utilization and makespan of our algorithm. We also compare AC-DS with a simple policy based on equal resource sharing and another feedback-driven adaptive scheduler at the bottom level. Finally, we study the impact of different quantum patterns and reallocation costs on the performances of these schedulers.

4.1 A Malleable Parallel Job Model

Many parallel job models exist but very few of them generates malleable parallel jobs, which take the internal parallelism variations of the jobs into account. In this paper, we derive a novel malleable parallel job model for the empirical evaluation of adaptive scheduling algorithms.

Our model is based on the traditional moldable job models, which generate parallel jobs whose processor allocations cannot be changed over time once decided. Hence, these models only provide external information about the jobs, such as their work requirements, arrival patterns, average parallelism, etc. The key task of constructing malleable job model is, therefore, to represent the internal parallelism variations of parallel programs over time. To achieve that, we divide a parallel job generated by a moldable job model into a series of phases and each phase is captured by an internal structure randomly selected from one of the seven generic forms of distinct parallelism variation curves we have identified. These curves include Step, Log, Poly(II), Ramp, Poly(I), Exp and Impulse functions as shown in Fig. 2. The Step profile describes the stable parallelism requirement in a given period of time; the Impulse profile, on the other hand, represents the drastic variation of parallelism in instant time; the Ramp profile describes linear increasing and decreasing parallelism; the Exp, Log and the two kinds of Poly profiles describe sub-linear and super-linear changing parallelism, respectively. Moreover, these kinds of parallelism variation curves can reflect a wide range of real parallel program running patterns. For instance, the Impulse profile can emulate a drastic one-off increase in parallelism typically encountered in, e.g., a short parallel FOR loop, while the Step profile can represent a more stable data-parallel section of the job. The Ramp profile as well as the other profiles can model increases in the job's parallelism with different rates for spawning parallel threads. Fig. 3 demonstrates several ideal running parallelism patterns through real parallel program segments. In the figure, function F0() represents a thread with a large amount of computation invoked repeatedly by four different functions constructing the given parallelism profiles. As shown in the Fig. 3, function F1() consists of a fully parallelized FOR loop without interdependency profiling the Step curve; functions F2(), F3() and F4() recursively spawns themselves and other threads with different calling patterns, hence creating various rates of increasing parallelism.

These kinds of internal parallelism profiles provide a flexible way to construct malleable jobs whose parallelism changes with time. However, it is also a challenge to maintain consistency with the original moldable job model. When implementing our model, we follow a basic rule to maintain such consistency and ensure that all kinds of internal variation curves are coherent with each other. Specifically, we always maintain the same work, average parallelism and length for each phase regardless of the variation curve, as shown in Fig. 2. Moreover, we combine a pair of increasing and decreasing profiles together to create a basic parallelism variation block. To ease generation, we first construct the Step profile which is the simplest one to ensure that the work and the average parallelism adhere to those initially generated from the moldable job model. Then, other parallelism variation blocks are derived from the Step profile by varying the degree of the internal parallelism curves but

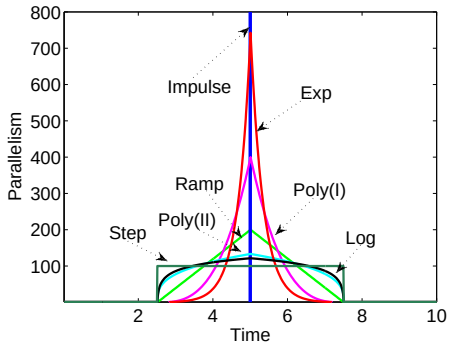


Figure 2: Seven parallelism variation curves described by Step, Log, Poly(II), Ramp, Poly(I), Exp and Impulse functions.

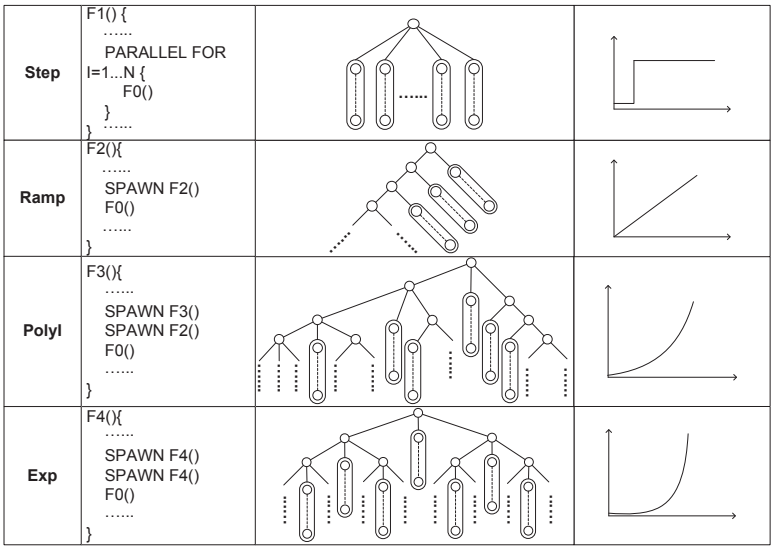


Figure 3: Sample parallel program segments and their corresponding parallelism variation over time.

