J Supercomput (2012) 60:311–337
DOI 10.1007/s11227-010-0461-8

Improving grid performance through processor allocation considering both speed heterogeneity and resource fragmentation

Po-Chi Shih · Kuo-Chan Huang · Yeh-Ching Chung

Published online: 11 August 2010

Abstract Grid performance are usually measured by the average turnaround time of all jobs in the system. A job's turnaround time consists of two parts: queue waiting time and actual execution time, which in a heterogeneous grid environment, are severely affected by the resource fragmentation and speed heterogeneity factors. Most existing processor allocation methods focus on one of these two factors only. This paper proposes processor allocation methods, which consider both resource fragmentation and speed heterogeneity, to improve system performance of heterogeneous grids. Extensive simulation studies have been conducted to show that the proposed methods can effectively deliver better performance under most resource and workload conditions.

Keywords Grid · Speed heterogeneity · Resource fragmentation · Processor allocation

1 Introduction

Grid performance can be measured by several different performance metrics. Among them, the most popular one may be average turnaround time of all jobs in the system. A job's turnaround time consists of two parts: queue waiting time and actual

P.-C. Shih (✉) · Y.-C. Chung
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, ROC
e-mail: shedoh@sslab.cs.nthu.edu.tw

Y.-C. Chung
e-mail: ychung@cs.nthu.edu.tw

K.-C. Huang
Department of Computer and Information Science, National Taichung University, Taichung, Taiwan, ROC
e-mail: kchuang@mail.ntcu.edu.tw

execution time. Shortening the time of these two parts thus becomes the main focus of different job management mechanisms. Job management in parallel and distributed systems usually entails two steps. Job scheduling decides the job sequence for processor allocation. Processor allocation is concerned with the assignment of the required number of processors for the selected job.

Traditionally, job scheduling methods are thought to be responsible for queue waiting time, while processor allocation methods for actual execution time. For example, Shortest-Job-First (SJF) has long been known to be the best scheduling method for producing the shortest average waiting time on single processor [32]. Backfilling based methods [26, 33] have been widely used in parallel system to manipulate the job execution order for reducing the waiting time of some jobs through improving system utilization. Processor allocation methods usually try to allocate a job to the fastest computing resource among those available in a distributed system for it to run as soon as possible.

However, things become complicated in a heterogeneous grid environment, where processor allocation not only influence job execution time but also affect queue waiting time. A heterogeneous grid usually integrates several parallel computers or clusters which may run at different speeds and be located in geographically distant sites. Communications between processors within the same site are usually achieved through high-speed networking devices, while messages passed across different sites have to go through a much slower wide-area network or Internet. A job allocated to a pool of processors within the same site can usually run faster than if it is assigned to processors across different sites. Therefore, the system tends to allocate a job within a single site to achieve high performance.

Because of speed heterogeneity, allocating a job onto different clusters may lead to different job execution times. Moreover, different processor allocation decisions for a certain job may also affect the queue waiting time of the jobs behind it because of the resource fragmentation issue. Resource fragmentation in grid environments, similar to the external fragmentation of memory space for dynamic memory management in operating systems [32], occurs when there is no single cluster being able to accommodate a parallel job although the number of total free processors of all clusters is larger than the job's requirement. Once resource fragmentation occurs, the jobs in the queue would be kept waiting for more time, inducing longer queue waiting time. Processor allocation decisions determine the probability of resource fragmentation. Therefore, processor allocation becomes a crucial issue in a heterogeneous grid environment because it affects both queue waiting time and job execution time. An effective processor allocation method for heterogeneous grid environments has to cope with both speed heterogeneity and resource fragmentation. However, most existing processor allocation methods focus on one of these two issues only.

For the resource fragmentation issue, earlier research results [19, 20] in homogeneous grids showed that the *best-fit* policy can deliver good performance. In this policy, upon allocation decision making a particular site in the grid is chosen on which a parallel job will leave the least number of free processors if it is allocated to that site. Regarding speed heterogeneity, the fastest-first policy [21] is a commonly used heuristic which always allocate a job to the fastest one among those sites able to accommodate it. This paper proposes processor allocation methods which intelligently switch between the best-fit and the fastest-first policies at runtime to achieve good system performance. Extensive experiments based on simulations under different workload conditions were conducted to evaluate the proposed methods.

It is believed that no single processor allocation method can always perform the best under all possible workload conditions. However, careful and extensive analysis of the performances of different methods under various workload conditions could lead to better understanding of the root causes of the performance difference between the methods. The understanding could in turn help develop more effective processor allocation methods.

The remainder of this paper is organized as follows. Section 2 discusses related works. Section 3 presents the grid model and assumptions of this paper. In Sect. 4, we analyze the potential strength of existing allocation methods and present the proposed *intelligent* allocation methods. Section 5 presents and discusses experiment results. Conclusion of this paper is given in Sect. 6.

2 Related works

Both job scheduling [22, 23] and processor allocation [27, 31] received a lot of research attention on earlier hypercube-based parallel computers. On a hypercube computer, allocating a job to different sub-cubes, although having little or no impact on that single job's performance, might lead to diverse overall system performance. This is because different allocation decisions lead to different distributions of left-over processors and, in turn, different probabilities of successful allocation of subsequent jobs. Later, when switch-based parallel computers and cluster-based computing systems being widely used, job scheduling become a more important issue than processor allocation. This stemmed from the fact that on such systems allocation can be made with any portion of the system and with any number of processors, in contrast to the power-of-two restriction on earlier hypercube computers. Therefore, the resource fragmentation problem was eliminated and processor allocation seemed straightforward. Many research efforts [16, 17, 24, 26, 33] have been spent on the job scheduling issue on such switch-based parallel computers or cluster-based computing systems. However, as *grid* [14, 18] becomes a promising computing platform, the resource fragmentation problem is coming back again and processor allocation needs to deal with it.

England and Weissman in [9] analyzed the costs and benefits of load sharing of parallel jobs in the computational grid. Experiments were performed for both homogeneous and heterogeneous grids. However, in their works simulations of a heterogeneous grid only captured the differences in capacities and workload characteristics. The computing speeds of nodes on different sites are assumed to be identical. In this paper, we deal with heterogeneous grids in which nodes on different sites may have different computing speeds.

As for job scheduling in grid environments, previous works discussed several strategies for a grid scheduler. One approach is the modification of traditional list scheduling strategies for usage on grid [10, 11, 13, 19, 30]. Some economic based methods are also being discussed [7, 8, 12, 35]. For processor allocation, there are several methods possible for selecting which site to allocate a job. In homogeneous grid environments, the major issue for processor allocation is resource fragmentation. Earlier simulation studies in our previous work [20] and in the literature [19] showed that the *best-fit* policy can deliver the best performance. In this paper, we explore possible processor allocation methods for a heterogeneous grid environment, where nodes on different sites may run at different speeds.

Some previous works investigated the co-allocation issues in grid environments [1–6, 10, 11, 13, 19, 20, 29, 34], where a parallel job can run across different sites to resolve the resource fragmentation problem. However, multisite parallel execution may incur high communication overhead because of the slower communication speed between different sites compared to the intra-site communication. Moreover, not all communication libraries support such kind of cross-site parallel execution. Even worse is that in heterogeneous grid environments cross-site parallel execution can lead to inefficient resource utilization because of speed heterogeneity. Therefore, in this paper, we focus on processor allocation methods which allocate an entire parallel job onto a single site in a grid.

3 Grid model

In order to evaluate the performance of processor allocation methods in a grid environment, a grid model is first presented to clarify the assumption of the underlying computing platform. The grid model consists of a resource model, a job model, and a performance model. In the resource model, there are several independent computing sites with their own local workload and management system. The grid integrates the sites and shares their incoming jobs. Each participating site itself is a homogeneous parallel computer system with the same computing speed while the grid is heterogeneous in the sense that nodes on different sites may differ in computing speed and different sites may have different numbers of nodes. The nodes within each site are linked with a fast interconnection network that does not favor any specific communication pattern [16]. The parallel computer system uses space-sharing and run the jobs in an exclusive fashion. Once a job starts execution, it runs to finish without interrupt. Neither live migration nor job preemption is allowed. Each job can only be executed in single site, that is, single job executed on multiple sites (it is sometimes referred to multisite execution or resource co-allocation) is not considered in this paper.

In the job model, each job is submitted to the system in an on-line manner, that is, without any knowledge of future job submission. The jobs under consideration are restricted to batch jobs because this job type is dominant on most parallel computer systems running scientific and engineering applications. Each job is assumed to be rigid and is associated with a requirement of a specific number of processors. We assume that the job execution time is not known in advanced. For the sake of simplicity, we assume a global queue to accommodate all the submitted jobs and a global grid scheduler to handles all job scheduling and resource allocation activities. Local schedulers are only responsible for starting the jobs after their allocation by the global scheduler. Theoretically a single central scheduler could be a critical limitation concerning efficiency and reliability. However, practical distributed implementations are possible, in which site-autonomy is still maintained but the resulting schedule would be the same as created by a central scheduler [14].

To evaluate the proposed allocation methods, we used the *average turnaround time* (*ART*) of all jobs as the performance metric in all simulations. The *ART* is defined by (1).

$$ART = \frac{\sum_{\forall \text{job}\, i} endTime_i - submitTime_i}{TotalNumberOfJobs} \tag{1}$$

To clarify the following presentation, we first define several terms which will be used hereafter in this paper.

- CS_j—the computing speed of site j.
- *AveCS*—the average computing speed of all sites.
- NP_j—the number of free processors in site j.
- NRP_i—the number of required processors of job i.
- *WQ*—the waiting queue which contains all jobs waiting for available resources in their arriving order.
- $Size_{WQ}$—the size of waiting queue, represented by the total number of jobs in the waiting queue.
- $S_{\text{bf}}(i)$— represent the site selected for computing job i by the *best-fit* method.
- $S_{\text{ff}}(i)$—represent the site selected for computing job i by the *fastest-first* method.
- *FJ*—the first job in *WQ*.

