The Journal of Supercomputing (2020) 76:10212–10239
https://doi.org/10.1007/s11227-020-03246-6

Two-level utilization-based processor allocation for scheduling moldable jobs

Ying-Jhih Wu[1] · Shuo-Ting Yu[1] · Kuan-Chou Lai[1] · Amit Chhabra[2] · Hsi-Ya Chang[3] · Kuo-Chan Huang[1]

Published online: 16 March 2020

Abstract
Most modern parallel programs are written with the moldable property. However, most existing parallel computing systems treat such parallel programs as rigid jobs for scheduling, resulting in two drawbacks. The first is inflexibility and inefficiency in processor allocation, leading to resource fragmentation and thus poor performance. The second is about usage inconvenience, requiring users to figure out the best number of processors for executing a job. As HPC as a service emerges, moldable job scheduling has become an important research issue for achieving both high performance and user convenience. This paper presents our research work on developing new processor allocation approaches for moldable job scheduling based on two-level resource utilization calculation, preemptive job execution, and dual-criteria iterative improvement. A series of simulation experiments have been conducted to evaluate the proposed approaches and compare them to previous methods. The experimental results demonstrate significant performance improvement in terms of average turnaround time.

Keywords Resource utilization · Processor allocation · Moldable job scheduling · HPC as a service

1 Introduction

Parallel jobs, according to their flexibility in parallelism, can be classified into four types [1]: (1) rigid, (2) moldable (3) evolving, and (4) malleable. A rigid job can only run with a specific number of processors specified by the user upon job submission. Moldable jobs are flexible in the number of processors to use, but the number cannot be changed during execution. Malleable and evolving jobs are similar to moldable jobs in that they all have the potential to run with different parallelisms in

✉ Kuo-Chan Huang
kchuang@mail.ntcu.edu.tw

Extended author information available on the last page of the article

contrast to rigid jobs. However, they are even more flexible, being able to change the number of used processors dynamically during execution. While both evolving and malleable jobs can change their processor requirements during execution, the change is application initiated for evolving jobs, but system initiated for malleable jobs [2, 3].

Most modern parallel programs are written with the moldable property. For example, the famous benchmark program HPL,[1] used for ranking the top 500 supercomputers in the world,[2] has the moldable property, so that it can be used for evaluating the performance of various parallel computers of different scales easily without modifying its source code. Another example is a dynamic moldable tree search (DMTS) framework proposed for solving the Boolean satisfiabilitфy problem in [4]. The DMTS framework was designed to run on various parallel computing infrastructures, including clusters, computational grids, and clouds. Several MPI (message passing interface) libraries are available for developing moldable parallel programs conveniently. However, most existing workload management systems on parallel computing platforms usually treat such moldable programs as rigid jobs when scheduling them. Ignoring parallel jobs' moldable property results in two major drawbacks: inefficient processor allocation and user inconvenience.

Users accessing traditional HPC (high-performance computing) facilities are usually required to specify the number of processors to use upon job submission. A job scheduler on the HPC facility will allocate processors to each job for exclusive use according to the specified number of processors. However, if a job requests a number of processors larger than currently available, it would have to wait while the available processors are kept idle, resulting in both degraded resource utilization and enlarged job turnaround time. Usage inconvenience is because users usually do not know how to determine a most appropriate number of processors for achieving the best performance. This is because the turnaround time of a submitted job is composed of two parts: waiting time and execution time. Waiting time measures the time period between job submission and when the requested processors become available. Execution time is the time spent on executing a job with exclusive use of a specific number of processors. A user might know how many processors could lead to the shortest execution time for his job, but usually has no idea about how long the waiting time could be once requesting a specific number of processors, because of resource competition among jobs.

A possible solution to overcome the above drawbacks is let the job scheduler to determine a most appropriate number of processors for each moldable job. Therefore, processor allocation for moldable job scheduling is becoming an ever important issue as the concept of HPC as a service (HPCaaS) [5] emerges. HPCaaS is a new service model aiming to transform traditional high-performance computing into a more convenient and accessible practice. Two important concerns of HPCaaS are easier access to HPC facilities and efficient parallel job scheduling for good performance [5]. Unlike users of traditional HPC facilities, who usually run parallel

[1] https://www.netlib.org/benchmark/hpl/.

[2] https://www.top500.org/.

programs developed by themselves, most HPCaaS users might run parallel applications developed by others as services and do not know their parallelism characteristics well. In such cases, users simply want to get their jobs done as soon as possible and do not want to or even have no idea on how to specify an appropriate number of processors for the best application performance. On the other hand, the job scheduler on the HPCaaS platform would have better knowledge than a user to determine the most appropriate number of processors to use, benefiting both overall system performance and the specific job's turnaround time.

This paper presents our research work on developing new processor allocation approaches for moldable job scheduling based on iterative processor allocation adjustment. Four processor allocation approaches are proposed and evaluated in this paper. The first is a utilization-based approach using two-level resource utilization calculation. The second is a bounded allocation approach which would limit the maximum number of processors that each moldable job could use. The third is a preemptive processor allocation approach which can preempt and reschedule running jobs to improve overall system performance. Both the second and the third approaches were developed for resolving the issue that earlier jobs occupy too many resources to allow efficient execution of jobs coming later. The fourth is a dual-criteria approach which can avoid occasional processor allocation pitfalls where high resource utilization leads to poor job turnaround time when system workload is at a medium or light level.

A series of simulation experiments based on publically available workload data have been conducted to evaluate the proposed approaches and compare them to previous methods [6–8] in the literature. The experimental results demonstrate significant performance improvement in terms of average turnaround time. The remainder of this paper is organized as follows. Section 2 discusses related works on parallel job scheduling. Section 3 presents our processor allocation approaches and compares them with previous methods in [6–8] using illustrative examples. Section 4 presents the results of performance evaluation. Section 5 concludes this paper and discusses possible future research directions.

2 Related work

Parallel job scheduling has long been an important research topic for operating and managing high-performance computing platforms efficiently. Most of the earlier research works focused on rigid jobs since it is the way traditional workload management systems on HPC platforms treat submitted parallel jobs. The problem of scheduling parallel jobs can be further divided into two critical issues: job sequencing and processor allocation. Job sequencing determines the execution order of jobs, while processor allocation deals with how many and which processors to use for each job.

Although for rigid jobs the numbers of processors to use are already specified by the submitting users, job schedulers still have to determine the exact portion of processors in a system for allocation which might have great influence on subsequent jobs and thus the overall system performance on some parallel computers of specific internetworking architecture, e.g., hypercube. Therefore, both job sequencing [9, 10] and processor allocation [11, 12] received a lot of research attention on earlier hypercube-based parallel computers. However, on recent commonly used switch-based parallel computers and cluster systems, processor allocation for rigid jobs has become straightforward and received less research attention than job sequencing since allocation to different portions of system resources has negligible performance impact on jobs.

First-come–first-serve (FCFS) might be the most widely used job sequencing policy in practical computing systems because of its simplicity and fairness property. With the FCFS policy, a job scheduler allocates processors to the jobs in the waiting queue for execution according to their ascending order of submission time. The major problem of FCFS is its unnecessary low resource utilization [13]. Therefore, many research efforts [1, 13, 14, 15, 16] have been spent on developing more effective job sequencing mechanisms for scheduling rigid jobs on parallel computers. Backfilling is one of the job sequencing mechanisms receiving a lot of research attention [17–22].

In contrast to strict FCFS, backfilling is a flexible and effective strategy that tries to make a balance between raising resource utilization and maintaining a certain degree of fairness. EASY backfilling is the first well-known backfilling method proposed in [15] and widely used in many production parallel systems, e.g., ANL/IBM SP system [15] and MAUI scheduler [23]. With the EASY backfilling mechanism, a job scheduler would grant the first job in the waiting queue a reservation of system resources and then allow some subsequent jobs to run out of their order in the waiting queue provided that such out-of-order execution would not delay the job with reservation. EASY backfilling has several variants [24–26] which differ in the order of jobs to be backfilled.

Backfilling mechanisms need the information of job execution time to arrange resource reservation and backfilling activities. To adopt backfilling approaches, users are usually asked to provide estimated job execution time upon job submission in practical systems. However, such estimation is hardly to be precise since for many jobs the actual execution time could only be known after the finish of their execution. Wong and Goscinski investigated the influence of inaccurate estimation of job execution time on the performance of the EASY backfilling approach [20]. There are also research efforts devoted to improve the accuracy of performance prediction of parallel applications [25, 27, 28, 29].