with the same phase length, work requirement and average parallelism. Therefore, any combination of the variation curves for a job is ultimately consistent with the original moldable one.

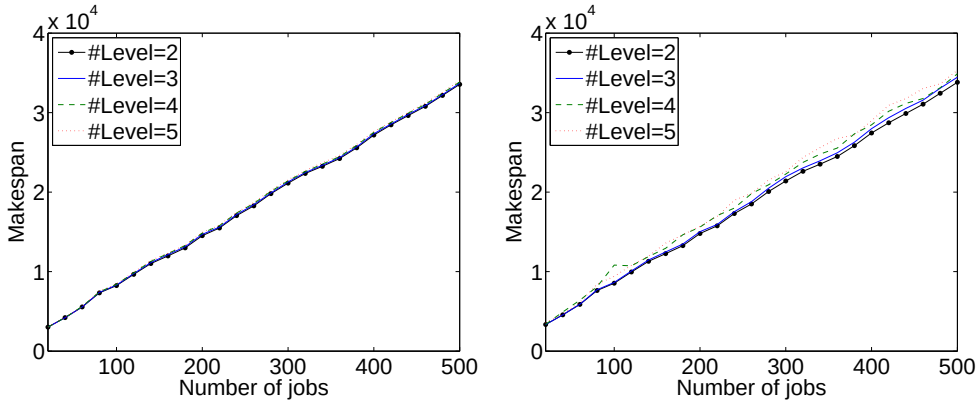
4.2 Simulation Setup

Based on the new malleable job model, we build a multi-level scheduling framework, which simulates the execution of parallel jobs on 256 processors. In this framework, we implement a request-allotment protocol to support the feedback mechanism and the processor reallocation among different levels. The number of levels K in the system hierarchy is increased from 2 to 5, and for a given K the tree structure that represents the system is randomly generated with the number of children of each intermediate node uniformly selected from $[1, 5]$. Each level has its independent scheduling quantum to aggregate processor requirements from its children and to readjust resource allocations. The quantum length at the bottom level, denoted by L, is set to be $1ms$. The malleable workloads are generated by following Downey's moldable job model [19], and the profile type of each internal phase is randomly selected with the phase length set to be $10 \cdot L$. In Downey's model, the job arrivals are modeled by a Poisson process, and the arrival rate is related to the system load. The load of the system is in turn proportional to the number of jobs, which in our experiments is increased from 20 to 500 with an increment of 20 each time. Each released job is randomly assigned to one of the leaf nodes in the system hierarchy. The parameters used in Downey's model are listed in Table 1.

Besides implementing our hierarchical algorithm AC-DS, we also implement two other natural scheduling algorithms and compare their performances. The first one, called EQUI-EQUI, is based on the simple equi-partitioning (EQUI) scheduler [21, 22] that divides the received processors at each node evenly among its immediate children that still contain unfinished jobs. The other scheduler is the feedback-driven adaptive scheduler AG-DS introduced in Section 3.5. To compare the performances of these scheduling algorithms, we

Table 1: Parameters used in Downey's model

Parameters	Value
Number of jobs in each workload (n)	$160 * \rho$
Offered load (ρ)	[0.5 3]
Average parallelism (A)	[1 256]
Job size parameter (β_1)	-0.14
Job size parameter (β_2)	0.073



(a) AC-DS (b) AG-DS

Figure 4: Scalability of AC-DS and AG-DS with respect to makespan when increasing the number of levels.

use processor utilization and makespan as the metrics. For any algorithm, its cost at a particular load is taken by carrying out the experiments 10 times and taking the average.