4 Intelligent allocation methods

In this section, we begin by analyzing the pros and cons of existing processor allocation methods in Sect. 4.1. Then several intelligent allocation methods are introduced from Sect. 4.2 to 4.4. The name "intelligent" represents the idea of taking advantage of different existing allocation algorithms by dynamically switching the allocation decision among them intelligently. We expect that the proposed intelligent methods can effectively deliver better performance under most workload and resource conditions.

4.1 Existing processor allocation methods

We first analyze two existing processor allocation methods named *best-fit* and *fastest-first*. The *best-fit* method [19, 20] allocates a job to the site which can yield the smallest resource fragmentation. This scheme works fine in a homogeneous grid. However, in a heterogeneous grid with computing speed differences among participating sites, the *best-fit* method may not perform well since it does not consider the speed heterogeneity [21]. In such an environment another processor allocation method called *fastest-first* has been proposed [21]. The *fastest-first* method focuses on speed heterogeneity in a heterogeneous grid and allocates a job to the fastest one among all the sites which can accommodate the job. Since *fastest-first* does not consider the difference between the amount of required processors and a site's free capacity, it may result in larger fragmentation than *best-fit*.

Besides, the relative performance of these two methods would largely depend on several factors such as computing speed heterogeneity, system loading, workload

Table 1 Relative strength analysis of different allocation methods

	System loading (High)	System loading (Low)
Speed heterogeneity (High)	undistinguishable	fastest-first
Speed heterogeneity (Low)	best-fit	undistinguishable

condition, and so on. In this paper, speed heterogeneity is measured by the variance of computing speeds of all participating sites in a grid. System loading can be simply observed and represented by the average length of the job waiting queue. Workload condition includes many attributes such as job arrival process, probability distribution of the numbers of required processors, execution time distribution, etc. Some of these parameters can be seen as random variables that dynamically change with time (e.g., system loading and workload condition). It is hard to have any allocation method that can surpass all other methods in all workload conditions. To this end, we focus on identifying the potential strength of each existing allocation method under different conditions, and then propose new allocation methods to combine all the advantages.

Table 1 shows the relative strength analysis of *best-fit* and *fastest-first* under different levels of speed heterogeneity and system loading. Since *best-fit* does not consider speed difference among participating sites, it is more suitable to be used when speed heterogeneity is low. Additionally, *best-fit* were shown to yield less resource fragmentation and lead to higher resource utilization than the *first-fit* method, which inspects the participating sites in a fixed order and allocates a job to the first site found to be able to accommodate the job [20]. Since *fastest-first* can be viewed as another form of *first-fit* if the sites in a grid are arranged in the descending order of computing speed, *best-fit* can be expected to outperform *fastest-first* in reducing resource fragmentation and raising resource utilization. When system loading is high, resource utilization rate is crucial to the overall system performance. Therefore, *best-fit* has higher potential to perform better than *fastest-first* when system loading is high. It is then clear that in the case of low speed heterogeneity and high system loading, *best-fit* is a better choice. On the contrary, when resource heterogeneity is high and system loading is low (one can image the extreme case when the waiting queue is always empty), computing speed of each job has higher influence on the overall system performance than the resource fragmentation effect. Therefore, *fastest-first* can potentially perform better than *best-fit* in this case.

To utilize the result in Table 1, we further analyze the situation when allocation operation takes place. In general, an allocation event is triggered when a new job is submitted to the system or when a running job finishes its execution. For each allocation event the system tries to continuously allocate as many jobs as possible. It stops allocation only when there are no sites being able to accommodate the first job in the waiting queue or when the waiting queue becomes empty. Note that not every triggered allocation event leads to actual allocation since there might be no enough resources or no jobs to allocate. Table 2 classifies all possible allocation events into four types of situations according to the status of waiting queue and the causes that trigger the events. The symbol “X” represents that there will be no actual allocation in that situation. Since we apply FCFS as the scheduling policy, $Size_{WQ} > 0$ implies

Table 2 Classification of allocation events

	Submit	Finish
$Size_{WQ} = 0$	(a)	X
$Size_{WQ} > 0$	X	(b)

that there is no site being able to accommodate the first job in waiting queue and that the newly submitted job must wait in the rear of waiting queue. For the case that $Size_{WQ} = 0$ and the triggering event is job finish, there are no jobs to allocate and therefore no actual allocation happens. Only situations (a) and (b) in Table 2 would lead to actual allocation if there is any site which can accommodate the submitted job or the first job in waiting queue.

We use a parentheses pair to represent speed heterogeneity and system loading. For example, (low, high) represents the situation that heterogeneity is low and loading is high. In situation (a), $Size_{WQ} = 0$ implies the system loading is low so it comprises the (low, low) or (high, low) cases mentioned in Table 1. In situation (b), it comprises all the cases in Table 1 since $Size_{WQ} > 0$ does not imply the system loading is high. Therefore, we need some further strategies to distinguish different cases in each situation to make better allocation decisions. These strategies will be discussed in the following sections.

4.2 Threshold-based approaches

The main idea of the threshold-based approaches is to dynamically switch the allocation decision between *best-fit* and *fastest-first* based on the analysis in Sect 4.1. The first allocation method we propose is named *static intelligent* (*SI*). In *SI*, we use the waiting queue length to distinguish between the situation (a) and situation (b) based on Table 2. In situation (a), we simply apply *fastest-first* and in situation (b) *best-fit* is employed. The decision of *SI* is shown in (2).

$$Decision\ SI = \begin{cases} best\text{-}fit, & \text{if } Size_{WQ} > 0 \\ fastest\text{-}first, & \text{if } Size_{WQ} = 0 \end{cases} \tag{2}$$

Since *static intelligent* method does not consider the speed heterogeneity, another allocation method named *threshold intelligent* (*TI*) is then proposed which further exploits the analysis result in Table 1. To do so, two problems must be considered. The first is how to determine the speed heterogeneity and the system loading while the second is how to set the threshold value appropriately. For speed heterogeneity (*SH*), we measure it by calculating the variance of the computing speeds of all the participating sites by (3).

$$SH = \frac{\sum_{\forall\, site\ j} (CS_j - AveCS)^2}{TotalNumberOfSites} \tag{3}$$

For system loading (*SL*), we calculate the ratio of the total number of required processors of all jobs in waiting queue to the total number of free processors in all sites by (4). Note that the denominator in (4) might be zero when processors in all sites are all occupied by some jobs. In this case, we force the denominator to be one to prevent the divide-by-zero problem.

$$SL = \frac{\sum_{\forall \text{job } i \text{ in } WQ} NRP_i}{\sum_{\forall \text{ site } j} NP_j} \tag{4}$$

Based on above *SH* and *SL* calculations, the *TI* method works as follows. First, it uses the same way as the *static intelligent* method to distinguish between situations (a) and (b). In (a), it applies *speed heterogeneity threshold* (*SHT*) to choose *fastest-first* or *best-fit* based on (5). In (b), a *system loading threshold* (*SLT*) is employed for choosing *fastest-first* or *best-fit* based on (6).

$$\textit{Decision TI} \text{ in situation (a)} = \begin{cases} \textit{best-fit}, & \text{if } SH \leq SHT \\ \textit{fastest-first}, & \text{if } SH > SHT, \end{cases} \tag{5}$$

$$\textit{Decision TI} \text{ in situation (b)} = \begin{cases} \textit{best-fit}, & \text{if } \; SL > SLT \\ \textit{fastest-first}, & \text{if } SL \leq SLT \end{cases} \tag{6}$$

For the values of *SHT* and *SLT*, we use simulations to decide appropriate settings which will be discussed in Sect. 5.2.

4.3 Queue-based approach

The main idea of queue-based approach is to further exploit the information provided by job waiting queue to guide the allocation decision. In this paper though we assume neither job execution time nor job submission time are known in advance, the jobs that have already been submitted and waited in job queue can be seen as an "off-line" behavior. Thus, the job execution order and the required number of processors of each job in waiting queue provide new information and new opportunities to develop more efficient intelligent allocation methods.

In this section, we will present the queue-based intelligent allocation method named *adaptive intelligent 2* (*AI2*), which is inspired by the *adaptive* allocation strategy presented in our previous work [21] (we will call it *adaptive intelligent* (*AI*) hereafter in this paper). In the following, we first introduce the *AI* before going through *AI2*.

In both *AI* and *AI2*, we first distinguish between situation (a) and situation (b) using the same way as in the *static intelligent* method. In situation (a), since there is no job in waiting queue, both methods directly compare the computing speeds of the sites selected by different allocation methods to make the allocation decision. The allocation decision is determined by (7), where *FJ* stands for the first job in the waiting queue.

$$\textit{Decision AI \& AI2} \text{ in situation (a)} = \begin{cases} \textit{best-fit}, & \text{if } CS_{S_{\text{bf}}(FJ)} \geq CS_{S_{\text{ff}}(FJ)} \\ \textit{fastest-first}, & \text{if } \; CS_{S_{\text{bf}}(FJ)} < CS_{S_{\text{ff}}(FJ)} \end{cases} \tag{7}$$

In situation (b), these two methods act differently. In *AI*, with the information provided by waiting queue, the decision is made based on a calculation of which method can further allocate more jobs in waiting queue for immediate execution. The number of allocation $NoA(i)$ of job i is calculated by (8)

$$NoA(i) = \begin{cases} 1, & \text{if job } i \text{ can be allocated to at least one site} \\ & \text{and } NoA(i-1) = 1. \\ 0, & \text{if job } i \text{ can not be allocated to any site.} \end{cases} \tag{8}$$

$NoA_{\mathrm{bf}}(i)$ and $NoA_{\mathrm{ff}}(i)$ are used to denote the number of allocation using *best-fit* and *fastest-first,* respectively. Thus the total number of successful allocations by *best-fit* and *fastest-first* are denoted by $TNoA_{\mathrm{bf}}$ and $TNoA_{\mathrm{ff}}$ and defined in (9) and (10), respectively.

$$TNoA_{\mathrm{ff}} = NoA_{\mathrm{ff}}(FJ) + \sum_{\forall i \text{ in } WQ \text{ exclude } FJ} NoA_{\mathrm{bf}}(i) \tag{9}$$

$$TNoA_{\mathrm{bf}} = \sum_{\forall i \text{ in } WQ} NoA_{\mathrm{bf}}(i) \tag{10}$$

A value *AIScore* which represents the relative performance of applying *best-fit* or *fastest-first* for the first job in waiting queue is then calculated by (11).

$$AIScore = CS_{S_{\mathrm{ff}}(FJ)} - CS_{S_{\mathrm{bf}}(FJ)} - (TNoA_{\mathrm{bf}} - TNoA_{\mathrm{ff}}) \times CS_{S_{\mathrm{bf}}(FJ)} \tag{11}$$

The allocation decision of *AI* for the first job in waiting queue in situation (b) is then determined by (12).

$$Decision\ AI \text{ in situation (b)} = \begin{cases} \textit{best-fit}, & \text{if } AIScore < 0 \\ \textit{fastest-first}, & \text{if } AIScore \geq 0 \end{cases} \tag{12}$$

The pseudo code of the *AI* allocation method is shown in Fig. 1. Note that though in (9) and (10) we attempt to allocate as many jobs as possible, however, these steps are just a simulation to calculate the *AIScore* and will not actually perform allocation for those jobs. In the end, we only make the allocation decision for the first job in waiting queue.