There are also many research works dealing with the problem of scheduling rigid jobs with deadlines [30–34]. Earliest deadline first (EDF) has been one of the most well-known job sequencing heuristics for scheduling jobs with deadline. EDF was adopted to schedule real-time tasks on multicore processors in [35]. In [36], EDF was applied to maintain timeliness and data freshness in real-time database research. Another popular job sequencing heuristic, called least-laxity-first (LLF), was used in [32] for scheduling jobs with deadlines in distributed systems. For scheduling moldable jobs, it is hard to apply the LLF heuristic since the required execution time of a moldable job changes with the number of processors used, hindering the calculation of an appropriate laxity.

Job deadline is a crucial issue in real-time systems. Priority-based approaches were presented in [30] for scheduling tasks with deadlines. Many scheduling approaches for real-time systems concerning deadline were reviewed and compared in [37], including EDF, LLF, and rate-monotonic algorithms. Some research works, e.g., [31, 38], consider the scheduling issues of tasks with deadline constraints in cloud computing environments. The work in [31] presented an adaptive resource management policy for handling requests of deadline-bound applications. Pop investigated the problem of remote scheduling of periodic and sporadic tasks with deadline constraints on cloud computing platforms in [38].

As an increasing number of modern parallel applications are designed to have the moldable property, moldable job scheduling has received significant research attention for the past few years. In [39, 40], Srinivasan et al. proposed an aggressive fair-share strategy and a combined moldable scheduling strategy for moldable jobs, adopting a profile-based allocation scheme. The strategies keep track of the information about all the free-time slots available in current schedule and scan them to find the most suitable one for a moldable job at each scheduling activity, considering the effects of partition size on the performance of the application. The strategies thus need to have the knowledge of job execution time. Our approach in this paper also utilizes the knowledge of job execution time and uses a profile-based scheduling mechanism. However, our approach takes into account resource utilization and adopts an iterative procedure to make better scheduling decisions.

Cirne and Berman explored the issues of selecting appropriate partition sizes in application-level moldable job scheduling [41, 42]. In their work, users would provide a set of candidate requests for different numbers of processors, and an application-level scheduler is used to effectively select the most suitable request based on resource availability and workload conditions. Sabin et al. [43] proposed an iterative algorithm for moldable job scheduling. Their approach features utilizing jobs' parallel efficiency characteristics to achieve better schedules. In this paper, our approach also takes into consideration jobs' parallel efficiency information when making processor allocation decisions. However, in addition to jobs' parallel efficiency, our approach also considers the effects of resources' idle time slots on the overall system performance using two-level resource utilization calculation. Caniou et al. [44] studied the benefits of having a middleware able to automatically submit and reallocate requests for moldable job execution from one site to another in a grid environment. The middleware can automatically tune the number of processors to be used by the moldable jobs for shorter turnaround time.

Huang proposed and evaluated four heuristics for allocating processors to moldable jobs in [45]. Those heuristics require no prior information of moldable jobs, e.g., job execution time, and make the processor allocation decision for each moldable job at its start time instead of submission time, i.e., when it becomes the first job in the waiting queue. The heuristics utilize only the information about the number of free processors in the system and the content of waiting queue when making processor allocation decisions. A processor allocation approach for moldable job scheduling was proposed in [7] for HPCaaS, which takes advantage of job execution time information to improve overall system performance by increasing resource utilization. However, the approach in [7] only considers the effects of processors' idle time slots when estimating resource utilization. On the other hand, in this paper, our two-level resource utilization calculation also takes into consideration the effects of parallel jobs' efficiency on resource utilization and thus has potential for producing better schedules.

The extreme-ending moldable approach (EEMA) in [6] is similar to the adaptive scaling down approach in [45], where if a parallel job requests a number of processors which at that moment is larger than the number of free processors, instead of keeping the job waiting in queue, the scheduler automatically scales the job down to use exactly the number of free processors for immediate execution. This approach can effectively prevent parallel jobs requesting large numbers of processors from unnecessary waiting. However, it does not address the whole issue of idle time slots well since it would not scale up parallel jobs requesting small numbers of processors to fully utilize the available resources. Therefore, our approach with two-level resource utilization calculation has potential to achieve better performance than EEMA.

Wu et al. proposed a resource allocation scheme, named HRF (highest revenue first), for moldable jobs in [8]. They first proposed an indicator called revenue per processor (RP), which expresses the revenue of shortening runtime with every allocated processor. Based on this indicator, the HRF scheme always allocates processors to the job with the highest RP, in order to reduce the average turnaround time of all jobs. HRF is similar to our approach in the aspect of allocating more processors to jobs of higher parallel efficiency. Moreover, both HRF and our approach decide the number of allocated processors at schedule time. By combining the processor allocation schemes with different job selection schemes, there could be various moldable job scheduling policies as shown in [8]. In HRF, there are two parameters, α and *threshold*, to set before the algorithm can be applied. However, an appropriate setting of these parameters is no easy since it depends on system workload and parallel programs' speedup characteristics as shown in the experiments in [8]. On the other hand, our approaches are easier to apply since there are no such kinds of parameters to be set manually by system administrators. In addition, the parameter *threshold* in HRF only considers the effects of parallel jobs' efficiency characteristics, while our approaches also take into consideration the influences of idle time slots.

Compared to the huge amount of previous research work on rigid job scheduling in the literature, there is still a significant research gap in moldable job scheduling. The previous work also shows that processor allocation is a major issue in scheduling moldable jobs, and job execution time information is valuable for making good allocation decisions. Table 1 summarizes and compares the characteristics of the previous parallel job scheduling approaches discussed in the above.

To accelerate the execution of some high-performance demanding applications, e.g., training of AI systems, or accommodate various applications of different parallelism characteristics, there is an increasing interest in heterogeneous computing systems, e.g., parallel computers equipped with both CPUs and GPUs. Applications can conduct heterogeneous computation by exploiting different kinds of parallelism on CPUs and GPUs, respectively. However, developing parallel applications to exploit both CPUs and GPUs is a much more challenging

Table 1 Comparison of parallel job scheduling approaches (Y: yes; N: no)

References	Job type	Deadline constrained	Need execution time information	Scheduling issues focused
[6]	Moldable	N	N	Processor allocation
[44]	Moldable	N	Y	Processor allocation
[41]	Moldable	N	Y	Processor allocation
[42]	Moldable	N	Y	Processor allocation
[45]	Moldable	N	N	Processor allocation
[7]	Moldable	N	Y	Processor allocation
[13]	Rigid	N	Y	Job sequencing
[43]	Moldable	N	Y	Processor allocation
[39]	Moldable	N	Y	Processor allocation
[40]	Moldable	N	Y	Job sequencing and processor allocation
[8]	Moldable	N	Y	Processor
[9]	Rigid	Y	Y	Job sequencing
[10]	Rigid	N	Y	Job sequencing and processor allocation
[11]	Rigid	N	N	Processor allocation
[12]	Rigid	N	N	Processor allocation
[25]	Rigid	N	Y	Job sequencing
[22]	Rigid	N	Y	Job sequencing
[35]	Rigid	Y	Y	Job sequencing
[30]	Rigid	Y	Y	Job sequencing
[31]	Rigid	Y	Y	Job sequencing
[32]	Rigid	Y	Y	Job sequencing
[33]	Rigid	Y	Y	Job sequencing
[34]	Rigid	Y	Y	Job sequencing

work than writing parallel programs running on homogeneous CPUs. In addition, the demand for heterogeneous computation to improve performance is also application dependent. Therefore, usually both homogeneous and heterogeneous parallel computations for different applications are conducted on a heterogeneous computing platform containing both CPUs and GPUs.

Regarding parallel job scheduling, although there is a new need for research on GPU scheduling [46, 47] or co-scheduling of both CPUs and GPUs [48, 49], scheduling parallel jobs running on homogeneous CPUs is still an important and relevant research direction with many challenging issues to explore [50, 51]. In this paper, we focus on the processor allocation problem of moldable job scheduling on homogeneous computing systems.

Energy consumption is also an important issue on some computing platforms, e.g., embedded systems and cloud computing environments, especially as green computing has become a worldwide concern. Therefore, energy-aware scheduling approaches have received a lot of research attention recently [52–54]. However, our research work in this paper focus on the improvement of execution performance which is still very important and relevant as shown in [46, 47, 50, 51].

In addition to effective job scheduling, there are still others issues which would affect application execution performance greatly. Application software optimization is an important one of them which needs to consider many parameters at compile time and runtime. The challenge is to determine an optimal set of parameters in a specific parallel computing environment, which is outside the scope of our research work in this paper. A systematic review on this topic can be found in [42], which focuses on the techniques based on machine learning or meta-heuristics.

3 Processor allocation in moldable job scheduling

This section explores the issues of processor allocation for moldable job scheduling. We first discuss three previous methods proposed in [6–8] and then present our four processor allocation approaches, followed by illustrative examples demonstrating the advantages of our approaches compared to previous methods.

