J. Parallel Distrib. Comput. 74 (2014) 2662–2672

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Proactive scheduling in distributed computing—A reinforcement learning approach

Zhao Tong [a], Zheng Xiao [a,*], Kenli Li [a], Keqin Li [a,b]

[a] College of Information Science and Engineering, Hunan University, Changsha, China
[b] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

HIGHLIGHTS

- Propose the concept and method of proactive scheduling.
- Formulate dynamic scheduling as a MDP problem.
- Develop an online scheduling algorithm based on reinforcement learning.
- Demonstrate our learning-based algorithm stable with lower average response time.

ARTICLE INFO

Article history:
Received 11 April 2013
Received in revised form
4 March 2014
Accepted 17 March 2014
Available online 21 March 2014

Keywords:
Distributed computing
Markov decision process
Queueing model
Reinforcement learning
Task scheduling

ABSTRACT

In distributed computing such as grid computing, online users submit their tasks anytime and anywhere to dynamic resources. Task arrival and execution processes are stochastic. How to adapt to the consequent uncertainties, as well as scheduling overhead and response time, are the main concern in dynamic scheduling. Based on the decision theory, scheduling is formulated as a Markov decision process (MDP). To address this problem, an approach from machine learning is used to learn task arrival and execution patterns online. The proposed algorithm can automatically acquire such knowledge without any aforehand modeling, and proactively allocate tasks on account of the forthcoming tasks and their execution dynamics. Under comparison with four classic algorithms such as Min–Min, Min–Max, Suffrage, and ECT, the proposed algorithm has much less scheduling overhead. The experiments over both synthetic and practical environments reveal that the proposed algorithm outperforms other algorithms in terms of the average response time. The smaller variance of average response time further validates the robustness of our algorithm.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

Nowadays almost all computers are connected to a local network or the Internet. The networked computers form a COW (cluster of workstations) or a distributed system. In distributed computing such as cluster computing and grid computing, online users submit their tasks anytime and anywhere to dynamic resources. Query processing in database management systems, particularly in a web-based database, is one that is often encountered in practice. Many queries arrive stochastically, and their execution plan has to be scheduled on the processing units with unstable performance. To determine when and where to dispose of these queries is referred to as task scheduling. Task scheduling is critical in exploiting the potential advantages of parallel and distributed systems [13].

In the above environments, task arrival and execution processes are stochastic. We take Google search service as an example to illustrate its impacts. A large number of users all over the world send their keyword queries to Google servers. Search engines use the MapReduce technique to divide a query into several classes of tasks, and then to map these tasks onto servers for execution. It involves three kinds of uncertainties.

- It is uncertain when and how many tasks will arrive because when and what kind of search query a user will initiate is unknown.
- It is uncertain how long a processing unit will take to execute a single task due to the dynamics of processors and networks. The performance of a server varies temporally. The network delay is hard to evaluate.

* Corresponding author.
E-mail addresses: tongzhao1985@yahoo.com.cn (Z. Tong), zxiao@hnu.edu.cn (Z. Xiao), lkl510@263.net (K. Li), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.jpdc.2014.03.007
0743-7315/© 2014 Elsevier Inc. All rights reserved.

- It is uncertain how many tasks are waiting in the queues for execution. As the system is not dedicated, users share all the processing units.

To adapt to those aforementioned uncertainties is a key to task scheduling. How to adapt to the consequent uncertainties, as well as scheduling overhead and response timeliness, are the main concern in this paper.

Scheduling algorithms mainly fall into two categories, i.e., static and dynamic scheduling. Static scheduling [15,18,4,32] originally emerged in parallel computing. A schedule for a parallel program is determined during compilation. To date, it means that scheduling happens before applications' running. In distributed computing, static scheduling fails because it is uncertain when and what kind of tasks may arrive. Instead, scheduling has to happen at runtime in our case, which is called dynamic scheduling. Because of online scheduling, the scheduling overhead and response time become important. But current dynamic scheduling algorithms [6,19,3,30] either have high scheduling overhead, resulting in a long queue for task admission [20], or unable to adapt to the uncertainties in task arrival and execution, leading to lagging response.

To address this problem, our preliminary work [31] tried to model task arrival and execution processes based on queueing theory as most scholars did [9,22,25], and proposed a semi-static scheduling algorithm. But we found that this approach gave impractical models on task arrival and execution. The performance gets worse under other models.