Here, we start to introduce the proposed *AI2* allocation method. The major difference in *AI2* is to take the speed heterogeneous into account. In *AI2*, we make the allocation decision in situation (b) by calculating which allocation method can allow subsequent jobs in waiting queue to consume more computing capacity. The computing capacity $CC(i)$ consumed by job i is calculated by (13).

$$CC(i) = \begin{cases} CS_j \times NRP_i, & \text{if job } i \text{ is allocated to site } j \text{ and } CC(i-1) > 0. \\ 0, & \text{if job } i \text{ can not be allocated to any site.} \end{cases} \tag{13}$$

We use $CC_{\mathrm{bf}}(i)$ and $CC_{\mathrm{ff}}(i)$ to denote the computing capacity consumed by job i when using *best-fit* and *fastest-first,* respectively. The total computing capacities consumed using *best-fit* and *fastest-first* are denoted by TCC_{bf} and TCC_{ff} and defined in (14) and (15), respectively.

$$TCC_{\mathrm{ff}} = CC_{\mathrm{ff}}(FJ) + \sum_{\forall i \text{ in } WQ \text{ exclude } FJ} CC_{\mathrm{bf}}(i), \tag{14}$$

Fig. 1 Pseudo code of the *adaptive intelligent* (*AI*) allocation method

```
Method AdaptiveIntelligent()
{
  calculate $S_{bf}(FJ)$ and $S_{ff}(FJ)$
  **if** $S_{bf}(FJ) = S_{ff}(FJ)$
    choose *the site suggested by both methods*
  **end if**
  **if** $Size_{WQ} = 0$ **and** event = Submit
    **if** $CS_{S_{bf}(FJ)} \geq CS_{S_{ff}(FJ)}$
      choose *best-fit*
    **otherwise**
      choose *fastest-first*
    **end if**
  **end if**
  calculate *AIScore*
  **if** *AIScore* >= 0
    choose *fastest-first*
  **otherwise**
    choose *best-fit*
  **end if**
}
```

$$TCC_{bf} = \sum_{\forall i \text{ in } WQ} CC_{bf}(i) \tag{15}$$

A value *AI2Score* which represents the relative performance of applying *best-fit* or *fastest-first* for the first job in waiting queue is then calculated by (16).

$$AI2Score = \frac{CS_{S_{ff}(FJ)}}{CS_{S_{bf}(FJ)}} \times \frac{TCC_{ff}}{TCC_{bf}} \tag{16}$$

The allocation decision for *AI2* in situation (b) is then determined by (17).

$$Decision\,AI2 \text{ in situation (b)} = \begin{cases} \textit{best-fit}, & \text{if } AI2Score \leq 1 \\ \textit{fastest-first}, & \text{if } AI2Score > 1 \end{cases} \tag{17}$$

The pseudo code of *AI2* is almost the same as *AI* (see Fig. 1), only replacing the *AIScore* with *AI2Score*.

4.4 Mixed approaches

The main idea of the mixed approaches is to combine the advantages of the threshold-based approaches and the queue-based approaches. Here the mixed intelligent allocation method named *threshold-based adaptive intelligent* (*TAI*) and *threshold-based adaptive intelligent 2* (*TAI2*) are proposed. In both methods, we extend the analysis result in Table 1 by replace the "undistinguishable" cases with *AI* or *AI2,* respectively. The idea of the *TAI* and *TAI2* is show in Table 3. Note that the *TAI* and *TAI2* are only

Table 3 The idea of the proposed *TAI* and *TAI2* allocation methods

TAI (*TAI2*)		System Loading (*SL*)	
		Low	High
Speed Heterogeneity (*SH*)	Low	*AI* (*AI2*)	*BF*
	High	*FF*	*AI* (*AI2*)

Table 4 Time complexity of each allocation method

Allocation method	Time complexity	Allocation method	Time complexity
BF	$O(m)$	*AI*	$O(mq)$
FF	$O(m)$	*AI2*	$O(mq)$
SI	$O(m)$	*TAI*	$O(m+q+mq)$
TI	$O(m+q)$	*TAI2*	$O(m+q+mq)$

different in the (low, low) and (high, high) cases where in *TAI* the *AI* method is invoked and in *TAI2* the *AI2* method is invoked, respectively. For both *TAI* and *TAI2*, we use the same way as in the *threshold intelligent* method to determine the speed heterogeneity and system loading. *SHT* and *SLT* are applied to distinguish the level of speed heterogeneity and system loading respectively.

For the values of *SHT* and *SLT* in both *TAI* and *TAI2*, we also apply simulations to decide appropriate settings which will be discussed in Sect. 5.2.

4.5 Time complexity analysis of all intelligent methods

Here, the time complexity is calculated for a single allocation event. Suppose that the grid environment has m sites and the average length of waiting queue is q. In *BF* and *FF*, the worse case is to traverse all m sites to make the allocation decision. So, the time complexity of these two methods is $O(m)$. In *SI*, based on $Size_{WQ}$ either *BF* or *FF* will be invoked for allocation accordingly. Thus, the time complexity of *SI* is a linear combination of the time complexity of *BF* and *FF*. Since both *BF* and *FF* have the same time complexity $O(m)$, the time complexity of *SI* is $O(m)$ as well. In *TI*, two threshold related variables *SH* and *SL* are required for calculation. Since *SH* is only required to be calculated once in grid start-up, its time complexity can be ignored. For *SL*, it needs to traverse all the jobs in waiting queue and all the sites, thus it takes $m+q$ step for the calculation. Then at most, m steps are required by either *BF* or *FF*. Therefore, the time complexity of *TI* is $O(2m+q)=O(m+q)$. In *AI* and *AI2*, it is required to simulate the allocation for all the jobs in waiting queue. Thus, the time complexity of both methods is $O(mq)$. Both *TAI* and *TAI2* require the calculation of *SL,* and then either *AI* or *AI2* is invoked for final decision. Therefore, the time complexity of both methods is $O(m+q+mq)$. Table 4 summarizes the time complexity of each allocation method mentioned above.

Table 5 Characteristics of the workload log on SDSC's SP2

Queue number	Number of jobs	Maximum execution time (sec)	Average execution time (sec)	Maximum NRP_i	Average NRP_i
Queue 1	5038	21922	393.99	8	3.27
Queue 2	8838	510209	7029.57	128	17.00
Queue 3	27075	172832	7357.81	128	12.56
Queue 4	12859	452520	10620.82	128	12.17
Queue 5	231	64828	1051.76	50	4.01
Total	54041				

5 Experiments and discussion

In the following experiments, both publicly downloadable workload traces and synthetic workload models [28] are used for performance evaluation of the proposed methods. The detail setting of the workload log and synthetic workload models are covered in Sect. 5.1. Section 5.2 presents the effects of various threshold settings on the performance of *TI*, *TAI*, and *TAI2*. Then the performance comparisons of the six proposed intelligent allocation methods are covered in Sect. 5.3. In Sect. 5.4, we study the potential of dynamic threshold setting as a further research direction.

5.1 Experimental settings

For real workload traces, we used the SDSC's SP2 workload log on [28] as the basic input workload in the following simulations. The workload log on SDSC's SP2 contains 73496 records collected on a 128-node IBM SP2 machine at San Diego Supercomputer Center (SDSC) from May 1998 to April 2000. After excluding some problematic records based on the completed field in the log, the simulations in this paper use 54041 job records as the input workload. The detailed workload characteristics are shown in Table 5.

In simulations, we use the following three attributes of each job record gotten from the SDSC SP2's workload log.

- **Number of processors.** It is the number of processors a job uses according to the data recorded in the workload log.
- **Submission time.** This provides the information about when a job is submitted to its home site.
- **Runtime.** It indicates the required execution time for a job using the specified number of processors on its home site. This information of runtime is required for driving the simulation to proceed. However, in our job scheduling methods, the job scheduler does not know the job runtime prior to a job's finish of execution. Therefore, they do not use this information to guide the determination process of job scheduling and allocation.

In the SDSC's SP2 system, the jobs in this log are put into five different queues and all these queues share the same 128 processors on the system. In the following simulations, this workload log will be used to model the workload on a computational grid

Table 6 Configuration of the computational grid

	total	site 1	site 2	site 3	site 4	site 5
Number of processors	442	8	128	128	128	50

consisting of five different sites whose workloads correspond to the jobs submitted to the five queues respectively. Table 6 shows the configuration of the computational grid under study. The number of processors on each site is determined according to the maximum number of required processors of the jobs belonged to the corresponding queue for that site.

To simulate the speed difference among participating sites, we define a speed vector, $speed = (sp_1, sp_2, sp_3, sp_4, sp_5)$, to describe the relative computing speeds of all the five sites in the grid, in which the value 1 represents the computing speed resulting in the job execution time in the original workload log. We also define a load vector, $load = (ld_1, ld_2, ld_3, ld_4, ld_5)$, which is used to derive different loading levels from the original workload data by multiplying the load value ld_j to the execution times of all jobs at site j.