3.1 Improved moldable job scheduling for HPCaaS

The work in [7] first investigates the influence of applications' speedup characteristics on moldable job scheduling strategies. The authors found that the parallel policy, i.e., starting as many jobs simultaneously as possible, can achieve better overall system performance for most speedup characteristics, e.g., Amdahl's law [55] and Downey's speedup model [56, 57], while the serial policy, i.e., allowing a job to use as many processors as possible, would perform better for the linear speedup model [58]. Then, they developed a moldable job scheduling method for parallel applications whose speedup characteristics conform to Amdahl's law [55] or Downey's speedup model [56, 57]. The proposed method is based on a hybrid of serial and parallel policies, and takes advantage of job execution time information to improve the overall system performance.

The moldable job scheduling method in [7] works as follows. It maintains two queues for running and waiting jobs, respectively. On each job scheduling activity, it first scans the running queue to collect the time instants of expected future processor releases resulting from the finishes of running jobs based on the job execution time information. Then, it counts the number of jobs in the waiting queue. If the number of waiting jobs is less than the number of future processor release events, it simply adopts the serial policy and allocates only the first job for execution. On the other hand, if the number of waiting jobs is larger than the number of future processor release events, it uses the parallel policy to allocate the first n jobs for execution. The value of n is calculated by subtracting the number of running jobs from the number of waiting jobs. In this way, the proposed moldable scheduling method can improve resource utilization and result in shorter average turnaround time of all jobs.

Figure 1 shows an illustrative example schedule produced by the moldable job scheduling method in [7], which demonstrates its superiority over the simple parallel

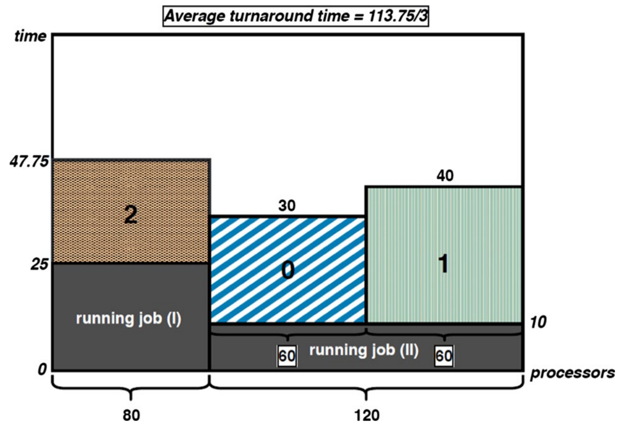

Fig. 1 Moldable job scheduling method in [7]

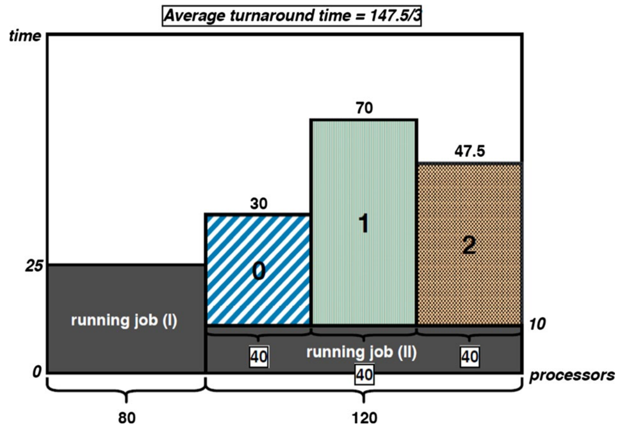

Fig. 2 Parallel policy

policy, i.e., the schedule in Fig. 2. In Fig. 2, the parallel policy tends to run all the jobs in queue, tasks I, II, and III, simultaneously after the first running job finishes. However, if the waiting queue remains empty till the second running job finishes, the released 80 processors will become idle and result in degraded resource utilization. The parallel policy also leads to longer execution time for tasks I, II, and III in this case. On the other hand, in Fig. 1, task III is not allocated immediately after the first running job finishes. Instead, its allocation is delayed until the second running job finishes. In this way, although the waiting time of task III is increased, all three tasks are allocated more processors for execution, compared to Fig. 2, resulting in shorter turnaround time in average. The experimental results in [7] show that the proposed moldable job scheduling method outperforms previous approaches [39, 40] significantly.

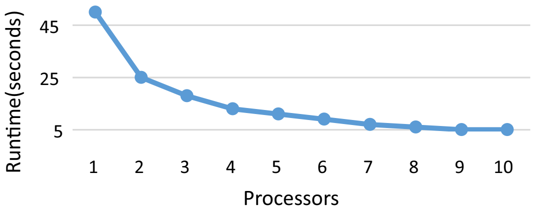

Fig. 3 Typical relationship between runtime and number of processors for a parallel job

3.2 HRF resource allocation scheme

The highest revenue first (HRF) allocation scheme proposed in [8] is based on an indicator named RP (revenue per processor) expressing the shortening runtime of a job if an extra processor is allocated to it. The RP of a job varies with the number of processors already allocated to it. Moreover, since the parallel efficiency of a job is usually not 100% and decreases as the number of used processors increases, the RP of a job also decreases as more processors are allocated. Figure 3 describes a typical relationship between a job's runtime and the number of allocated processors. The RP is about 25 when only one processor is allocated to the job, but declines to about two after five processors have been allocated.

Each time the content of the waiting queue changes, the HRF scheme will be applied to redetermine the number of allocated processors for each moldable job in it. On each scheduling activity, the HRF scheme prepares a certain budget of processors and allocates them to all the moldable jobs. The processor budget is calculated by multiplying a parameter α to the total number of processors in a system. In addition to the processor budget shared by all moldable jobs, each job is also imposed an upper limit on the number of processors it can use, controlled by another parameter *threshold*. HRF iterates a loop for $\alpha \times m$ times, where m is the number of processors in the system. In each iteration, the job with the maximum RP is found and allocated one more processor. If the number of allocated processors of any considered job has reached its upper limit, i.e., ($threshold \times m$), it will not be considered in later iterations.

3.3 Extreme-ending moldable approach (EEMA)

An extreme-ending moldable approach (EEMA) was proposed in [6] for parallel job scheduling. EEMA differs from HRF and our approaches in that each job is specified a desired number of processors to use upon submission by the user although it has the moldable property. Therefore, EEMA treats all jobs as rigid initially, trying to allocate the specified number of processors to them, and takes advantage of a job's moldable property only when its desired number of processors is larger than the number of free processors during scheduling. The experimental results in [6] show that EEMA can achieve better average turnaround time than previous rigid job scheduling methods.

The entire algorithm of EEMA is an iterative process inspecting each job in the waiting queue. For each job, EEMA first gets information of its desired number of processors and the number of currently free processors in a system. If current number of free processors is enough for the job's demand, EEMA just allocates the demanded processors to it. Otherwise, EEMA changes its demand to the number of currently free processors taking advantage of its moldable property and then allocates that number of processors to it.

3.4 Our two-level utilization-based processor allocation

Resource utilization has long been known to have great influence on the performance of both overall system and individual jobs when dealing with job scheduling issues. Therefore, several backfilling approaches for scheduling rigid jobs have been developed based on mechanisms increasing resource utilization [17, 13]. For moldable job scheduling, some previous works also noticed the importance of resource utilization, e.g., [43].

In this section, we present a new utilization-based processor allocation approach for moldable job scheduling, which adopts an iterative procedure to gradually improve the performance for jobs in the waiting queue by managing to increase resource utilization in the schedule. Each time when a new job is submitted into the system, a rescheduling activity will be triggered, and the utilization-based processor allocation approach could be applied to determine the processor allocations of all jobs in the waiting queue. The resultant schedule is kept in a profile structure which records the future start time, expected finish time, and the number of processors to use for each job, like the profile mechanism in [17, 13]. On the other hand, when a running job finishes, the system will start certain jobs for execution according to the schedule recorded in the profile.