How to adapt to those uncertainties is still challenging. In this paper we use an approach from machine learning to learn task arrival and execution patterns online. In fact, scheduling is a decision problem. We formulate dynamic scheduling as a MDP (Markov decision process) problem. Because of those uncertainties, reinforcement learning is an effective method to solve an uncertain MDP problem. The proposed algorithm takes the previous allocations as training samples and adjusts its policy accordingly. It automatically acquires the knowledge of task arrival and execution without any aforehand modeling, and proactively allocate tasks on account of the forthcoming tasks and their execution dynamics. Under comparison with four classic algorithms such as Min–Min [6,3], Min–Max [6,3], Suffrage [19], and ECT [30], the proposed algorithm has much less scheduling overhead. The experiments over both synthetic and practical environments reveal that the proposed algorithm outperforms other algorithms in terms of the average response time. The smaller variance of average response time further validates the robustness of our algorithm.

This paper makes the following contributions.

- Proposing the concept and method of proactive scheduling with high adaptability and low scheduling overhead;
- Formulating dynamic scheduling as a MDP problem on account of the uncertainties of task arrival and execution;
- Developing an online scheduling algorithm based on reinforcement learning, which extremely enhances adaptability to the uncertainties in distributed computing;
- Demonstrating that our learning-based algorithm has lower scheduling overhead, effectively reduces the average response time, and is stable with lower variance.

The remainder of this paper is organized as follows. Section 2 reviews the related work on dynamic scheduling. Section 3 describes the motivation of minimizing the average response time and proactive scheduling. Section 4 defines the scheduling problem and formulates it by MDP. Section 5 gives the learning based scheduling algorithm. Section 6 presents a comparative study of our algorithms with the related work. Section 7 concludes this paper.

2. Related work

Task scheduling is a non-trivial problem and well known to be NP-hard even for non-preemptive scheduling of independent tasks [27]. As mentioned before, static scheduling makes decision before runtime. It acquires tasks, their dependency represented by a DAG (directed acyclic graph), resource performance as a prior knowledge. However, because of the uncertainties in distributed computing, such knowledge can only be acquired at runtime. Traditional static scheduling is not applicable in distributed computing.

There are two modes in dynamic scheduling for independent tasks. One is called the batch mode, which starts scheduling after a batch of tasks have arrived. Min–Min [6,3], Min–Max [6,3], and Suffrage [19] are three such typical algorithms. In the Min–Min approach, a scheduler calculates MCTs (minimum completion times) for tasks in the batch on resources. Then, it maps the task with the minimal MCT first. In contrast, the task with the maximal MCT has higher allocation priority in Min–Max. In Suffrage, it first maps tasks which suffer the most if not allocated right now. Usually, the suffrage value is the difference between its MCT and the second MCT. The size of a batch depends on the number of tasks or a fixed temporal interval. In the batch mode, tasks wait for scheduling until the size of a batch is reached. So tasks arriving earlier have to wait, which prolongs task response time. Besides, the time complexity of such algorithms is proportional to the size of batch times the number of processing units, because it needs to compute MCT of all pairs. For these two reasons, the batch mode algorithms result in high scheduling overhead. Hence, the online mode arises. Algorithms of this mode schedule a task immediately after its arrival. ECT is a typical algorithm which assigns tasks to the processing unit of the MCT. This mode nearly takes forthcoming tasks into account while the batch mode considers several tasks once making scheduling decision. So the batch mode has limited adaptability compared with the online mode.

Except for stochastic arrival of tasks, dynamic nature of resources is the other difficulty for task scheduling in distributed systems [12]. The following methods are usually employed to estimate task execution like MCT and adapt to the dynamic nature. (1) On-time information from the third party software component—For instance, GIS (Grid Information Service) in Grid is a software component, singular or distributed, that maintains information about people, software, services, and hardware that participate in a computational grid, and makes that information available upon request [1]. (2) Performance prediction—Most algorithms rely on performance estimates when conducting scheduling. Prediction is based on historical record [33] or workload modeling [7,10]. For example, most works predict resource performance under a queueing model [31,9,22,25]. (3) Rescheduling—Rescheduling changes previous schedule decisions based on a fresh resource status [24,29]. The method (1) needs extra communication cost and delay, the method (2) is hard to ensure providing high prediction accuracy with a simple algorithm, and the method (3) is available on condition that the infrastructure provides job migration.