In order to evaluate the performance of the proposed methods on various workload conditions, we conducted a series of experiments by varying three adjustable parameters listed in Table 7. *Speed heterogeneity* (*SH*), represented by the variance of computing speeds of all sites, ranges from 0 to 0.24 with a step of 0.06. For better understanding of the influence of *SH*, setting $SH = 0.06$, 0.12, 0.18, and 0.24 will make the speed of the fastest site 1.9, 2.7, 4.3, and 5.6 times faster than the slowest site, respectively. $SH = 0$ reduces to the homogeneous case. We use a normal distribution generator with *mean* set to 1 and *variance* equal to the *SH* setting. To generate the computing speed for each site, sp_1, sp_2, and sp_3 are generated by the generator while sp_4 and sp_5 are calculated to satisfy (18) and (19).

$$\sum_{j=1}^{5} sp_j \times NP_j = C, \tag{18}$$

$$\frac{\sum_{j=1}^{5}(sp_j - 1)^2}{5} = SH \tag{19}$$

In (18), C is a constant to represent the total computing capacity of the entire grid. Here, C is set to 442, the total number of processors in all sites, to represent the computing capacity when all sites have the same computing speed index of one.

System loading (*SL*), ranging from 1 to 4 with a step of 0.5, is simulated by setting the load of each site to the *SL* value (e.g., the load vector is set to (2, 2, 2, 2, 2) when $SL = 2$). The following uses the average length of waiting queue in a homogeneous case ($SH = 0$) with the *best-fit* method as an example to show the effect of *SL*. The length of waiting queue will be 0.9, 3.4, 7.8, 19.8, 98, 1126, and 2618 when *SL* is set to 1, 1.5, 2, 2.5, 3, 3.5, and 4, respectively.

Table 7 Parameters for experiments with workload log

Speed heterogeneity (*SH*)	$\{0, 0.06, 0.12, 0.18, 0.24\}$
System loading (*SL*)	$\{1, 1.5, 2, 2.5, 3, 3.5, 4\}$
Resource configuration (*RC*)	$\{100\%, 75\%, 50\%, 25\%\}$

Table 8 Characteristic of SDSC's SP2 workload with respect to various *RC* settings

RC setting	Number of jobs	Maximum NRP_i	Average NRP_i
$RC = 100\%$	54041	128	12.29
$RC = 75\%$	54305	96	12.23
$RC = 50\%$	54534	64	12.18
$RC = 25\%$	58890	32	11.28

Resource configuration (*RC*), defined in (20), ranges from 100% to 25% with a step of 25% in the simulations.

$$RC = \frac{\mathrm{Max}(NRP_i),\ \forall \text{job}\, i}{\mathrm{Max}(NP_j), \forall \text{site}\, j} \tag{20}$$

With $RC = 100\%$, we use the maximum number of requested processors of all jobs in each queue as the size of that site, which are 8, 128, 128, 128, 50 corresponding to queue 1 to queue 5, respectively. This resource setting was used for all simulations. For other *RC* settings, we simulated it by cutting a job that exceeds the specified percentage into several small jobs. For example, when $RC = 25\%$, a job requesting 100 processors was cut into four small jobs, where three of them each requested 32 processors ($128 \times 25\%$) and the last one asked for the remaining 4 processors. Table 8 shows the characteristics of SDSC's SP2 workload with respect to different *RC* settings.

Note that only *SL* will change the amount of workload brought into the system while the other two parameters neither change the total computing capability of all resources nor change the average workload brought into the system.

For synthetic workload models, totally 10 models (including their source code) are available on the parallel workload archive [28]. Since in this paper we only consider rigid job type, two models named Feitelson96 [15] and Lublin99 [25] are chosen according to the *jobs* field. In Feitelson96, 30000 jobs were generated with the maximum number of processors per job to be 128 and the maximum job execution time to be 64800 seconds. In Lublin99, 30000 jobs were generated as well with the maximum number of processors per job to be 128 and the maximum job execution time to be 162754 seconds. The detail characteristics of these two models can be found in [15, 25].

In both models, two adjustable parameters are used to simulate various workload conditions as shown in Table 9. The definition of *SH* is the same as in the workload log case. However, for *SL*, we use the mean of inter-arrival-time to represent the *system loading*. Parameters ARR_FACTOR and AARR are used to adjust the mean of inter-arrival-time in the source code of Feitelson96 and Lublin99 respectively. The following uses the average length of waiting queue for the homogeneous case ($SH = 0$) with the *best-fit* method as an example to show the effect of *SL*. In Feitelson96, the length of waiting queue will be 0.9, 2.7, 13.5, 43.3, and 523.6 when *SL* is set to 600, 500, 400, 350, and 300, respectively. In Lublin99, the length of waiting queue will be 0.9, 3.2, 9.3, 24.2, and 377.3 when *SL* is set to 10.5, 10, 9.5, 9, and 8.5, respectively.

5.2 Performance of various threshold settings for *TI*, *TAI*, and *TAI2*

To find a good threshold setting, we apply simulations on the workload log with various threshold pairs (*SHT*, *SLT*). All the combinations of four *SHT* values (0.05, 0.1, 0.15, and 0.2) and five *SLT* values (1, 1.5, 2, 2.5, and 3) are simulated under all the combinations of 5 *SH* values and 7 *SL* values (see Table 7). Figures 2, 3, and 4 show the result of each threshold pair with all *SH* settings and $SL = 2.5$ for *TI*, *TAI*, and *TAI2*, respectively. It can be observed that smaller *SHT* lead to better performance in average. Nevertheless, for *SLT*, no single threshold value can constantly surpass all the other settings.

Here, a new performance metric is introduced to evaluate the performance of each threshold settings, named *total normalized performance improvement ratio* (*TNPIR*). *TNPIR* is the summation of *normalized performance improvement ratio* (*NPIR*) for all parameter settings. Here, we have totally 35 sets of parameter settings (5 *SH* values

Table 9 Parameters for experiments with synthetic workload models

Speed heterogeneity (*SH*)	{0, 0.06, 0.12, 0.18, 0.24}
System loading (*SL*) in Feitelson96	{600, 500, 400, 350, 300}
System loading (*SL*) in Lublin99	{10.5, 10, 9.5, 9, 8.5}

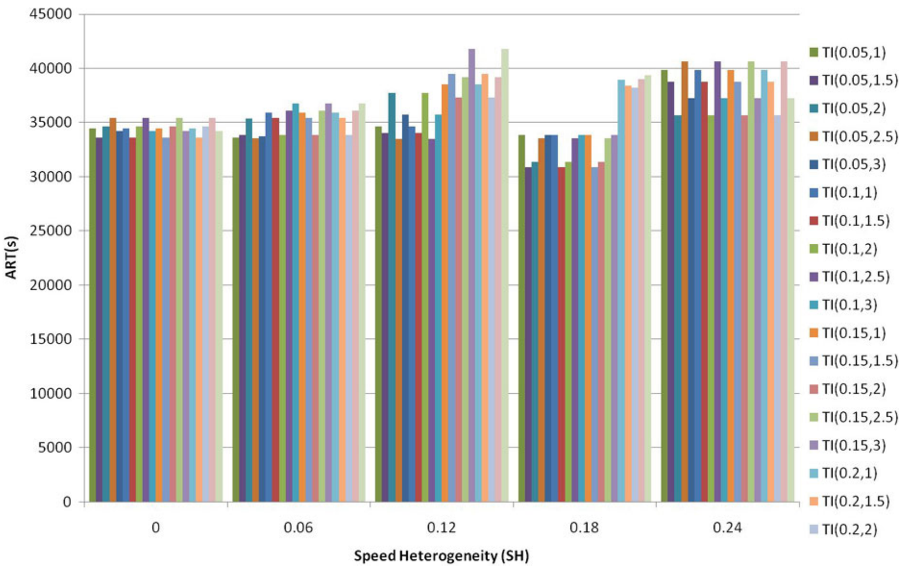

Fig. 2 *AverageResponseTime(ART)* of *TI* with respect to various *SHT* and *SLT* settings

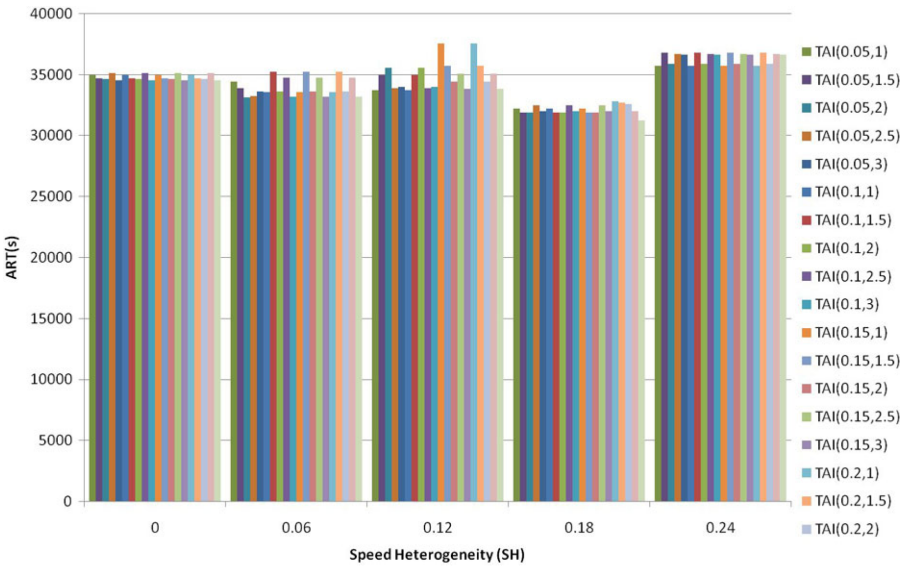

Fig. 3 *AverageResponseTime(ART)* of *TAI* with respect to various *SHT* and *SLT* settings