The utilization-based processor allocation approach is described in detail in Algorithm 1, where MLS() is a procedure like the one in [59] applying simple FCFS-based list scheduling to update the execution schedule of all the jobs according to current processor allocation result and Utilization() calculates the resultant resource utilization of the updated schedule. The main structure of the algorithm is an iterative procedure repeating until the resultant resource utilization does not keep increasing. In each iteration, the algorithm tries to find every time instant when there are still some free processors not utilized well, and allocate them in a one-processor-at-a-time manner to those jobs of higher parallel efficiency based on their current processor allocations. Finally, a job with

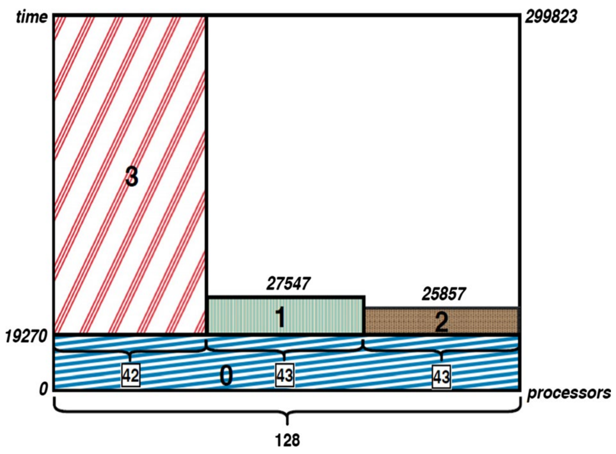

Fig. 4 Before one more processor allocated

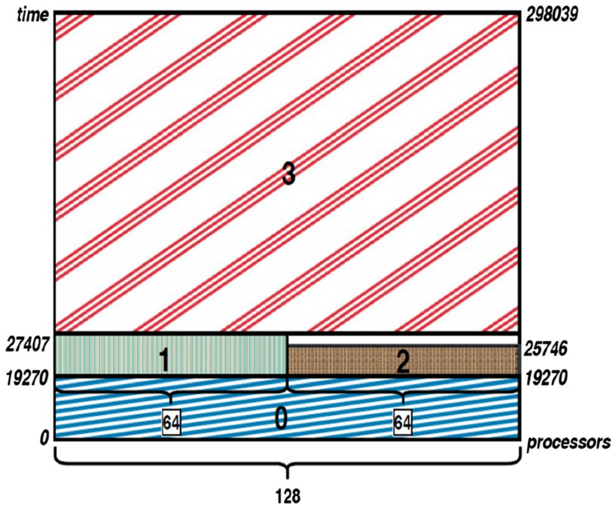

Fig. 5 After one more processor allocated

the latest finish time will be chosen to allocate one more processor intentionally. Such arrangement would delay the job's start time, but has the potential for reducing the execution time of it and all the jobs starting at the same time with it in the original schedule, resulting in a better schedule. Figures 4 and 5 show such an example with four tasks running on a parallel system of 128 processors, illustrating the execution schedules before and after task 3, i.e., the task with the latest finish time, being allocated one more processor, respectively.

Algorithm 1: Utilization-Based Processor Allocation for Moldable Jobs

```
Input:
    Q // the job waiting queue
    S // current schedule containing all running and waiting jobs except the
        newly submitted one
Output:
    S// updated schedule containing all running and waiting jobs
Variables: $np_t$ //number of free processors at time instant t
        s_list //a set of jobs starting at a certain time instant
        modified //a flag indicating whether the schedule has been altered or not

1: Remove the data for each waiting job in Q from S
2: Initialize each job in Q to use only one processor;
3: S = MLS( );
4: u = Utilization( );
5: do
   {
6:   modified = false;
7:   foreach (time instant t in the S when there are jobs finishing)
       {
8:       s_list = the jobs starting at time t in S;
9:       if ($np_t$ > 0 and s_list is not empty)
           {
10:          while ($np_t$ > 0)
               {
11:              pick the job in s_list with highest parallel efficiency;
12:              increase the number of processors reserved for the job by one;
13:              $np_t$--;
               }
14:          modified = true;
15:          S = MLS( );
16:          u = Utilization( );
17:          break;
           }
       }
18:  if (modified == false)
       {
19:      pick the job of the latest finish time in S;
20:      increase the number of processors reserved for the job by one;
21:      S = MLS( );
22:      u = Utilization( );
       }
23: } while (u keeps increasing)
```

In the above algorithm, the resource utilization of a schedule is calculated by a two-level mechanism as defined in the following formula (1), where *i* stands for each job in the schedule, np_i is the number of processors used by job i, t_i is the required job execution time, and E_i is the parallel efficiency of job i with np_i. The required job execution time, i.e., t_i, could be provided by users or some performance prediction algorithms, which is out of the scope of our research work in this paper. We assume the availability of such job execution time information as in most parallel job scheduling research [13, 18, 19, 22]. The parallel efficiency of a job running on a specific number of processors is defined in formula (2), i.e., dividing the speedup by the number of processors used, as in most parallel computing textbooks [58], where speedup with n processors is calculated by formula (3), dividing the execution time with one processor by the execution time with n processors.

The denominator in formula (1) measures the total resources occupied by the schedule in a time–space manner, where np is the total number of processors in the parallel system. The total resources occupied are calculated by multiplying np to the time period from the earliest start time to the latest finish time in the schedule. The numerator in formula (1) measures the actual portion of resources efficiently utilized by the jobs in the schedule, calculated by multiplying each job's execution time, number of processors used, and parallel efficiency together, and then summing the values of all jobs up. Since formula (1) considers two factors of resource utilization, i.e., idle time slots and the parallel efficiency of each job in the schedule, we call it a two-level mechanism for calculating resource utilization. Compared to most traditional ways for calculating resource utilization, which consider only the effects of idle time slots in the schedule, the proposed two-level resource utilization calculation mechanism is expected to more accurately measure actual resource utilization for finding better schedules.

$$\text{Utilization} = \frac{\sum_i t_i \times np_i \times E_i}{np \times (\text{finish time of latest job} - \text{earliest start time of running jobs})} \tag{1}$$

$$E = \frac{S}{n} \tag{2}$$

$$S = \frac{T_1}{T_n} \tag{3}$$

The following presents an example illustrating the superiority of our utilization-based processor allocation approach over the previous method in [7]. The example contains five moldable jobs for scheduling on a 128-processor parallel system. Their submission time and required serial execution time are listed in Table 2. In this example, Amdahl's law [55] is used as the speedup model, which can be represented by the following speedup formula, where α is the fraction of an application's parallelizable workload and n is the number of processors used.

$$S(n) = \frac{1}{(1-\alpha) + \frac{\alpha}{n}} \tag{4}$$

Each parallel job has the same α value set to 0.7.

Table 2 Job information for the example

Job ID	Submission time	Serial execution time (s)
Job 0	0	74,916
Job 1	68	89,760
Job 2	175	24
Job 3	3097	83,729
Job 4	8535	20,880

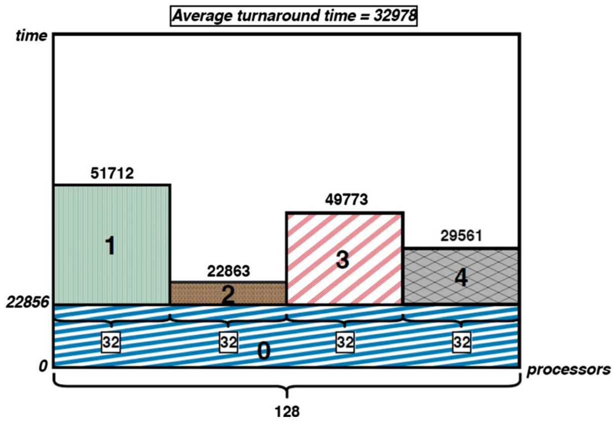

Fig. 6 Method in [7] (utilization = 4.069%)

Figure 6 shows the resultant schedule produced by the method in [7]. Since job 0 arrives first, it is allocated all 128 processors for execution immediately. Before job 0 finishes, all the other four jobs have arrived in the waiting queue. Therefore, on the finish event of job 0, the method in [7] adopts the *parallel policy* to start all the four jobs simultaneously. Figure 7 shows the schedule produced by our utilization-based processor allocation approach, where only three jobs, i.e., jobs 1, 2, and 3, are started right after the finish of job 0, and job 4 is scheduled to run after job 2 finishes. Compared to Fig. 6, it is obvious that the schedule in Fig. 7 not only allows the four jobs, i.e., 1, 2, 3, and 4, to use more processors for execution, but also increases resource utilization between time zero and the finish time of job 1. Therefore, the schedule produced by our approach achieves a shorter average job turnaround time than the schedule in Fig. 6 produced by the method in [7].