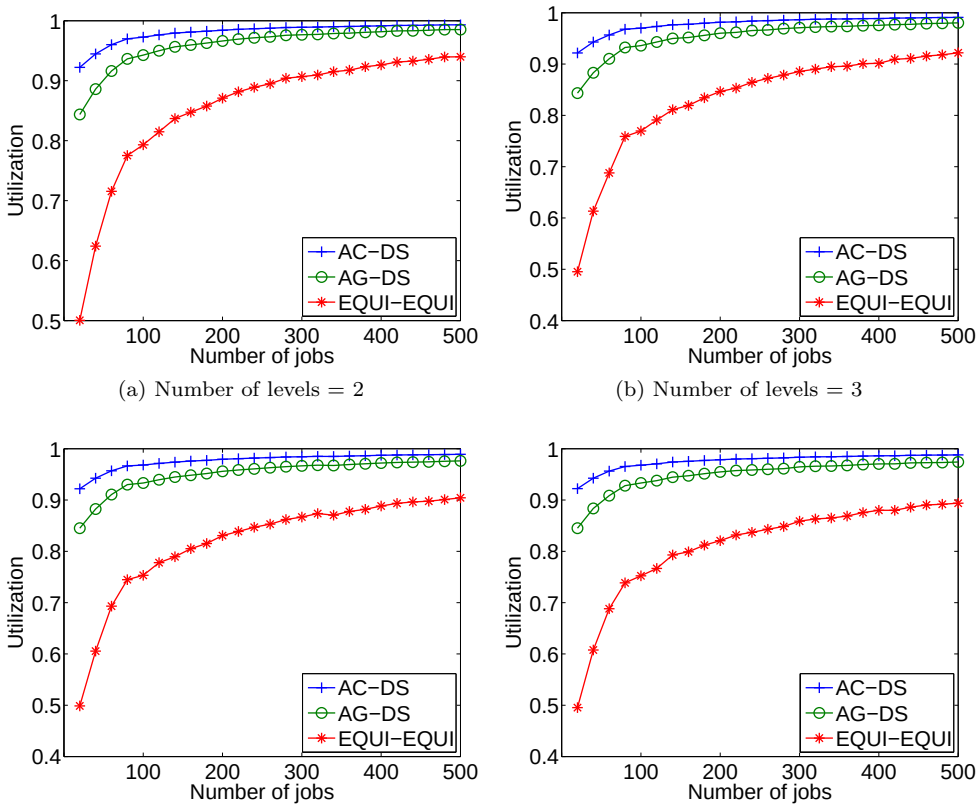
4.3 Simulation Results

(1) *Scalability of feedback-driven scheduling policies*

The scalability of a hierarchical scheduling algorithm measures its capability to respond to the increasing number of hierarchical levels, which to a large extent reflects the complexity of the system. From the simulation results, which is shown in Fig. 4, we can see that AC-DS achieves slightly better scalability than AG-DS. The makespan of AC-DS nearly converge to a single line when the number of levels increases from 2 to 5. On the other hand, the performance of AG-DS experiences some instability with increasing number of levels. In particular, AG-DS exhibits slightly degrading performance when the number of levels reaches 4, as shown in Fig. 4b. The reason is that the task scheduler AC used by AC-DS provides a more stable and efficient feedback scheme than the scheduler AG used by AG-DS when calculating the processor requests. This influences the aggregate resource feedbacks and hence the overall performance of the algorithms.

(2) *Utilization comparison of different scheduling policies*

Fig. 5 shows the utilizations of AC-DS, AG-DS, and EQUI-EQUI when the number of levels increases from 2 to 5. From the simulation results, we can see that the utilization of AC-DS is much better than that of the other two scheduling algorithms. The main reason is that AC-DS takes advantage of its stable scheduler in providing parallelism feedbacks and therefore has the ability to efficiently reallocate processors among the nodes and jobs. The simulation results also demonstrate that the utilizations of the two feedback-driven scheduling algorithms AC-DS and AG-DS are significantly better than that of EQUI-EQUI for a wide range of workloads. Specifically, both AC-DS and AG-DS achieve a higher and more stable utilization that reaches nearly 90%. On the other hand, the utilization of EQUI-EQUI is heavily influenced by the system load. The reason is that EQUI-EQUI is blind to the resource requirements at both job and node levels when allocating processors and thus it inevitably wastes a lot of processor cycles. Only when the system is heavily loaded, the utilization gap of the three algorithms becomes smaller because in this case there are not enough processor resources to be reallocated, although some nodes or jobs may still have high processor requirements. The simulation results demonstrate that the feedback-driven adaptive schedulers are more suitable to the situation where the system has light to medium loads. When the system load is heavy, however, the benefit of adaptive scheduling may be



(a) Number of levels = 2

(b) Number of levels = 3

(c) Number of levels = 4

(d) Number of levels = 5

Figure 5: Utilization comparisons of AC-DS and AG-DS with EQUI-EQUI.