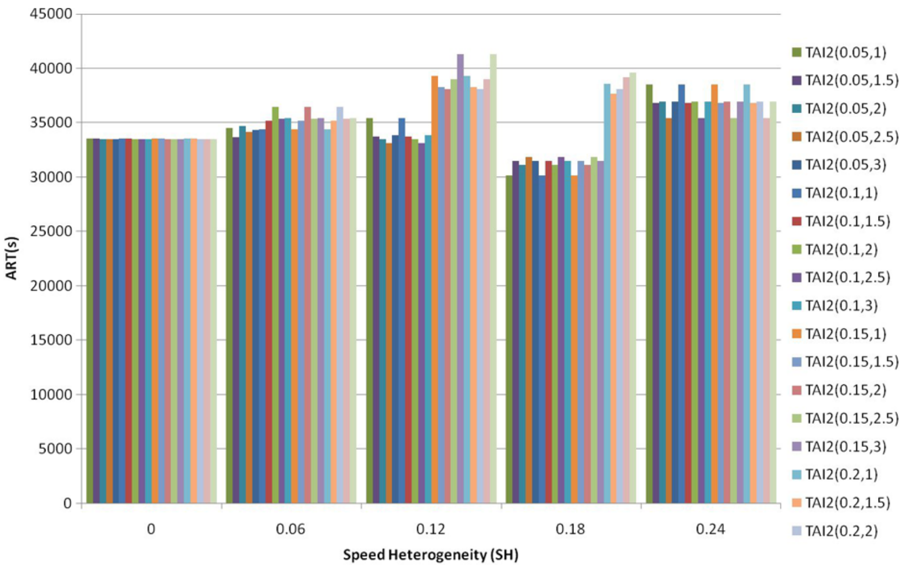

Fig. 4 *AverageResponseTime(ART)* of *TAI2* with respect to various *SHT* and *SLT* settings

and 7 *SL* values). *NPIR* is normalized with respect to the average *ART* (*AveART*) of all 20 sets of threshold settings defined in (21)

$$AveART = \frac{\sum_{\forall tp \in TP} ART_{tp}}{20} \tag{21}$$

Table 10 Total normalized performance improvement ratio (*TNPIR*) of each threshold pair for *TI*, *TAI*, and *TAI2*

Threshold pair (*SHT*, *SLT*)	(0.05, 1)	(0.05, 1.5)	(0.05, 2)	(0.05, 2.5)	(0.05, 3)	(0.1, 1)	(0.1, 1.5)	(0.1, 2)	(0.1, 2.5)	(0.1, 3)
TI	102%	85%	**110%**	92%	96%	98%	58%	80%	39%	66%
TAI	7%	**85%**	41%	27%	51%	−7%	24%	9%	−8%	39%
TAI2	**125%**	91%	82%	115%	120%	83%	56%	74%	70%	64%
Threshold pair (*SHT*, *SLT*)	(0.15, 1)	(0.15, 1.5)	(0.15, 2)	(0.15, 2.5)	(0.15, 3)	(0.2, 1)	(0.2, 1.5)	(0.2, 2)	(0.2, 2.5)	(0.2, 3)
TI	39%	−15%	34%	−57%	22%	−80%	−159%	−112%	−200%	−102%
TAI	−54%	16%	−1%	−35%	24%	−83%	−17%	−26%	−53%	−6%
TAI2	44%	−32%	−25%	−18%	1%	−106%	−147%	−185%	−156%	−160%

where ART_{tp} is the *ART* of the threshold pair *tp* and *TP* is a set containing all the threshold pairs. The *NPIR* of each threshold pair *tp* is defined in (22).

$$NPIR(tp) = \frac{AveART - ART_{tp}}{AveART} \tag{22}$$

Table 10 shows the *TNPIR* for *TI*, *TAI*, and *TAI2*, respectively. According to the best result of this table, the threshold pairs (0.05, 2), (0.05, 1.5), and (0.05, 1) are chosen for *TI*, *TAI*, and *TAI2,* respectively.

5.3 Experimental results and discussions

Here, the six intelligent methods, named *static intelligent* (*SI*), *threshold intelligent* (*TI*), *adaptive intelligent* (*AI*) [21], *adaptive intelligent 2* (*AI2*), *threshold-base adaptive intelligent* (*TAI*) and *threshold-based adaptive intelligent 2* (*TAI2*), are compared with the *best-fit* (*BF*) [20], and *fastest-first* (*FF*) [21] methods. Both the workload log and synthetic workload models are used for performance evaluation. To avoid bias of particular speed setting, 10-speed vectors for each *SH* value are randomly generated for simulation. All presented results in this section are the average value of 10 experiments with those10 speed vectors.

Figures 5, 6, 7, and 8 shows the performance result of each allocation method under the workload log with $RC = 100\%$, 75%, 50%, and 25%, respectively. Each subfigure shows the simulation result performed by varying the *SH* form 0 to 0.24 with respect to a specific *SL* setting. From all the figures, we can observe that in the (low, high) case (see Figs. 5–8 (e), (f), and (g) with $SH = 0$) *best-fit* surpasses *fastest-first.* This observation is consistent with our analysis in Table 1. The experimental results in Figs. 5–8 (a), (b), (c), and (d) also confirm another analysis in Table 1, which indicates that *fastest-first* outperforms *best-fit* for case (high, low).

Figures 9 and 10 shows the results for Feitelson96 and Lublin99 workload models, respectively. The result for workload models is generally consistent with our finding

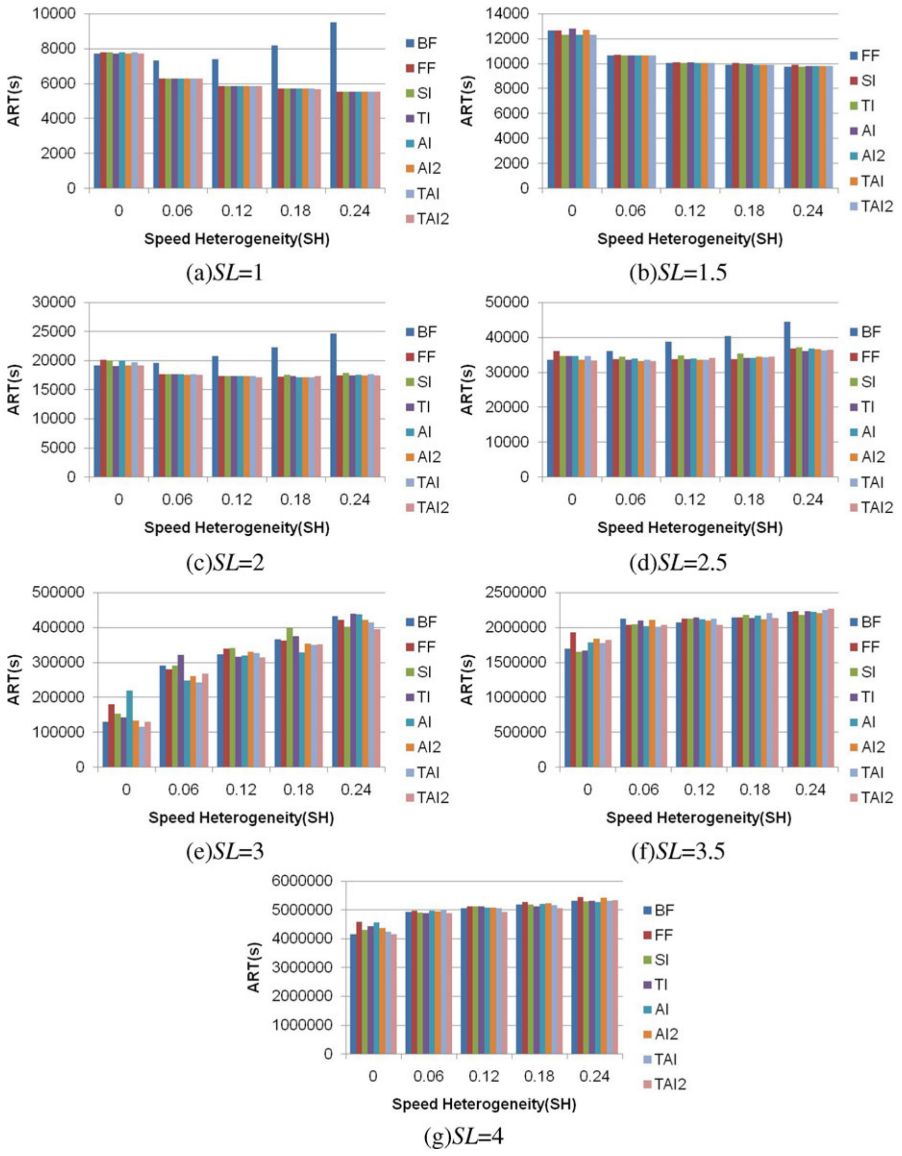

Fig. 5 Result of workload log for *RC* = 100% with respect to various *SH* and *SL* settings

for the workload log. Moreover, all the above observations show that neither *BF* nor *FF* can always perform the best under all possible workload conditions.

In order to clearly present the performance of the proposed *intelligent* methods, we use the metric *TNPIR*, which is first introduced in Sect. 5.2, with some modification to evaluate the performance of each allocation method. The modification is that here the *NPIR* is normalized with respect to the best result of *BF* and *FF* and is defined