Dynamic scheduling is shortsighted. In order to get a global optimization, scholars proposed new dynamic algorithms to adapt to the task arrival and execution processes. Grid schedulers like GridWay [17] and gLite WMS [16] only passively adapt to resource performance based on simple prediction models. AppLeS approach in [2] generates a schedule that not only considers predicted expected resource performance, but also the variation in that performance. Authors in [26] proposed a dynamic and self-adaptive task scheduling scheme based upon application-level and system-level performance prediction. Authors in [11] presented a resource planner system that reserves resources for the subsequent jobs. Most of these methods are passive and adjust schedules when performance varies. In addition, the impact of task arrival pattern is barely considered.

However, our learning based approach belongs to the online mode. It incurs low scheduling overhead. Furthermore, it

proactively makes scheduling decision based on the learned knowledge of both task arrival and execution patterns. It has better adaptability.

3. Motivation

In distributed computing, response time is believed to be a more important metric than makespan. Scheduling overhead and adaptability have considerable impact on performance of dynamic scheduling algorithms.

3.1. Scheduling overhead

Dynamic scheduling happens at runtime and scheduling time is a part of the task processing time, so that scheduling overhead has a negative influence.

In Ref. [20], the workload of some OSG (Open Science Grid) clusters is provided. The number of tasks waiting in the OSG cluster at University of California, San Diego could be as many as 37,000 tasks in the waiting queue at peak. The best-known scheduling algorithm for real-time tasks [5] takes more than 11 h to make admission control decisions on the 14,000 tasks that arrived in an hour. So the scheduling overhead could be substantial. A slow scheduling algorithm will result in a congestion at the scheduler and severely affect the response time of tasks.

3.2. Response time vs. makespan

In most previous works, makespan, which is the time for all tasks to be processed, is viewed as the major optimization goal. Unlike them, our work concerns task response time, which is the time for a single task to be processed [23]. Task response time is a metric from a user's perspective. It becomes important, as users are sometimes waiting online in distributed computing.

Let τ_i^j represent a task, where i denotes the task class and j is the sequence number. And P_k is the kth processing unit. A schedule *schl* is a function from tasks to processing units, i.e., $schl(\tau_i^j) = P_k$. The processing time of τ_i^j on P_k includes waiting and execution time right after its arrival, denoted by $PT_k(\tau_i^j)$. And P_k spends time CT_k to complete all the allocated tasks since the system starts running. Then makespan of *schl* is $\max_{P_k}\{CT_k\}$, while the average response time is

$$\frac{1}{|T|} \cdot \sum_{P_k} \sum_{\tau_i^j} PT_k(\tau_i^j)$$

where $|T|$ is the total number of tasks.

With shorter makespan, distributed systems are able to complete more tasks per time unit, resulting in higher throughput. Shorter response time means that users wait shorter for the feedback of their queries. Makespan indicates the steady-state speed of system or the capability of parallelism. In contrast, response time is the instantaneous speed for an individual user. Usually large response time may lead to a long makespan, but this is not a corollary. Sometimes, prolonging makespan a little bit may reduce the average processing time of individual tasks, lowering the response time. The following example tells that different schedules of the same makespan have different response time.

Suppose a simple distributed system has three processing units P_1, P_2, and P_3. There are two classes of tasks named t_1 and t_2. The time P_1, P_2, and P_3 take to execute t_1 is 6, 9, and 9 h respectively, while for t_2 they are 3, 5, and 6 h (refer to Table 1). Four tasks $\{\tau_1^1, \tau_2^2, \tau_2^3, \tau_2^4\}$ in total arrive in order. From the instant τ_1^1 arrives, others are arriving in 1, 2 and 5 h later. A task cannot be scheduled before its arrival. Fig. 1 shows two schedules. A column presents the tasks scheduled to one processing unit. The rounded rectangles represent the execution times of the tasks. In schedule (a) P_1 takes the longest time while P_3 takes the longest time in schedule (b). So they have the same minimum makespan 9. But the average response time of Fig. 1(a) is $(6 + 6 + 5 + 4)/4 = 5.25$. Task τ_2^4 has to wait for one hour before execution. If we move task τ_1^1 from P_1 to P_3, we make the average response time shorter, i.e., $(9 + 3 + 5 + 3)/4 = 5$ as Fig. 1(b) shows.

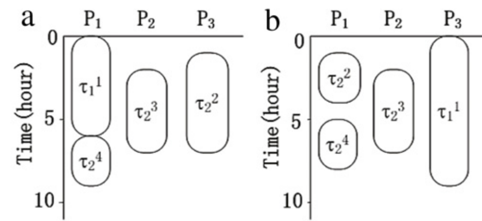


Fig. 1. Example: response time vs. makespan.