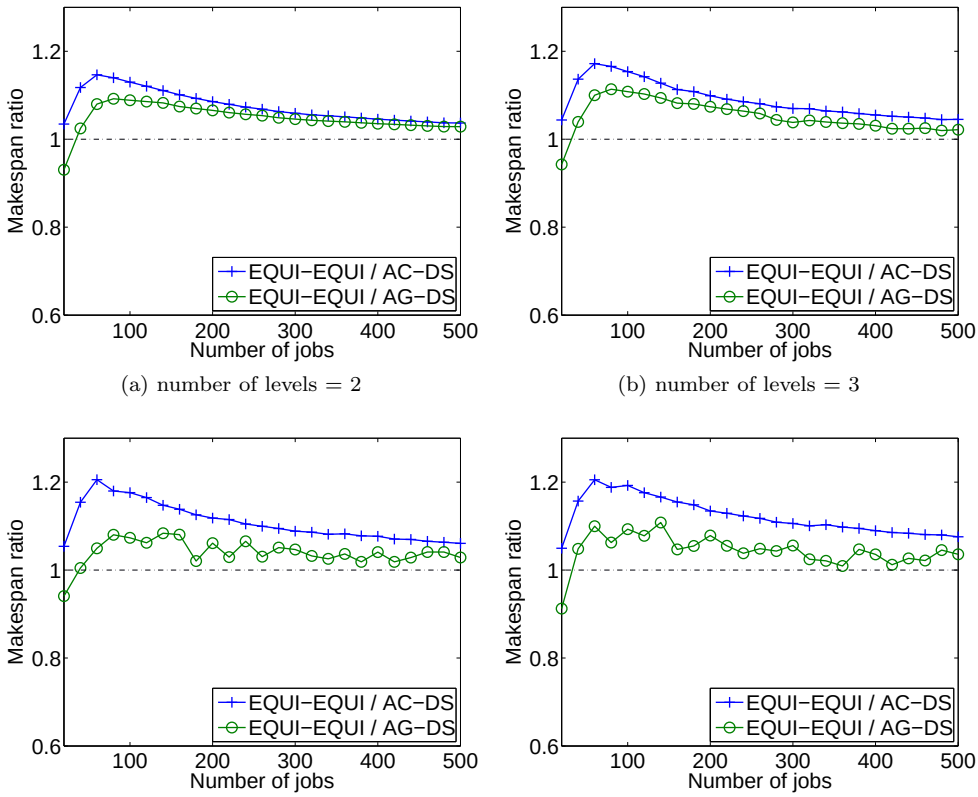
offset by the cost of reallocation overhead.

(3) *Makespan comparison of different scheduling policies*

Due to the advantage of proactive parallelism feedbacks, both AC-DS and AG-DS achieve better performance than EQUI-EQUI with respect to the makespan, as shown in Fig. 6. Note that in this set of figures, we present the makespan ratios by normalizing the makespans of the two feedback-driven schedulers with that of EQUI-EQUI for easier comparison. As we can see from the figure, AC-DS has better and more stable performance than the other two algorithms, especially with increasing number of hierarchical levels. For example, when the number of levels is 3, the makespan of AC-DS improves over AG-DS by only 4% on average while the improvement becomes 16% on average when the number of levels is 4. Moreover, compared with the two feedback-driven schedulers, only when the system has a small number of jobs, EQUI-EQUI shows its advantages with slightly better makespan. The reason is that under light load almost all jobs can be easily satisfied by EQUI-EQUI, which provides sufficiently good performance without the need of adaptive processor allocation. With increasing system load, however, the competition for resources becomes more intensive among the nodes and the jobs. Therefore, the performances of the feedback-driven algorithms become better than that of EQUI-EQUI, since they can dynamically adjust the processor allocations based on the jobs' execution history. When the system load continues to become much heavier, as shown in Fig. 6, the performances of AC-DS and AG-DS tend to converge to that of EQUI-EQUI, because in this case any job can only receive very few processors most of time, and thus frequent processor reallocations have no obvious benefits.