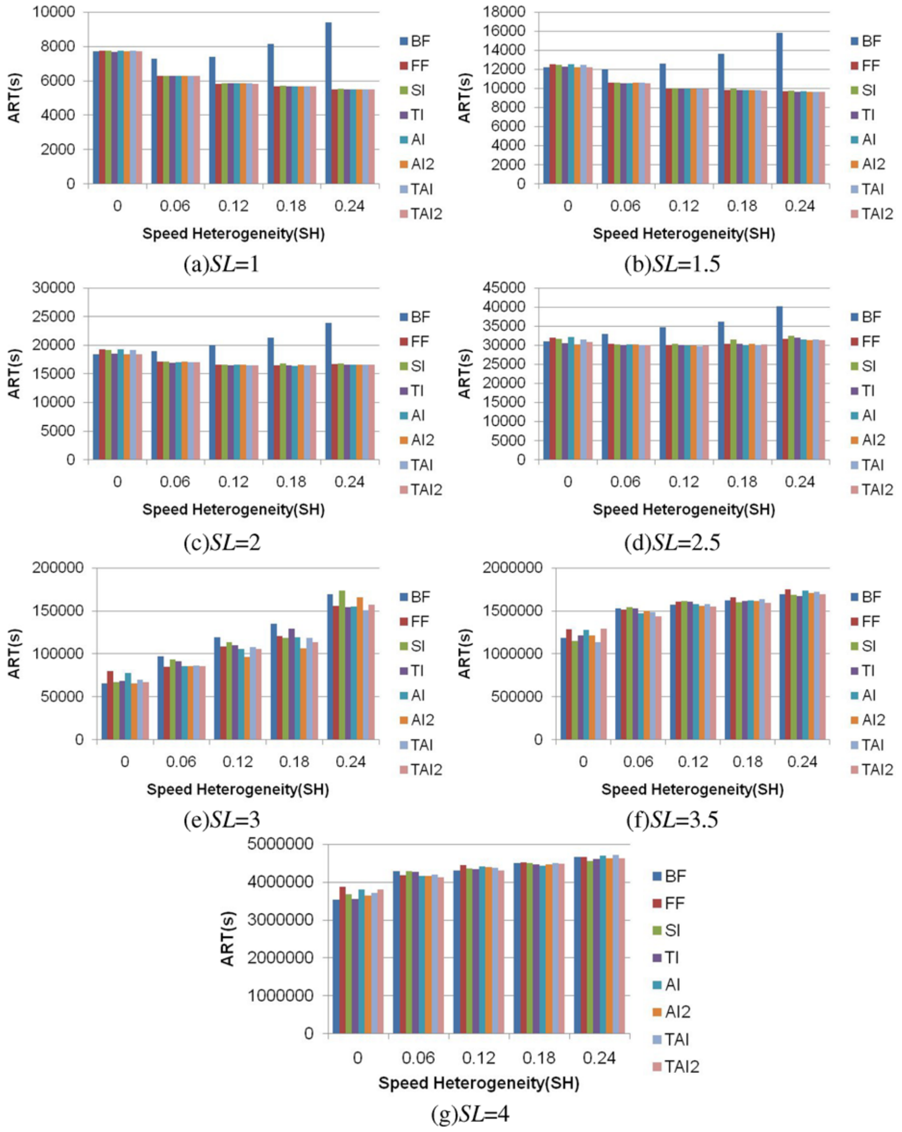

Fig. 6 Result of workload log for $RC = 75\%$ with respect to various SH and SL settings

in (23),

$$NPIR(am) = \frac{\text{Min}(ART_{BF}, ART_{FF}) - ART_{am}}{\text{Min}(ART_{BF}, ART_{FF})} \tag{23}$$

where *am* is a variable to represent a specific allocation method and ART_{am} is the average response time of the allocation method *am*. The *TNPIR* is the summation of the *NPIR* values for the set of all parameter settings. For the experiments with the workload log, there are totally 140 parameter settings covering all combinations of

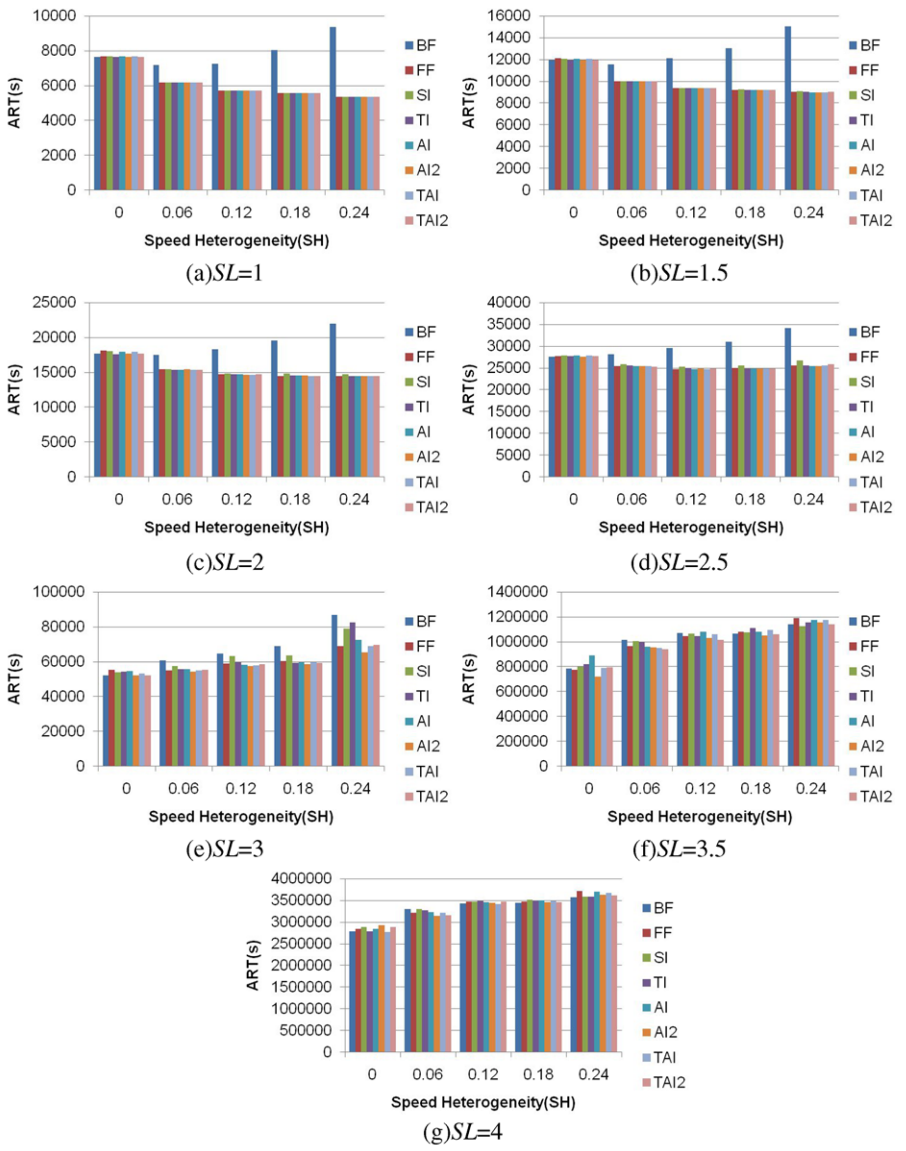

Fig. 7 Result of workload log for $RC = 50\%$ with respect to various *SH* and *SL* settings

5 *SH* values, 7 *SL* values, and 4 *RC* values. For workload models, each model consists of 25 parameter settings.

The performance evaluations using metric *TNPIR* for the workload log and workload models are shown in Tables 11 and 12, respectively. The result in Table 11 shows that all the proposed *intelligent* methods (*SI*, *TI*, *AI2*, and *TAI2*) result in performance improvement compared to *BF* and *FF*, especially for *TI*, *AI2*, and *TAI2*. This observation also holds for Lublin99 and Feitelson96 models showed in Table 12. These

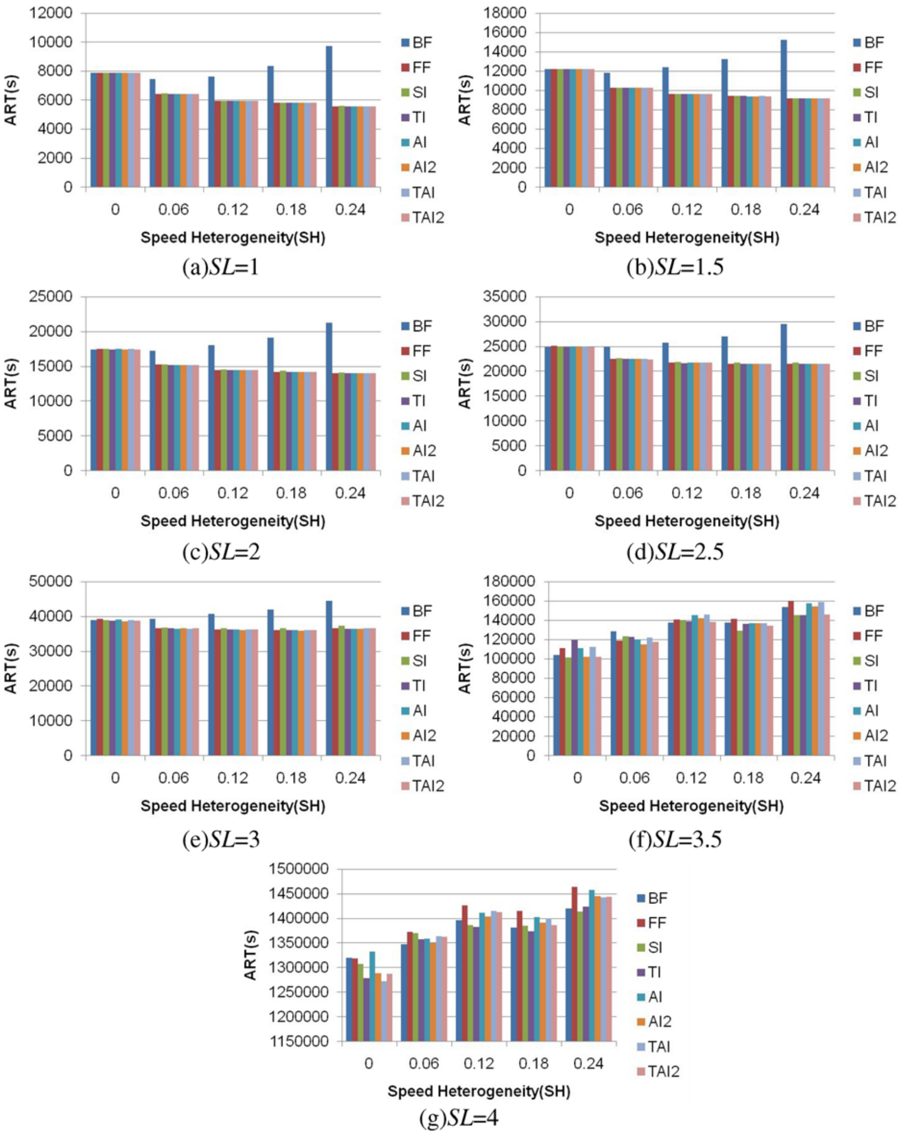

Fig. 8 Result of workload log for $RC = 25\%$ with respect to various *SH* and *SL* settings

results show that the proposed *intelligent* allocation methods can yield better performance in terms of *ART* than *BF* and *FF*.