Table 1
Execution time matrix.

Task type	P_l	P_2	P_3
t_1	6	9	9
t_2	3	5	6

From this example, makespan and response time stand for different performance of scheduling.

3.3. Proactive scheduling

In the above example, if we knew a bunch of tasks of type t_2 would arrive after task τ_1^1, it is better to reserve the fastest resource for them. In the schedule (b) in Fig. 1, the fast resource P_1 is reserved for future tasks of type t_2. Sacrificing the task τ_1^1 by increasing its response time may profit the subsequent tasks. Consequently, proactive scheduling is such a concept that takes the possible scheduling of forthcoming tasks into account and then finds out a globally optimal scheme.

A proactive scheduling algorithm has higher adaptability. In order to schedule proactively, it is necessary to predict the patterns of task arrival and execution. Many scholars attempt to model them based on queueing theory [9,22,25]. Our preliminary work [31] assumes that tasks arrive in the form of a Poisson process, and proposes a semi-static scheduling algorithm for the same problem studied in this paper. But it fails under other forms of arrival and execution patterns. In Section 6, we will give a brief introduction to it and demonstrate its limitations compared with the algorithm in this paper. Through learning, the new algorithm can adapt to various kinds of task arrival and execution patterns.

4. Problem description and modeling

4.1. Problem description

According to the application and scenario of a network virtual laboratory of university, which is a project we are working on, the scheduling problem is described by Fig. 2. The system supports a number of experiments. Tasks are classified based on types of experiments. The tasks of two types have different sizes or data demands. There is a unique scheduler which takes charge of assigning tasks to processing units in a distributed system. Arrival tasks are pushed into the arrival queue at the scheduler. The scheduler retrieves one task each time from the arrival queue, and then allocates it to a processing unit and puts it into the local queue of the processing unit. Processing units execute tasks in its

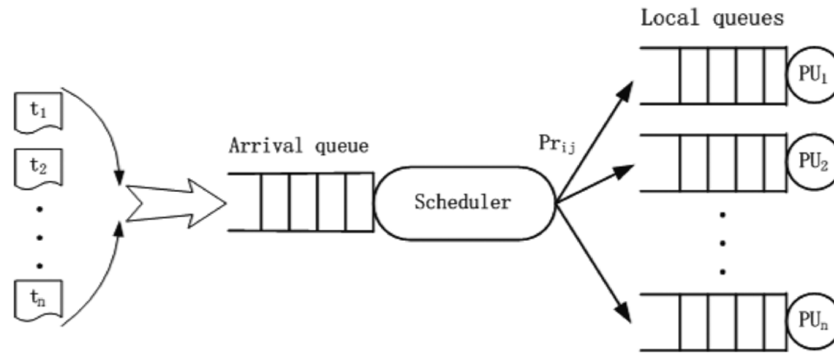


Fig. 2. The studied task scheduling problem.

Table 2
Notations in the queueing model.

Notation	Description
m	the number of types of tasks
n	the number of processing units
t_i	the ith type of tasks
PU_j	the jth processing unit
λ	the task arrival rate of the system
λ_j	the task arrival rate on PU_j
μ	the scheduling rate of the scheduler
μ_j	the service rate of PU_j
μ_{ij}	the service rate of PU_j for task type t_i
ρ_i	probability that currently arrived task is of type t_i
Pr_{ij}	probability of scheduling tasks of type t_i to PU_j

local queue in the way of FCFS (first come first served). Processing units are heterogeneous. They can commit part of the tasks or at different speeds.

In this scheduling problem above, tasks are independent and cannot be interrupted once it is being executed. So it is actually a centralized, independent, and non-preemptive scheduling problem.

In Fig. 2, queueing involves two levels. The length of a local queue partially depends on resource performance. The length of the arrival queue is influenced by scheduling overhead. If the scheduling overhead is high, tasks will stay longer in the arrival queue and the response time will increase. So the processing time of tasks includes scheduling time, execution time, and waiting time in two queues. Previous works usually consider one-level queueing and neglect the influence of scheduling overhead.

A summary of notations is given in Table 2.

The distributed system includes n processing units $\{PU_1, PU_2, \ldots, PU_n\}$ and handles m kinds of tasks. λ is the task arrival rate. μ is the scheduling rate. If $\mu \gg \lambda$, the arrival queue keeps empty and can be ignored. μ_j is the service rate of PU_j, depending on the performance of PU_j. Because different task types have different sizes or data demands, the service rate is different, denoted by μ_{ij}. The set $\{t_1, t_2, \ldots, t_m\}$ represents task types submitted by users. The currently arrived task is of type t_i with a probability of ρ_i. In Fig. 2, Pr_{ij} means the probability the scheduler assigns tasks of type t_i to PU_j.