3.5 Bounded processor allocation

Although the proposed utilization-based processor allocation approach in average can achieve better performance than previous methods, e.g., [7], as illustrated in the previous section, we found that in some special scenarios it might lead to

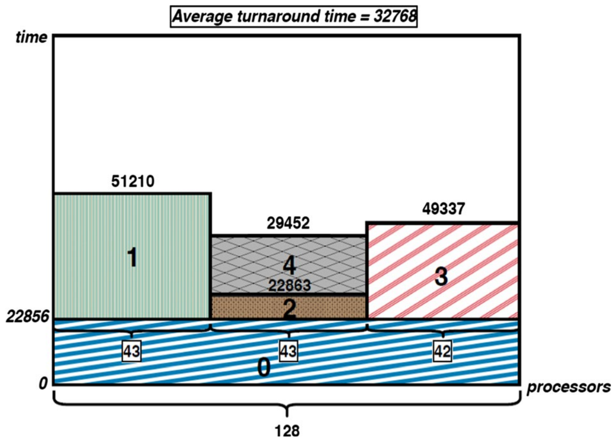

Fig. 7 Our approach (utilization = 4.109%)

inappropriate processor allocation and thus longer job turnaround time. The problem is because it would tend to allocate all free processors to a job if there is only such job in the waiting queue. Such processor allocation can maximize resource utilization and shorten the job's turnaround time. However, in some scenarios, after the job has been allocated all free processors and starts, there soon come new job submissions and these subsequent jobs would have to wait for available processors since all processors are in use, resulting in longer turnaround time for the subsequent jobs and thus longer average turnaround time for all jobs.

For example, in the five-job example in the previous section, job 0 arrives at time zero and there are no other jobs in the waiting queue at that moment, so job 0 is allocated all processors for execution. Therefore, subsequent jobs 1, 2, 3, 4 cannot start their execution until job 0 finishes and releases the occupied processors, as shown in Figs. 6 and 7. To avoid such scenarios, we add a bounded allocation mechanism to the utilization-based processor allocation approach, which sets an upper bound for the number of processors to be allocated to each job even if there are more free processors available. Figure 8 shows the schedule for the same five-job example resulted from the new processor allocation approach with bounded allocation. It is obvious that the bounded allocation mechanism effectively improves resource utilization and the average turnaround time of the five jobs, compared to Fig. 7. As shown in Fig. 8, the bounded allocation mechanism prevents job 0 from consuming up all processors at time zero, which allows job 1 to start earlier and thus lead to shorter average turnaround time of all the five jobs.

Theoretically, there would be a best value for the upper bound in the bounded processor allocation mechanism, which could result in the highest overall system performance. However, such a best value is in general workload dependent. Therefore, the proposed bounded processor allocation mechanism does not specify a fixed upper bound in it. In practical use, system administrators could try to find the best value by simulation-based analysis of the workload characteristics in the target systems.

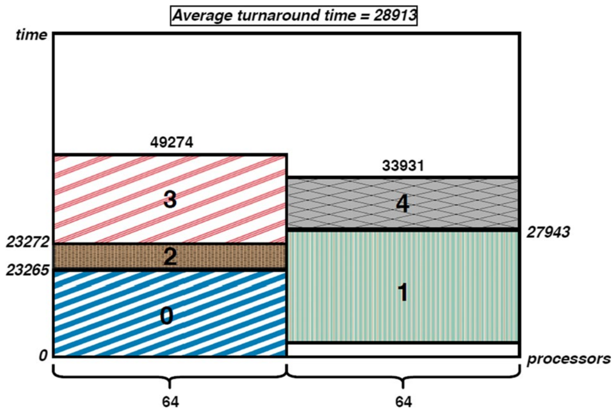

Fig. 8 Utilization-based processor allocation with bounded allocation (utilization = 4.27%)

3.6 Preemptive processor allocation

The bounded allocation mechanism presented in the previous section is a conservative approach since it imposes a strict limit on the maximum number of processors each job can get. This limitation could bring no benefits or even worse performance if there are no new arriving jobs until the running job finishes, because the unallocated processors are unnecessarily kept idle. This section presents a more aggressive approach, called preemptive processor allocation, to avoid the potential drawback of the bounded allocation mechanism.

In contrast to the bounded allocation mechanism, this approach allows a job to use all free processors for execution when there are no other jobs in the waiting queue. However, if later there are new arriving jobs before the job finishes its execution, the scheduler will make a decision between two possible processor allocation arrangements. This first is to keep the new jobs waiting until the running job finishes, and the second choice is to stop the running job and put it back to the waiting queue for reallocation and re-execution together with the new arriving jobs. The scheduler will determine which one of the two potential choices could lead to better average turnaround time of these jobs, and then conduct the corresponding processor allocation.

Figure 9 shows the final schedule produced by the utilization-based processor allocation approach augmented with preemptive allocation for the same five-job example in Table 2. In the final schedule, job 0 runs from time instant 3097 to 26,762 although it is submitted at time zero. Originally, job 0 starts its execution right upon its submission at time zero. However, when jobs 1, 2, and 3 arrive at different time instants later, the execution of job 0 is accordingly and restarted using less processors in order to allow subsequent jobs to run earlier, which is expected to achieve better overall performance of the five jobs. The part of schedule concerning jobs 0, 1, 2, and 3 in Fig. 9 is the rescheduling result after the arrival of job 3. That is why job 0 starts at time instant 3097 instead of its submission time.

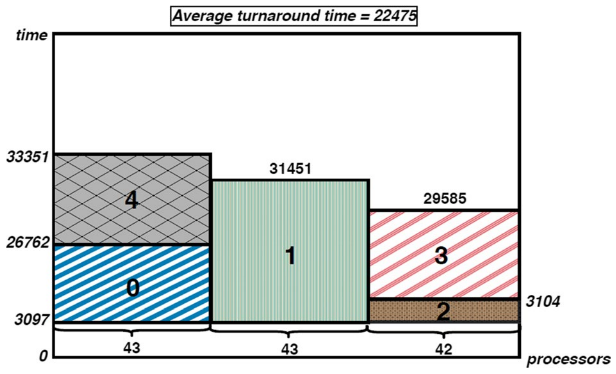

Fig. 9 Utilization-based processor allocation and preemptive allocation (utilization = 6.309%)

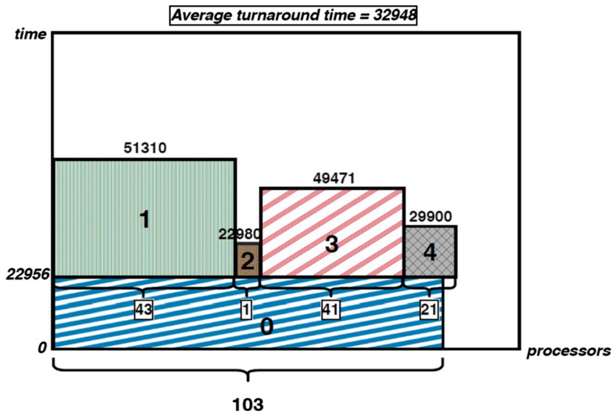

Fig. 10 HRF (utilization = 4.101%)

Compared to Fig. 7, Fig. 9 shows that preemptive processor allocation, like bounded processor allocation, can also overcome the drawback of the original utilization-based processor allocation approach presented in Algorithm 1, since job 0 would not consume up all free processors finally. Moreover, comparing Figs. 8 and 9 indicates that preemptive processor allocation can achieve better performance than bounded processor allocation, since job 2 could start earlier and resource utilization is also improved because of jobs' higher parallel efficiency resulting from fewer allocated processors.

Figures 10 and 11 show the resultant schedules by HRF [8] and EEMA [6], respectively, for the same five-job example in Table 2. However, since EEMA requires users to specify the numbers of requested processors upon job submission as described in Sect. 3.3, we present the requested numbers of processors for the five jobs in Table 3. Comparing Figs. 9, 10, and 11 indicates that our utilization-based processor allocation approach augmented with preemptive allocation outperforms HRF and EEMA significantly. As shown in Fig. 10, since HRF limits the maximum

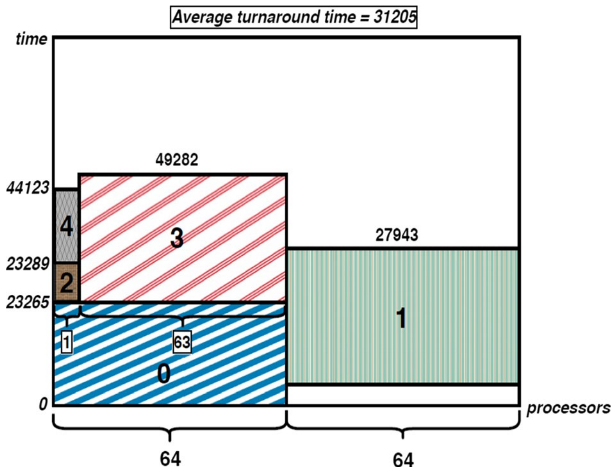

Fig. 11 EEMA (utilization = 4.269%)

Table 3 Job information of the example for HRF and EEMA

Job ID	Submission time	Serial execution time (s)	Number of requested processors
Job 0	0	74,916	64
Job 1	68	89,760	64
Job 2	175	24	1
Job 3	3097	83,729	64
Job 4	8535	20,880	16

number of processors that a job can use and allocates processors according to the total number of processors in a system instead of currently free processors, job 0 cannot uses all processors at time zero, resulting in 25 unnecessarily idle processors. Regarding EEMA, since it would allocate just the numbers of processors to jobs according to their original requests if free processors are enough, the processor allocation results might not lead to good resource utilization and average turnaround time as shown in Fig. 11 because of lower parallel efficiency of jobs, compared to our approach.