For the performance of the proposed *AI2* and our previous work *AI*. The results in Tables 11 and 12 show the same trend that *AI2* outperforms *AI* significantly. Even in the mixed approach, we can observe that *TAI2* outperforms *TAI*. Thus, we can conclude that the new proposed queue-based allocation method *AI2* does improve the performance of our previous work *AI* in terms of *ART*.

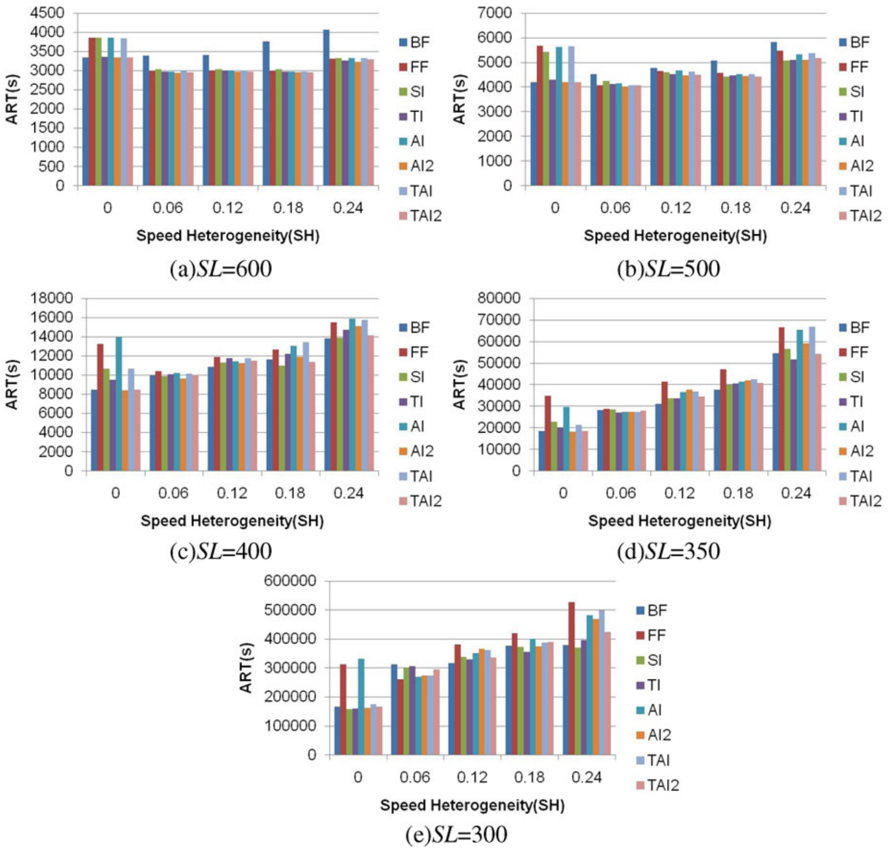

Fig. 9 Result of Feitelson96 workload model with respect to various *SH* and *SL* settings

Table 11 *Total normalized performance improvement ratio* (*TNPIR*) of each allocation method for workload log under all 140 parameter settings (each *RC* setting contains 35 set of results)

RC setting	*BF*	*FF*	*SI*	*TI*	*AI*	*AI2*	*TAI*	*TAI2*
$RC = 100\%$	−473%	−93%	−69%	−44%	−86%	−17%	4%	14%
$RC = 75\%$	−522%	−60%	−53%	−21%	−36%	17%	−10%	7%
$RC = 50\%$	−600%	−25%	−75%	−45%	−41%	19%	−13%	1%
$RC = 25\%$	−594%	−29%	−4%	−7%	−24%	2%	−22%	7%
Total	−2189%	−208%	−202%	−117%	−186%	22%	−40%	29%

For the performance of the three proposed approaches (threshold-based, queue-based, and mixed) we observe that the queue-based approach is better than the threshold-based approach (*AI2* surpasses *TI* and *SI*) and the mixed approach is better than the queue-based approach (*TAI2* surpasses *AI2*) in the result with the workload log. However, for workload models, the above observation does not hold. The relative performance between the methods is *TAI2* > *TI* > *AI2* > *SI* in Feitelson96 model

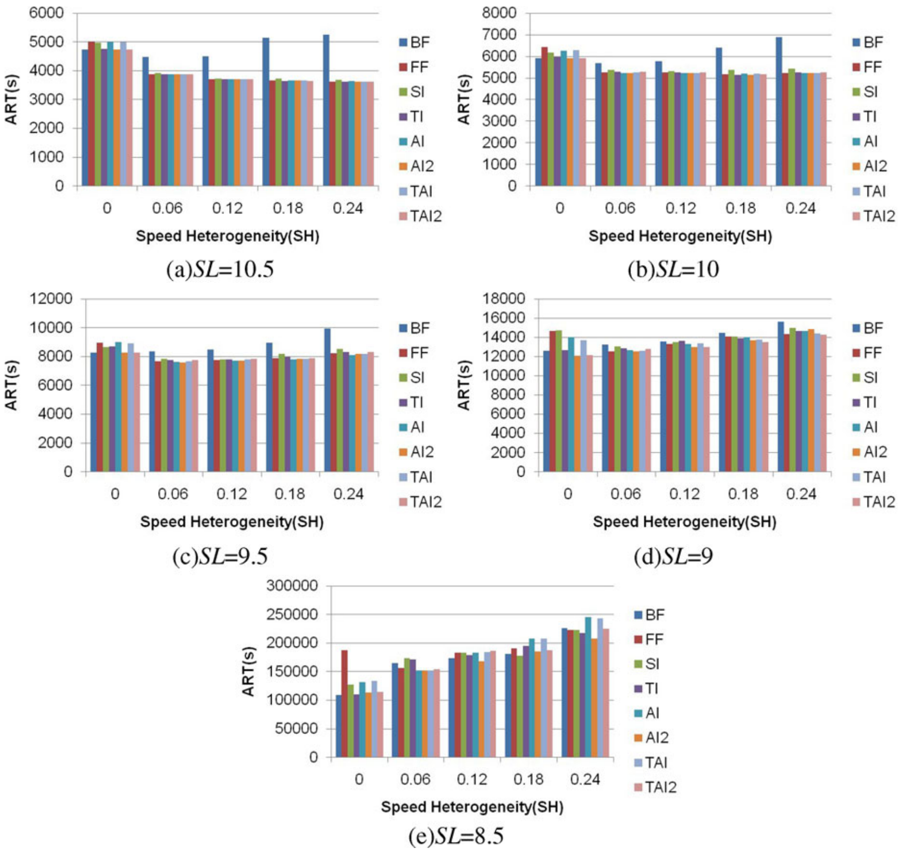

Fig. 10 Result of Lublin99 workload model with respect to various *SH* and *SL* settings

Table 12 *Total normalized performance improvement ratio* (*TNPIR*) of each allocation method for Feitelson96 and Lublin99 workload models under all 25 parameter settings

Workload model	*BF*	*FF*	*SI*	*TI*	*AI*	*AI2*	*TAI*	*TAI2*
Feitelson96	−127%	−469%	−121%	−56%	−396%	−67%	−236%	−43%
Lublin99	−274%	−122%	−97%	−36%	−79%	15%	−77%	−10%

while in Lublin99 model the relationship is *AI2* > *TAI2* > *TI* > *SI*. By examining the results of *AI2* and *TAI2* in each parameter setting, we found that *TAI2* is relatively more stable than *AI2*. The standard deviation of *AI2* and *TAI2* calculated according to *NPIR* is shown in Table 13. We infer that the queue-based approach has great potential to outperform threshold-based approach since it utilizes the information provided by waiting queue. However, since the threshold-based approach is proposed according to our analysis in Table 1, it is expected to deliver more stable performance than the queue-based approach. In another word, the performance of the threshold-based approach is assumed to be close to the best result of *BF* and *FF* but may not be better

Table 13 The standard deviation of *AI2* and *TAI2* calculated according to *NPIR*

Method	Workload log $RC = 100$	Workload log $RC = 75$	Workload log $RC = 50$	Workload log $RC = 25$	Feitelson96 model	Lublin99 model
AI2	0.024	0.031	0.019	0.010	0.079	0.022
TAI2	0.022	0.026	0.012	0.011	0.049	0.023

than the best of *BF* and *FF*. Thus, the mixed approach could deliver better performance than the threshold-based approach and achieve more stable performance than the queue-based approach in average. The results showed in Tables 11 and 12 are consistent with the above inference that *TAI2* successfully integrates the advantages of the queue-based and threshold-based approaches, therefore, always better than *TI* and more stable than *AI2*.

Furthermore, we find in some cases the *TNPIR* values of *AI2* and *TAI2* are positive. Note that *TNPIR* is normalized according to the best result of *BF* and *FF* for each parameter setting. This positive result demonstrates that the new proposed queue-based allocation method *AI2* and the mixed allocation method *TAI2* can dynamically adapt to the better allocation decision between *BF* and *FF* in variant workload conditions, and thus deliver better performance under most workload and resource conditions.

5.4 Potential of dynamic threshold setting

In the above experiments, for the threshold-based approaches, a fixed (*SHT*, *SLT*) pair is used throughout all different workload and resource conditions, *i.e.*, different *SH*, *SL*, and *RC* values. In practical environment, administrators can adjust the (*SHT*, *SLT*) setting under different workload and resource conditions. In this section, we further study the potential of the threshold-based method with dynamic threshold setting. Here we concentrate on three allocation methods with adjustable threshold values, which are *TI*, *TAI*, and *TAI2*. For these three methods, all the combinations of four *SHT* values (0.05, 0.1, 0.15, and 0.2) and five *SLT* values (1, 1.5, 2, 2.5, and 3) are put into simulation for each workload and resource condition and we choose the best result in all these 20 threshold pairs to represent the performance of a method under that workload and resource condition. In this experiment, we can observe the potential (the best performance that can be achieved) of the dynamic threshold approach. All 8 allocation methods are simulated together with all 140 parameter settings (see Table 7) for the workload log and 25 parameter settings (see Table 9) for each workload model. The performance metric *TNPIR* normalized with respect to the best result of *BF* and *FF* is used to evaluate the performance of each allocation method. Note that here we only generate one speed vector for each SH value for simulation.