(4) *Impact of scheduling quantum and reallocation cost*

For the feedback-driven algorithms, scheduling quantum and processor reallocation cost are important system parameters, which may significantly affect the overall performance. In this section, we focus on evaluating the impact of these parameters on the performance of feedback-driven scheduling algorithms and compare them with that of EQUI-EQUI, whose reallocation cost is much lighter at runtime. In our simulation, we fix the number of levels to be 5 and the quantum length at a particular level is set to be $(QF)^{level-2} \cdot L$, where *level* denotes the level index that ranges from 2 to 5, L is the quantum length at level 2, and QF is a quantum factor that denotes different quantum patterns. For example, when QF is set to be 1, the quantum lengths of all levels are the same, namely L. When QF is set to be 2, the quantum lengths from the lowest level to the highest level are $L, 2L, 2^2L, 2^3L$ respectively. Furthermore, to evaluate the impact of reallocation cost, we



(a) number of levels = 2

(b) number of levels = 3

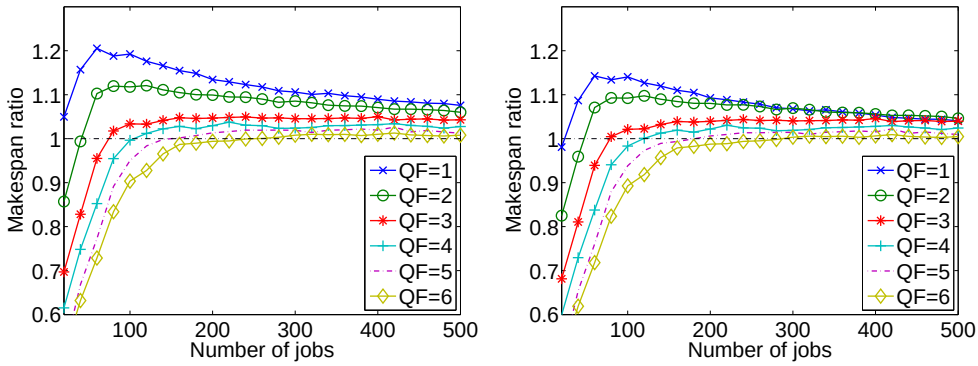
(c) number of levels = 4

(d) number of levels = 5

Figure 6: Makespan comparisons of AC-DS and AG-DS with EQUI-EQUI.

set the delay of reallocating a processor from one job to another to be proportional to the smallest quantum length L, i.e., $L \cdot CF$, where CF is a cost factor set to be $1/10, 1/5, 1/2$ respectively. Hence, the overall cost of reallocating x processors from a job is given by $x \cdot L \cdot CF$. Since the reallocation cost should be successively more expensive when climbing up the hierarchical tree, the corresponding delay at a high-level node is calculated by accumulating all its children's delay when its quantum expires. Note that we only focus on AC-DS in this section, as AG-DS has similar results.

We can see from Fig. 7 that different quantum patterns indeed have impacts on the performances of the scheduling algorithms. As shown in Fig. 7a, compared with EQUI-EQUI, the makespan of AC-DS tends to become larger when QF increases. For example, the makespan ratio of EQUI-EQUI over AC-DS is 1.13 on average when QF is set to be 1 while the ratio becomes only 1.02 on average when QF is set to be 6. This means that the feedback-driven scheduling algorithm has more benefits when QF is small since they can adjust processor allocations more effectively in this case. However, smaller QF also leads to larger reallocation cost and hence affect performance. As shown in Fig. 7b, when the reallocation cost CF is set to be $1/10$, the performance of AC-DS with $QF = 1$ clearly degrades compared to EQUI-EQUI, and the degradation is obviously more significant than the other values of QF. To clearly show the impact of reallocation cost on AC-DS, Fig. 8 gives the simulation results when the number of jobs is fixed to be 300 and the reallocation cost CF is varied from 0 to $1/2$. From these results, we can see that increases in reallocation cost has a larger impact for the relative performance of AC-DS when the quantum factor QF is small. For example, when CF is set to be $1/10$, the best makespan ratio of AC-DS is achieved at $QF = 2$, while AC-DS achieves the best makespan ratio with $QF = 4$ when CF is changed to $1/2$. In summary, these simulation results suggest that if the reallocation overhead in the system is small enough, having a uniform quantum length across different levels will give the feedback-driven scheduling algorithms more benefits. Otherwise, gradually increasing the length of the scheduling quantum when climbing up the system hierarchy seems to be a better option in order to achieve the optimal performance from the feedback-driven schedulers.



(a) EQUI-EQUI / AC-DS with CF = 0 (b) EQUI-EQUI / AC-DS with CF = 1/10

Figure 7: Impact of different quantum patterns on AC-DS with respect to makespan.