3.7 Dual-criteria processor allocation

In general, increasing resource utilization is an effective approach to improving performance for moldable job scheduling as shown in the literature and previous sections. However, we found that in some scenarios higher resource utilization does not necessarily lead to shorter average turnaround time. Therefore, we changed line 23 in Algorithm 1 into a dual-criteria while-loop, so that the iterative processor allocation process would continue only when both increasing resource utilization and

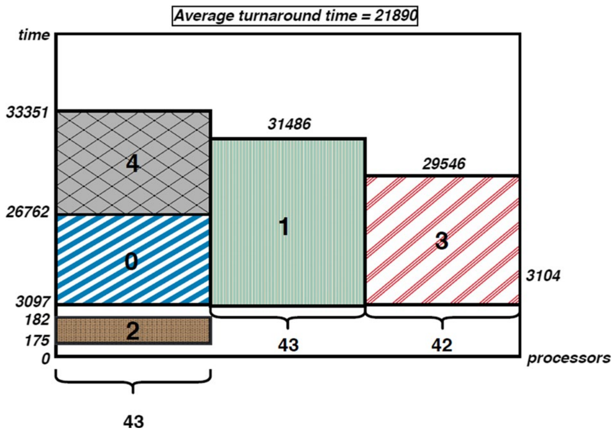

Fig. 12 Dual-criteria utilization-based processor allocation (utilization = 6.309%)

Table 4 Processor allocation results after each job arrives

Time instants	Original utilization-based approach		Dual-criteria utilization-based approach	
	Jobs	Number of processors	Jobs	Number of processors
Job 0 submission	Job 0	128	Job 0	128
Job 1 submission	Job 0	128	Job 0	64
	Job 1	128	Job 1	64
Job 2 submission	Job 0	64	Job 0	43
	Job 1	64	Job 1	42
	Job 2	64	Job 2	43
Job 3 submission	Job 0	43	Job 0	43
	Job 1	43	Job 1	42
	Job 2	42	Job 3	43
	Job 3	42		
Job 4 submission	Job 4	43	Job 4	43

decreasing average turnaround time are true. The resultant dual-criteria utilization-based processor allocation approach not only achieves shorter average turnaround time, but also saves scheduling time because of fewer iterations conducted.

Figure 12 is the schedule produced by the dual-criteria utilization-based processor allocation approach augmented with preemptive allocation for the same five-job example in Table 2. Table 4 compares the processor allocation results after each job arrives for the original utilization-based approach, i.e., Figure 9, and dual-criteria utilization-based approach, i.e., Figure 12, both augmented with preemptive allocation. On the arrival of job 2, it would be allocated 64 processors by the original utilization-based approach. On the other hand, it would be allocated only 43 processors by the dual-criteria utilization-based approach. Allocating fewer processors to job

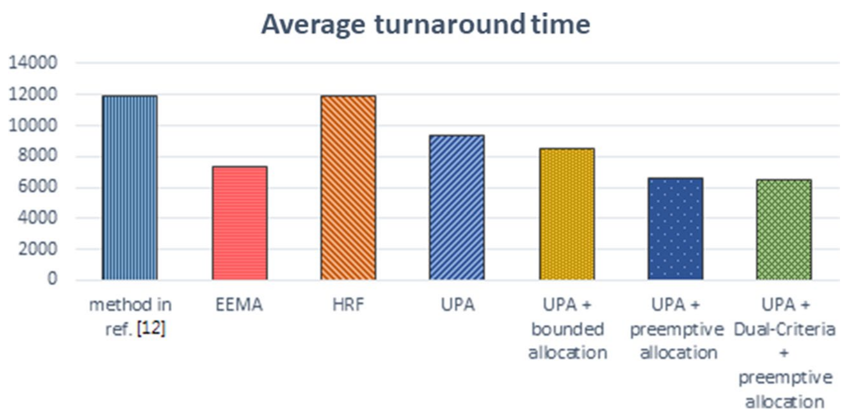

Fig. 13 Light workload with uniform α value

2 in the dual-criteria utilization-based approach allows it to start and finish earlier before the submission of job 3, and thus prevents it from being killed and rescheduled when job 3 arrives, which occurs in Fig. 9 where the original utilization-based approach is used. Moreover, allocating fewer processors to job 2 also leaves more processors for subsequent jobs, resulting in shorter average turnaround time. Comparing Figs. 9 and 12 indicates that the dual-criteria utilization-based approach has potential to achieve better performance than the original utilization-based approach.

4 Performance evaluation

This section presents the performance results of a series of simulation experiments evaluating our two-level utilization-based processor allocation approaches, compared to previous methods in the literature, including the method in [7], highest revenue first (HRF) [8], and extreme-ending moldable approach (EEMA) [6]. In the experiments, the parameters in HRF [8] were set to $\alpha = 0.8$ and threshold $= 0.9$ because such setting leads to the best performance in [8].

In the experiments, we simulated a 128-processor parallel system processing a set of moldable jobs derived from a public workload log on the parallel workload archive web site.[3] The workload log contains 73,496 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, our simulation experiments used 56,490 job records as the input workload. The upper bound in the bounded processor allocation mechanism was set to 64, i.e., half of the total processors in the system, as in the example of Fig. 8. The speedup of a job with different numbers of processors is calculated using Amdahl's law [55]

[3] https://www.cs.huji.ac.il/labs/parallel/workload/.

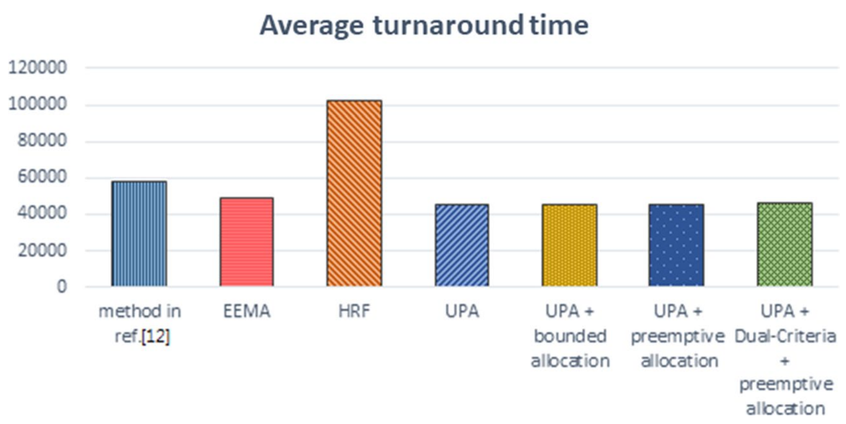

Fig. 14 Heavy workload with uniform α value

Figures 13 and 14 present the experimental results for light and heavy workloads, respectively. In both kinds of workloads, each parallel job has the same α value set to 0.7. To evaluate the effectiveness of each processor allocation mechanism presented in Sect. 3, in the simulation experiments our approach has four instances representing the original utilization-based processor allocation (UPA) approach in Algorithm 1, UPA augmented with bounded allocation, UPA augmented with preemptive allocation, and the final integrated dual-criteria UPA augmented with preemptive allocation, respectively. As shown in Fig. 13, both our UPA augmented with preemptive allocation method and dual-criteria UPA augmented with preemptive allocation method outperform all the three previous methods significantly when system workload is light. When confronting heavy workload as shown in Fig. 14, the performance results are quite different. Our approaches perform even better when system workload is heavy in that all the four UPA-based approaches outperform the three previous methods. However, under heavy workload, the approach that achieves the best performance is UPA instead of the dual-criteria UPA augmented with preemptive allocation method which performs the best in Fig. 13. This performance result indicates that resource utilization plays an even more important role in improving overall system performance as system workload gets higher.

Since in a real parallel computing environment, parallel jobs usually have different parallelism characteristics, we conducted another series of experiments, as shown in Figs. 15 and 16, where parallel jobs might have different ratios of parallelizable workload, i.e., α in Amdahl's law [55]. We randomly set the jobs' α values between 0.7 and 0.9 in the experiments. The experimental results in Figs. 15 and 16 show similar performance trends as in Figs. 13 and 14. Our utilization-based processor allocation approaches can achieve better performance in terms of average turnaround time than the three previous methods, and UPA leads to the best performance when system workload is high.