Tables 14 and 15 show the results for the workload log and workload models respectively. In order to avoid confusion, the symbol * is appended to the 3 allocation methods to indicate that the result is represented by the best result in 20 threshold settings. These results show that if we can find the best threshold setting for each workload and resource condition, the performance can be improved substantially, compared with the fixed threshold setting (the results of *TI*, *TAI*, and *TAI2* in Tables 11 and 12).

Table 14 The result for the potential of dynamic threshold setting for workload log in all 140 parameter settings (each *RC* setting contains 35 results)

RC settings	*BF*	*FF*	*SI*	*TI**	*AI*	*AI2*	*TAI**	*TAI2**
$RC = 100\%$	−627%	−127%	−126%	58%	−78%	1%	132%	123%
$RC = 75\%$	−605%	−61%	−135%	56%	−64%	22%	76%	103%
$RC = 50\%$	−657%	−25%	−133%	−6%	−62%	−20%	41%	46%
$RC = 25\%$	−615%	−22%	−37%	32%	−19%	7%	37%	40%
Total	−2504%	−235%	−431%	140%	−223%	10%	286%	313%

Table 15 The result for the potential of dynamic threshold setting for both workload models in all 25 parameter settings

Workload model	*BF*	*FF*	*SI*	*TI**	*AI*	*AI2*	*TAI**	*TAI2**
Fei96	−439%	−350%	−73%	78%	−233%	79%	25%	139%
Lub99	−371%	−120%	−146%	45%	−75%	−13%	27%	54%

For the potential of these 3 methods, *TAI2* is shown to have the most performance improvement compared to the other two methods in both the workload log and the two workload models. This finding motivates us to further study how to find the best threshold setting dynamically in the future.

6 Conclusions

This paper presents several intelligent processor allocation methods to improve system performance in heterogeneous grid environments. We first analyze the relative strength of existing allocation methods and present three different approaches to dynamically switch allocation decision based on current workload and resource conditions. Based on these approaches, several *intelligent* processor allocation methods are developed through considering both speed heterogeneity and resource fragmentation. Extensive simulation studies have been conducted to evaluate the proposed methods. The experimental results show that all the proposed *intelligent* methods result in better performance than *best-fit* and *fastest-first* individually. Furthermore, the proposed mixed allocation method *threshold-based adaptive intelligent 2* is shown to dynamically adapt to the better allocation decision between *best-fit* and *fastest-first,* and thus deliver better performance under most workload and resource conditions.

It is difficult to develop a processor allocation method which can always perform the best under all possible workload and resource conditions. In addition to the proposed methods, the extensive simulation analysis of different allocation methods under various workload and resource conditions in this paper can serve as a good basis for better understanding of the root causes of the performance difference between the methods. The understanding could in turn help develop more effective processor allocation methods in the future. The results in Sect. 5.4 show that the method with threshold setting has great potential to be further improved by dynamic threshold setting. It is thus a promising future research direction on how to find the best threshold settings dynamically under different workload and resource conditions.

Acknowledgements This paper is based upon work supported by National Science Council (NSC), Taiwan, under grants no. NSC 96-2221-E-007-130-MY3, NSC 96-2221-E-432-003-MY3, and 98-2218-E-007-005-. The authors also thank all of the people for comments and advice.

References

1. Banen S, Bucur AID, Epema DHJ (2003) A measurement-based simulation study of processor co-allocation in multicluster systems. In: The 9th workshop on job scheduling strategies for parallel processing. Lect notes comput sci, vol 2862. Springer, Berlin, pp 105–128
2. Brune M, Gehring J, Keller A, Reinefeld A (1999) Managing clusters of geographically distributed high-performance computers. J Concurr Comput Pract Exp 11(15):887–911
3. Bucur AID, Epema DHJ (2001) The influence of communication on the performance of co-allocation. In: The 7th international workshop on job scheduling strategies for parallel processing. Lect notes comput sci, vol 2221. Springer, Berlin, pp 66–86
4. Bucur AID, Epema DHJ (2002) Local versus global schedulers with processor co-allocation in multicluster systems. In: The 8th international workshop on job scheduling strategies for parallel processing. Lect notes comput sci. Springer, Berlin, pp 184–204
5. Bucur AID, Epema DHJ (2003) The performance of processor co-allocation in multicluster systems. In: Proceedings of the third IEEE international symposium on cluster computing and the grid (CC-Grid'03), pp 302–309
6. Bucur AID, Epema DHJ (2007) Scheduling policies for processor coallocation. IEEE Trans Parallel Distrib Syst 18(7):958–972
7. Buyya R, Giddy J, Abramson D (2000) An evaluation of economy-based resource trading and scheduling on computational power grids for parameter sweep applications. In: Proceedings of the second workshop on active middleware services (AMS2000)
8. Buyya R, Abramson D, Giddy J, Stockinger H (2002) Economic models for resource management and scheduling in grid computing. Special issue on grid computing environments. J Concurr Comput Pract Exp 14(13–15):1507–1542
9. England D, Weissman JB (2004) Costs and benefits of load sharing in computational grid. In: 10th workshop on job scheduling strategies for parallel processing. Lect notes comput sci, vol 3277. Springer, Berlin, pp 160–175
10. Ernemann C, Hamscher V, Schwiegelshohn U, Streit A, Yahyapour R (2002) On advantages of grid computing for parallel job scheduling. In: Proceedings of 2nd IEEE international symposium on cluster computing and the grid (CC-GRID 2002), pp 39–46
11. Ernemann C, Hamscher V, Streit A, Yahyapour R (2002) On effects of machine configurations on parallel job scheduling in computational grids. In: Proceedings of international conference on architecture of computing systems (ARCS 2002), pp 169–179
12. Ernemann C, Hamscher V, Yahyapour R (2002) Economic scheduling in grid computing. In: The 8th international workshop on job scheduling strategies for parallel processing. Lect notes comput sci, vol 2537. Springer, Berlin, pp 128–152
13. Ernemann C, Hamscher V, Yahyapour R, Streit A (2002) Enhanced algorithms for multi-site scheduling. In: Proceedings of 3rd international workshop grid 2002, in conjunction with supercomputing 2002, pp 219–231
14. Ernemann C, Hamscher V, Yahyapour R (2004) Benefits of global grid computing for job scheduling. In: Proceedings of the fifth IEEE/ACM international workshop on grid computing (GRID'04), pp 374–379
15. Feitelson DG (1996) Packing schemes for gang scheduling. In: Job scheduling strategies for parallel processing. Lect notes comput sci, vol 1162. Springer, Berlin, pp 89–110
16. Feitelson DG, Rudolph L (1995) Parallel job scheduling: issues and approaches. In: Proceedings of IPPS'95 workshop: job scheduling strategies for parallel processing, pp 1–18
17. Feitelson DG, Rudolph L, Schwiegelshohn U, Sevcik KC, Wong P (1997) Theory and practice in parallel job scheduling. In: Job scheduling strategies for parallel processing, pp 1–34

18. Foster I, Kesselman C (1999) The grid: blueprint for a new computing infrastructure. Morgan Kaufmann, San Mateo
19. Hamscher V, Schwiegelshohn U, Streit A, Yahyapour R (2000) Evaluation of job-scheduling strategies for grid computing. In: Proceedings of the 7th international conference on high performance computing (HiPC-2000), pp 191–202
20. Huang KC, Chang HY (2006) An integrated processor allocation and job scheduling approach to workload management on computing grid. In: Proceedings of the 2006 international conference on parallel and distributed processing techniques and applications (PDPTA'06), pp 703–709
21. Huang KC, Shih PC, Chung YC (2007) Towards feasible and effective load sharing in a heterogeneous computational grid. In: Proceedings of the second international conference on grid and pervasive computing
22. Kwon OH, Chwa KY (1998) An method for scheduling jobs in hypercube systems. IEEE Trans Parallel Distrib Syst 9(9):856–860
23. Kwon OH, Kim J, Hong SJ, Lee SG (1997) Real-time job scheduling in hypercube systems. In: Proceedings of 1997 international conference on parallel processing (ICPP '97), p 166
24. Lifka D (1995) The ANL/IBM SP scheduling system. In: proceeding of international parallel and distributed processing symposium, workshop job scheduling strategies for parallel processing, pp 295–303
25. Lublin U, Feitelson DG (2003) The workload on parallel supercomputers: modeling the characteristics of rigid jobs. J Parallel Distrib Comput 63(11):1105–1122
26. Mu'alem AW, Feitelson DG (2001) Utilization, predictability, workloads, and user runtime estimate in scheduling the IBM SP2 with backfilling. IEEE Trans Parallel Distrib Syst 12(6):529–543
27. Ni LM, Turner SW, Cheng BHC (1995) Contention-free 2D-mesh cluster allocation in hypercubes. IEEE Trans Comput 44(8):1051–1055
28. Parallel Workloads Archive (2010) http://www.cs.huji.ac.il/labs/parallel/workload/
29. Qin J, Bauer MA (2009) An evaluation of communication factors on an adaptive control strategy for job co-allocation in multiple HPC clusters. In: 2009 15th international conference on parallel and distributed systems, pp 391–398
30. Rao I, Huh EN (2008) A probabilistic and adaptive scheduling algorithm using system-generated predictions for inter-grid resource sharing. J Supercomput 45(2):185–204
31. Sharma DD, Pradhan DK (1995) Processor allocation in hypercube multicomputers: fast and efficient strategies for cubic and noncubic allocation. IEEE Trans Parallel Distrib Syst 6(10):1108–1122
32. Silberschatz A, Peterson J, Galvin P (1991) Operating system concepts. Addison-Wesley, Reading
33. Skovira J, Chan W, Zhou H, Lifka D (1996) The EASY-LoadLeveler API project. In: Job scheduling strategies for parallel processing, pp 41–47
34. Zhang W, Cheng AMK, Hu M (2006) Multisite co-allocation algorithms for computational grid. In: Proceedings of the 20th international parallel and distributed processing symposium, pp 906–926
35. Zhu Y, Han J, Liu Y, Ni LM, Hu C, Huai J (2005) TruGrid: a self-sustaining trustworthy grid. In: Proceedings of the first international workshop on mobility in peer-to-peer systems (MPPS), pp 815–821