5 Related Work

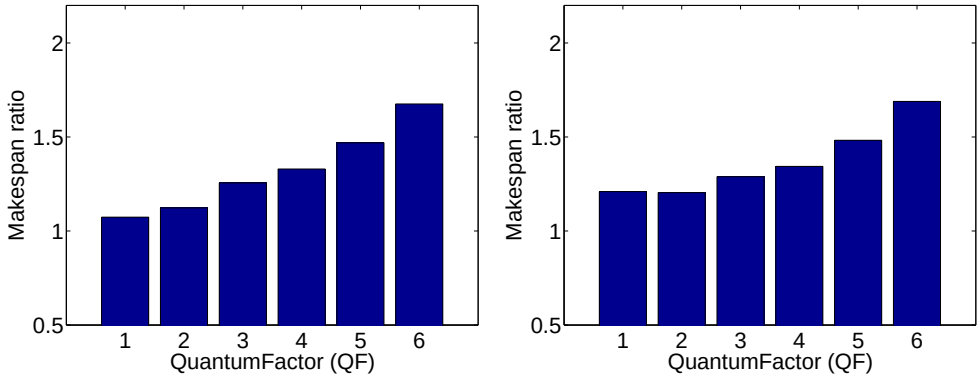
In this section, we review some related work on parallel workload modeling and non-clairvoyant adaptive scheduling.

Parallel Workload Modeling. According to the well-known classification by Feitelson and Rudolph [15], parallel jobs can be divided into three categories from the scheduling perspective, namely, rigid jobs, moldable jobs and malleable jobs. Rigid jobs are often scheduled by static schedulers as they cannot run on less or more processors than specified. Moldable jobs can run on a variable number of processors, but it cannot be modified once the job is started. Hence, the initial decision by the scheduler will determine the overall system performance. For malleable jobs, their processor allocation can be dynamically changed at runtime, and hence they provide the most flexibility for the schedulers to optimize performance. Many existing parallel job models [19, 23, 24, 25] exist, but they only consider rigid and moldable jobs, and to the best of our knowledge no previous work has explicitly modeled malleable jobs. This paper provides a malleable parallel job model by specifying a generic set of interval parallelism variation curves.

Adaptive Scheduling. To take advantages of malleable parallel jobs, adaptive scheduling has been extensively studied both theoretically and empirically in the literature. From theoretical perspective, Agrawal et al. [16, 10] studied adaptive scheduling using parallelism feedback. They proposed two adaptive schedulers, namely A-Greedy and A-Steal, based on a multiplicative-increase multiplicative-decrease strategy, and proved that both scheduling algorithms are efficient in terms of the running time and the processor waste for an individual job. In [8], He et al. combined A-Greedy and A-Steal with OS allocator DEQ [20] and proved that the resulting two-level adaptive schedulers AGDEQ and ASDEQ are O(1)-competitive in terms of the makespan. Under the same two-level adaptive scheduling framework, Sun et al. [9] proposed an improved adaptive scheduler ACDEQ, where the parallelism feedback is calculated by using an adaptive controller called A-CONTROL based on principles from the classical control theory. From algorithmic perspective, they proved that the two-level adaptive scheduler ACDEQ achieved a competitive ratio of O(1) with respect to the makespan.

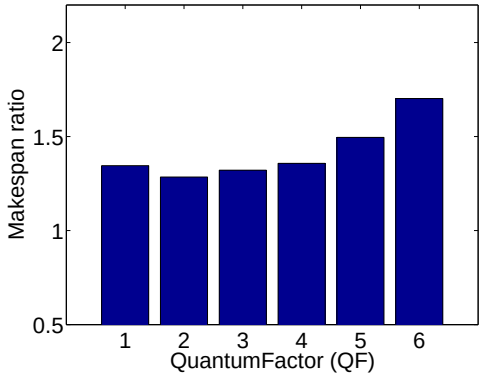
Many empirical studies on adaptive scheduling also exist in the literature. Agrawal et al. [6] presented experimental results on feedback-driven adaptive schedulers. They showed that the feedback-driven schedulers indeed have superior performance than the schedulers without parallelism feedback. He et al. [8] evaluated the performance AGDEQ under a wide range of workloads, and showed that it actually performs much better in practice than predicted by the theoretical bounds. Using simulations, Sun et al. [9] also confirmed that the ACDEQ scheduler with more stable parallelism feedback outperforms the other feedback-driven algorithms in terms of both individual job performance and makespan for a set of jobs. In addition, Weissman et al. [26], Corbalán et al. [27] and Sudarson et al. [28] have implemented various adaptive scheduling strategies on different platforms based on measurements of certain job characteristics such as speedup, efficiency, execution time, etc. All of them reported success in improving the system performances with adaptive scheduling.