As described in Sect. 3, both bounded processor allocation and preemptive processor allocation were proposed to improve the scheduling decisions when there is only one job in the waiting queue. Since such situations are more likely

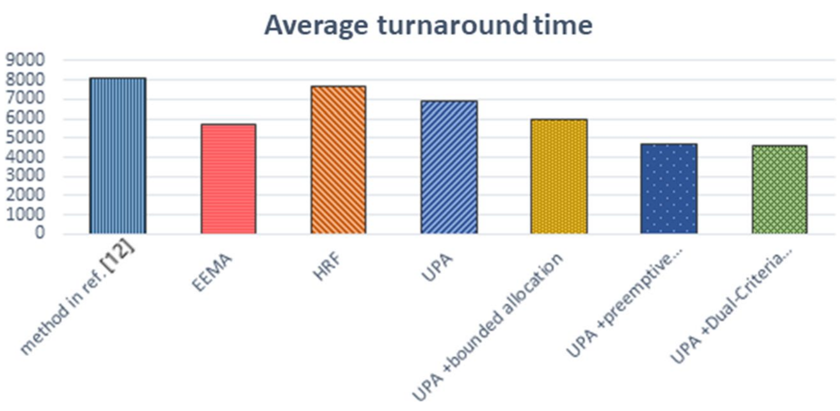

Fig. 15 Light workload with diverse α values

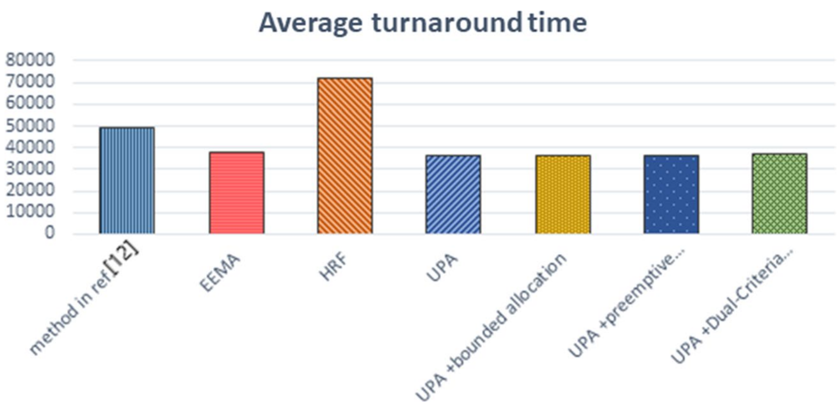

Fig. 16 Heavy workload with diverse α values

Table 5 Comparison of scheduling overhead (seconds)

	EEMA	HRF	Dual-criteria UPA augmented with preemptive allocation
Light workload	1.15	4.41	19.77
Heavy workload	1.51	34.62	2612.74

to happen under light workload, these two mechanisms led to significant performance improvement in the simulation experiments of light workload as shown in Figs. 13 and 15. On the other hand, it is rare, if not impossible, that the waiting queue contains only one single job under heavy workload. Therefore, bounded processor allocation and preemptive processor allocation could bring only negligible performance difference under heavy workload as shown in Figs. 14 and 16.

Our approach has a much more sophisticated decision process and thus would incur a significantly larger scheduling overhead than EEMA and HRF. Table 5 compares the scheduling overhead of our approach to EEMA and HRF by showing the required computation time for the three approaches to schedule the total 56,490 jobs in the experiments presented in Figs. 15 and 16. The data in Table 5 confirm the higher scheduling overhead of our approach. However, such overhead pays off in the significant reduction in average job turnaround time brought by our approach. For example, comparing Table 5 to Figs. 15 and 16, the overhead of scheduling the total 56,490 jobs by our approach pays off easily with the average reduction in a single job's turnaround time under light workload, compared to EEMA and HRF. Even under heavy workload, the scheduling overhead could pay off with the average turnaround time reduction of only two jobs.

5 Conclusions and future work

HPCaaS has become a promising trend for high-performance computing. Moldable job scheduling is an important research issue for realizing HPCaaS since it can not only lead to a much easier and convenient access model for HPC facilities, but also improve overall system performance. This paper presents our research work on developing a new two-level utilization-based processor allocation approach for moldable job scheduling. Experimental results show that our approach can achieve up to 23% performance improvement in terms of average turnaround time, e.g., Figure 15, compared to previous scheduling methods in the literature.

Among the methods evaluated in this paper, the improved moldable job scheduling for HPCaaS [7], HRF [8], and our approach are pure moldable job scheduling methods which do not require users to specify the number of processors to use upon job submission. This is an important feature for user convenience in realizing HPCaaS. On the other hand, EEMA [6] is an auxiliary moldable job scheduling approach, which treats all jobs as rigid initially, trying to allocate the requested numbers of processors to them, and takes advantage of a job's moldable property only when its desired number of processors is larger than the number of free processors. The experimental results show that only our approach can outperform EEMA [6] in all cases. Therefore, our approach is a promising progress toward HPCaaS since overall system performance is always the most important concern in high-performance computing.

In the future, several research directions are promising to further improve the performance of processor allocation for moldable job scheduling. The first is to develop new mechanisms replacing bounded allocation and preemptive allocation. Although the experimental results show that preemptive allocation outperforms bounded allocation, they both still result in some degree of resource inefficiency requiring further improvement. The second is about gaining insight into the interaction between resource utilization and job turnaround time. The experimental results for light and heavy workloads, respectively, show very different performance trends among the evaluated processor allocation methods. Further studies are also required for developing more efficient mechanisms controlling the iterative processor allocation process. Finally, despite achieving better schedules, the computation time of the proposed approach is significantly longer than previous methods, requiring further reduction.

In this paper, we focus on the processor allocation issue of moldable job scheduling. However, to realize the goal of HPCaaS, there are still other issues to be considered seriously, such as job priority or hardware faults which also have great influence on the quality of service (QoS). Fortunately, the processor allocation approaches proposed in this paper can blend well with the methods designed for other aspects of QoS in HPCaaS environments. For example, they can cooperate with necessary job sequencing methods or checkpoint and recovery techniques to provide job priority features or fault-tolerant computing environments.

References

1. Feitelson DG, Rudolph L, Schweigelshohn U, Sevcik K, Wong P (1997) Theory and practice in parallel job scheduling. Lect Notes Comput Sci 1291:1–34
2. Kessler C, Melot N, Eitschberger P, Keller J (2013) Crown scheduling: energy-efficient resource allocation, mapping and discrete frequency scaling for collections of malleable streaming tasks. In: Proceedings of the 23rd international workshop on power and timing modeling, optimization and simulation
3. Wu X, Loiseau P (2015). Algorithms for scheduling deadline-sensitive malleable tasks. In: Proceedings of the fifty-third annual Allerton conference
4. Asghar S, Aubanel E, Bremner D (2013) A dynamic moldable job scheduling based parallel SAT solver. Proceedings of the international conference on parallel processing, pp 110–119
5. AbdelBaky M, Parashar M, Kim H, JordanKirk EJ, Sachdeva V, Sexton J, Jamjoom H, Shae ZY, Pencheva G, Tavakoli R, Wheeler MF (2012) Enabling high performance computing as a service. IEEE Comput 45:72–80
6. Bagga S, Garg D, Arora A (2014) Moldable load scheduling using demand adjustable policies. Proceedings of the international conference on advances in computing communications and informatics
7. Huang KC, Huang TC, Tsai MJ, Chang HY, Tung YH (2013) Moldable job scheduling for HPC as a service with application speedup model and execution time information. J Converg Sect A Comput Commun 4(4):14–22
8. Wu S, Tuo Q, Jin H, Yan C, Weng Q (2015) HRF: a resource allocation scheme for moldable jobs. In: Proceedings of the 12th ACM international conference on computing frontiers
9. Kwon OH, Kim J, Hong SJ, Lee SG (1997) Real-time job scheduling in hypercube systems. In: Proceedings of international conference on parallel processing, p 166
10. Kwon OH, Chwa KY (1998) An algorithm for scheduling jobs in hypercube systems. IEEE Trans Parallel Distrib Syst 9(9):856–860
11. Ni LM, Turner SW, Cheng BHC (1995) Contention-free 2D-mesh cluster allocation in hypercubes. IEEE Trans Comput 44(8):1051–1055
12. 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
13. 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
14. Feitelson DG, Rudolph L (1995) Parallel job scheduling: issues and approaches. In: Proceedings of job scheduling strategies for parallel processing, pp 1–18
15. Lifka D (1995) The ANL/IBM SP scheduling system. In: Proceedings of the job scheduling strategies for parallel processing, pp 295–303
16. Skovira J, Chan W, Zhou H, Lifka D (1996) The EASY-LoadLeveler API project. In: Proceedings of the job scheduling strategies for parallel processing, pp 41–47
17. Feitelson, D.G., Weil, A.M. (1998). Utilization and predictability in scheduling the IBM SP2 with backfilling. In: Proceedings of the 12th Int'l parallel processing symposium, pp 542–546