Non-clairvoyant Scheduling. We now review some related work for the non-clairvoyant scheduling scenario. Non-clairvoyant scheduling was first introduced by Motwani et al. [29] in an attempt to design algorithms that are provably efficient for practical purposes. In multiprocessor environments, a well-known non-clairvoyant scheduling algorithm is EQUI (Equi-partitioning) [21, 22], which equally shares the available processors among all active jobs. For the makespan minimization problem, it was shown in [30] that EQUI achieves a competitive ratio of $O(\frac{\ln n}{\ln \ln n})$ when jobs are organized in two levels, where n is the total number of jobs in the system, and that no better ratio is possible. Two closely related work to ours in a similar setting are by Robert et al.

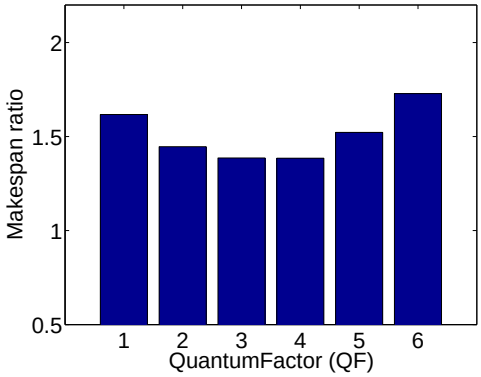


(a) EQUI-EQUI / AC-DS with CF = 0

(b) EQUI-EQUI / AC-DS with CF = 1/10



(c) EQUI-EQUI / AC-DS with CF = 1/5



(d) EQUI-EQUI / AC-DS with CF = 1/2

Figure 8: Impact of different reallocation costs on AC-DS with respect to makespan when number of jobs is fixed to be 300.

[30] and Sun et al. [31, 32]. In [30], the authors considered a three-level hierarchy by organizing the jobs in different job sets and present an online scheduling algorithm EQUI∘EQUI, which first splits evenly the available processors among the job sets, and then splits evenly the allocated processors among the jobs of each set. They considered the objective of set response time, i.e., the sum of makespan of all sets, and prove that EQUI∘EQUI achieves a competitive ratio of $(2+\sqrt{3}+o(1))\frac{\ln n}{\ln \ln n}$. The same performance metric was considered in [31], but the authors combined EQUI with the feedback-driven adaptive policies AGDEQ [8] and ACDEQ [9] to allocate the processor resources in a both fair and efficient manner under the non-clairvoyant setting. The proposed algorithm were shown to achieve $O(1)$-competitiveness. Finally, Sun et al. [32] generalized the result to an arbitrary number of hierarchical levels for the metric of set response time .

6 Conclusions

In this paper, we have focused on the problem of hierarchical scheduling for malleable parallel jobs on multilayered computing systems. We proposed a feedback-driven adaptive scheduling algorithms, called AC-DS, and showed that it achieves competitive and scalable performance in terms of makespan. A novel malleable job model is developed to verify the effectiveness of this algorithm. The results demonstrate that our algorithm has good scalability with increasing number of hierarchical levels and it outperforms two other natural schedulers for a wide range of workloads.

Acknowledgment

This work is partially supported by China National Hi-tech Research and Development Program (863 Project) under the grants No. 2011AA01A201, 2009AA01A131 and Natural Science Foundation of China under the grant No.61073011,61133004, 61173039.

References

[1] Surendran C, Purusothaman T, Balachandar R (2011) Performance analysis of a resource aggregator in a grid of grids environment. Computer Systems Science and Engineering, 26(4).

[2] Cao Y, Sun H, Qian D, Wu W (2012) Stable adaptive work-stealing for concurrent many-core runtime systems. IEICE Transactions on Information and Systems, 95:1407-1416.

[3] Cokuslu D, Hemeurlain A, Erciyes K (2012) Resource allocation for query processing in grid systems: a survey. Computer Systems Science and Engineering, 27(4).

[4] Chandra A, Shenoy P (2008) Hierarchical scheduling for symmetric multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 19:418-431.

[5] Abawajy J (2009) Adaptive hierarchical scheduling policy for enterprise grid computing systems. Journal of network and computer applications, 32:770-779.

[6] Agrawal K, He Y, Leiserson CE (2006) An empirical evaluation of work stealing with parallelism feedback. in: Proceedings of the International Conference on Distributed Computing Systems, Lisbon, Portugal, pp 19-29.

[7] He Y, Hsu W-J, Leiserson CE (2006) Provably efficient two-level adaptive scheduling. in: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, Saint-Malo, France, pp 1-32.

[8] He Y, Hsu W-J, Leiserson CE (2008) Provably efficient online non-clairvoyant adaptive scheduling. IEEE Transaction on Parallel and Distributed Systems, 19:1263-1279.

[9] Sun H, Cao Y, Hsu W-J (2011) Efficient adaptive scheduling of multiprocessors with stable parallelism feedback. IEEE Transactions on Parallel and Distributed Systems, 22:594-607.

[10] Agrawal K, Leiserson CE, He Y et al (2008) Adaptive work-stealing with parallelism feedback. ACM Transactions on Computer Systems 26:1-32.

[11] Sun H, Cao Y, Hsu W-J (2009) Competitive two-level adaptive scheduling using resource augmentation. in: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, Rome, Italy, pp 1-24.

[12] Dai Y, Levitin G, Trivedi K (2007) Performance and reliability of tree-structured grid services considering data dependence and failure correlation. IEEE Transactions on Computers, 56:925-936.

[13] Li C, Li L (2009) Three-layer control policy for grid resource management. Journal of Network and Computer Applications, 32:525-537.

[14] Han H, Kim S, Jung H et al (2009) A restful approach to the management of cloud infrastructure. in: Proceedings of the IEEE International Conference on Cloud Computing. Bangalore, India, pp 139-142.

[15] Feitelson D, Rudolph L (1998) Metrics and benchmarking for parallel job scheduling. in: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, Orlando, USA, pp 1-24.

[16] Agrawal K, He Y, Hsu W-J et al (2006) Adaptive scheduling with parallelism feedback. in: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York City, NY, USA, pp 100-109.

[17] Sun H, Hsu W-J (2008) Adaptive B-Greedy (ABG): a simple yet efficient scheduling algorithm. in: Proceedings of the International Parallel and Distributed Processing Symposium, Miami, FL, USA, pp 1-8.

[18] Borodin A, El-Yaniv R (1998) Online computation and competitive analysis. Cambridge University Press, New York, USA.

[19] Downey AB (1998) A parallel workload model and its implications for processor allocation. Cluster Computing, 1:133-145.

[20] McCann C, Vaswani R, Zahorjan J (1993) A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. ACM Transactions on Computer Systems, 11:146-178.

[21] Edmonds J (1999) Scheduling in the dark. in: Proceedings of the ACM Symposium on the Theory of Computing, Atlanta, GA, USA, pp 179-188.

[22] Edmonds J, Chinn DD, Brecht T et al (2003) Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. Journal of Scheduling, 6:231-250.

[23] Jann J, Pattnaik P, Franke H, Wang F, Skovira J and Riordan J (1997) Modeling of Workload in MPPs. in: Proceedings of Workshop on Job Scheduling Strategies for Parallel Processing, Geneva, Switzerland.

[24] Cirne W and Berman F (2001) A comprehensive model of the supercomputer workload In: Proceedings of the 4th Annual Workshop Workload Characterization, pp 140–148.

[25] Lublin U and Feitelson D (2003) The workload on parallel supercomputers: modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing, 63(11):1105–1122.

[26] Weissman JB, Abburi LR, England D (2003) Integrated scheduling: the best of both worlds. Journal of Parallel and Distributed Computing, 63:649-668.

[27] Corbalan J, Martorell X, Labarta J (2005) Performance-driven processor allocation. IEEE Transactions on Parallel and Distributed Systems, 16:599-611.

[28] Sudarsan R, Ribbens CJ (2007) ReSHAPE: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment, International Conference on Parallel Processing (ICPP), pp 44-44

[29] Motwani R, Phillips S, Torng E (1994) Nonclairvoyant scheduling. Theoretical Computer Science, 130:17-47.

[30] Robert J, Schabanel N (2007) Non-clairvoyant batch sets scheduling: Fairness is fair enough. in: Proceedings of the European Symposium on Algorithms, Eilat, Israel, pp 741-753.

[31] Sun H, Cao Y, Hsu W-J (2011) Fair and efficient online adaptive scheduling for multiple sets of parallel applications. in: Proceedings of the International Conference on Parallel and Distributed Systems, Tainan, Taiwan, pp 38-45.

[32] Sun H, Hsu W-J, Cao Y (2014) Competitive online adaptive scheduling for sets of parallel jobs with fairness and efficiency. Journal of Parallel and Distributed Computing, 74:2180-2192.