18. Srinivasan S, Kettimuthu R, Subrarnani V, Sadayappan P (2002) Characterization of backfilling strategies for parallel job scheduling. In: Proceeding of the conference on parallel processing (ICPP), pp 514–522
19. Wong AKL, Goscinski AM (2007) Evaluating the EASY-backfill job scheduling of static workloads on clusters. In: Proceedings of the IEEE international conference in cluster computing, pp 64–73
20. Wong AK, Goscinski AM (2008) The impact of under-estimated length of jobs on EASY-backfill scheduling. In: Proceedings of the IEEE parallel, distributed and network-based processing
21. Feitelson DG (2005) Experimental analysis of the root causes of performance evaluation results: a backfilling case study. IEEE Trans Parallel Distrib Syst 16:175–182
22. Shmueli E, Feitelson DG (2005) Backfilling with lookahead to optimize the packing of parallel jobs. J Parallel Distrib Comput 65:1090–1107
23. Jackson DB, Snell Q, Clement MJ (2001) Core algorithms of the Maui scheduler. In: Proceedings of the job scheduling strategies for parallel processing
24. Feitelson DG, Rudolph L, Schwiegelshohn U (2005) Parallel job scheduling—a status report. In: Proceedings of the job scheduling strategies for parallel processing, pp 1–16
25. Tsafrir D, Etsion Y, Feitelson DG (2007) Backfilling using system-generated predictions rather than user runtime estimates. IEEE Trans Parallel Distrib Syst 18:789–803
26. Chiang SH, Vasupongayya S (2008) Design and potential performance of goal-oriented job scheduling policies for parallel computer workloads. IEEE Trans Parallel Distrib Syst 19:1642–1656
27. Stanzani S, Cóbe R, Fialho J, Iope R, Gomes M, Baruchi A, Amaral J (2019) Towards a strategy for performance prediction on heterogeneous architectures. In: Senger H et al (eds) High performance computing for computational science—VECPAR 2018. Lecture notes in computer science, vol 11333, pp 247–253
28. Shen C, Tong W, Choo KR et al (2018) Performance prediction of parallel computing models to analyze cloud-based big data applications. Cluster Comput 21:1439–1454
29. Martínez V, Serpa M, Dupros F, Padoin EL, Navaux P (2018). Performance prediction of acoustic wave numerical kernel on Intel Xeon Phi processor. In: Mocskos E, Nesmachnow S (eds) High performance computing. CARLA 2017. Communications in computer and information science, vol 796, pp 101–110
30. Caron E, Chouhan PK, Desprez F (2004) Deadline scheduling with priority for client-server systems on the grid. In: Proceedings of the fifth IEEE/ACM international workshop on grid computing
31. Le G, Xu K, Song J (2013) Dynamic resource provisioning and scheduling with deadline constraint in elastic cloud. In: Proceedings of the international conference on service science
32. Perret Q, Charlemagne G, Sotiriadis S, Bessis N (2013) A deadline scheduler for jobs in distributed systems. In: Proceedings of the 27th international conference on advanced information networking and applications workshops
33. Zhao W, Ramamritham K, Stankovic JA (1987) Scheduling tasks with resource requirements in hard real-time systems. IEEE Trans Softw Eng 13(5):564–577
34. Yoon H, Ryu M (2015) Guaranteeing end-to-end deadlines for AUTOSAR-based automotive software. Int J Automot Technol 16(4):635–644
35. Li J, Luo Z, Ferry D, Agrawal K, Gill C, Lu C (2014) Global EDF scheduling for parallel real-time tasks. Real Time Syst 51(4):395–439
36. Li J, Xiong M, Lee VCS, Shu L, Li G (2013) Workload efficient deadline and period assignment for maintaining temporal consistency under EDF. IEEE Trans Comput 62:1–14
37. Herrtwich RG (1990) An introduction to real-time scheduling. ICSI Technique report, TR-90-035
38. Pop F (2013) Scheduling of sporadic tasks with deadline constrains in cloud environments. In: Proceedings of the IEEE 27th international conference on advanced information networking and applications
39. Srinivasan S, Krishnamoorthy S, Sadayappan P (2003) A robust scheduling strategy for moldable scheduling of parallel jobs. In: Proceedings of the 5th IEEE international conference on cluster computing, pp 92–99
40. Srinivasan S, Subramani V, Kettimuthu R, Holenarsipur P, Sadayappan P (2002) Effective selection of partition sizes for moldable scheduling of parallel jobs. Lect Notes Comput Sci 2552:174–183
41. Cirne W, Berman F (2000) Adaptive selection of partition size for supercomputer requests. Lect Notes Comput Sci 1911:187–207
42. Cirne W, Berman F (2002) Using moldability to improve the performance of supercomputer jobs. J Parallel Distrib Comput 62:1571–1601

43. Sabin G, Lang M, Sadayappan P (2006) Moldable parallel job scheduling using job efficiency: an iterative approach. In: Proceedings of the job scheduling strategies for parallel processing, Saint Malo, France
44. Caniou Y, Charrier G, Desprez F (2011) Evaluation of reallocation heuristics for moldable tasks in computational grids. In: Proceedings of the 9th Australasian symposium on parallel and distributed computing
45. Huang KC (2006) Performance evaluation of adaptive processor allocation policies for moldable parallel batch jobs. In: Proceedings of the 3th workshop on grid technologies and applications
46. Beheshti Roui M, Shekofteh SK, Noori H et al (2020) Efficient scheduling of streams on GPGPUs. J Supercomput. https://doi.org/10.1007/s11227-020-03209-x
47. Chen Q, Oh J, Kim S et al (2019) Design of an adaptive GPU sharing and scheduling scheme in container-based cluster. Cluster Comput. https://doi.org/10.1007/s10586-019-02969-3
48. Choi HJ, Son DO, Kang SG et al (2013) An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. J Supercomput 65:886–902
49. Chen L, Ye D, Zhang G (2013). Online scheduling on a CPU–GPU cluster. In: Chan TH, Lau LC, Trevisan L (eds) Theory and applications of models of computation, TAMC 2013, Lecture notes in computer science, vol 7876, pp 1–9
50. Zhou Z, Li F, Zhu H et al (2019) An improved genetic algorithm using greedy strategy toward task scheduling optimization in cloud environments. Neural Comput Appl. https://doi.org/10.1007/s00521-019-04119-7
51. Mortazavi-Dehkordi M, Zamanifar K (2020) Efficient deadline-aware scheduling for the analysis of Big Data streams in public Cloud. Cluster Comput 23:241–263
52. Baskiyar S, Abdel-Kader R (2010) Energy aware DAG scheduling on heterogeneous systems. Cluster Comput 13:373–383
53. Mei J, Li K, Li K (2014) Energy-aware task scheduling in heterogeneous computing environments. Cluster Comput 17:537–550
54. Maurya AK, Modi K, Kumar V et al (2019) Energy-aware scheduling using slack reclamation for cluster systems. Cluster Comput. https://doi.org/10.1007/s10586-019-02965-7
55. Kleinrock L, Huang JH (1992) On parallel processing systems: Amdahl's law generalized and some results on optimal design. IEEE Trans Softw Eng 18(5):434–447
56. Downey AB (1997a) A model for speedup of parallel programs. UC Berkeley EECS Technical Report No. UCB/CSD-97-933
57. Downey AB (1997b) A parallel workload model and its implications for processor allocation. In: Proceedings of the 6th international symposium on high performance distributed computing
58. Lewis TG, Rewini HE (1992) Introduction to parallel computing. Prentice-Hall International, Upper Saddle River
59. Radulescu A, Nicolescu C, van Gemund AJC, Jonker PP (2001) CPR: mixed task and data parallel scheduling for distributed systems. In: Proceedings of the 15th international parallel and distributed processing symposium
60. Memeti S, Pllana S, Binotto A et al (2019) Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review. Computing 101:893–936

Affiliations

Ying-Jhih Wu[1] · Shuo-Ting Yu[1] · Kuan-Chou Lai[1] · Amit Chhabra[2] · Hsi-Ya Chang[3] · Kuo-Chan Huang[1]

Ying-Jhih Wu
4a090912@stust.edu.tw

Shuo-Ting Yu
b10752a371@gmail.com

Kuan-Chou Lai
kclai@mail.ntcu.edu.tw

Amit Chhabra
amit.cse@gndu.ac.in

Hsi-Ya Chang
9203117@narlabs.org.tw

[1] Department of Computer Science, National Taichung University of Education, Taichung, Taiwan

[2] Computer Engineering and Technology Department, Guru Nanak Dev University, Amritsar, India

[3] National Center for High-Performance Computing, National Applied Research Laboratories, Hsinchu, Taiwan