4.2. MDP model

In distributed systems, tasks arrive at the scheduler stochastically. In order to schedule proactively, it is necessary to have complete information about the arrived tasks and the tasks to arrive, i.e., the knowledge of the arrival and execution processes. The allocation of current tasks not only depends on the allocation of tasks already arrived, but also influenced by tasks to arrive later. If tasks are assigned to processing units which have the shortest response time, the average response time of the whole distributed system may not be the shortest as shown by the motivation example in Section 3, because this greedy policy neglects the scheduling of forthcoming tasks. From this viewpoint, the scheduling problem for distributed computing appears to be a dynamic programming problem in which one cannot tell whether a single allocation is good or not until all tasks have been allocated. Scheduling performance depends on a series of allocations.

Each allocation is a single decision making. The current decision is related to the future decisions. Multiple such stages form a solution to the scheduling problem. Under this description, the Markov decision process (MDP) [14] can be used to model the scheduling problem. MDP is described by a quadruple (S, B, Γ, R). S is a set of states. B is a set of actions. Γ is a state transition function. A new state is reached after a certain action is conducted. R is the immediate reward function after taking an action. As to our scheduling problem, the MDP model is defined as follows.

- S is the set of task types, i.e., $S = \{t_1, t_2, \ldots, t_m\}$.
- B is the set of allocations, i.e., pairs of task type and processing unit, $B = \{(t_i, PU_j) \mid 1 \le i \le m, 1 \le j \le n\}$. The element $b_{ij} = (t_i, PU_j)$ in B means that a task of type t_i is assigned to the processing unit PU_j.
- $\Gamma : S \times B \to S$. If there is any dependence between tasks, the next state has something to do with the previous state. However, tasks are assumed to be independent in this paper, so the previous and successive states are irrelevant. The state transition only depends on the probability ρ_i (see Table 2). However, this does not cause trouble to the following algorithm. The knowledge of ρ_i can still be learned. For convenience of the following description, this function is kept as it was.
- $R : S \times B \to \Re$, where $\Re$ is the set of real numbers. $r \in \Re$ represents the immediate reward, for example, when taking the allocation $b_{ij} \in B$ for a task of type $t_i \in S$. In our case, R is a function of the processing time of a single task.

The state transition function Γ implies the task arrival process and the probability distribution of task types. The immediate reward function R implies the dynamic performance and the execution time of individual tasks. The capability of processing units determine possible actions.

Under MDP, we ought to evaluate schedules of tasks. The key to evaluation is to describe the influence of former and later allocations. Discounted accumulative reward is an objective function often used in MDP. Starting from the θth allocation, it is defined below:

$$\Psi(\theta) = r_\theta + \gamma r_{\theta+1} + \gamma^2 r_{\theta+2} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{\theta+i}, \quad \theta \ge 1 \qquad (1)$$

where γ is the discounted factor.

In Eq. (1), the reward at θth allocation depends not only on the immediate reward r_θ, but also on the immediate reward of the

forthcoming allocations. In other words, the allocation with the minimum response time currently may lead to a lagging response of forthcoming tasks. It is believed that the influence on future allocations decays. The later tasks arrive, the less influence they have on the current allocation. So a discount is added to each item. This function embodies proactive scheduling which takes future task allocations into account.

The discounted accumulative reward Ψ cannot be computed until all tasks have been assigned by a series of actions. It relies on the state transition function Γ and the immediate reward function R, which further rely on task arrival and task execution. In the next section an adaptive algorithm is proposed to find out a scheduling policy which maximizes Ψ. It uses historical online allocations to gradually adapt to task arrival and execution pattern without any priori knowledge.

5. An online learning-based adaptive scheduling algorithm

How to automatically adapt to the task arrival and execution processes is our main concern in this section. The following algorithm is about to solve these problems by learning the patterns of task arrival and execution online. The extra communication happens when collecting the processing time of a single task.

5.1. Algorithm description

Because of the uncertainties stated in Section 1, the next state $\Gamma(t_i, b_{ij})$ is uncertain and follows the probability distribution $PD_S = (\rho_1, \rho_2, \ldots, \rho_m)$ over set S. In addition, the immediate reward $R(t_i, b_{ij})$ is also stochastic, because of the varying load and network delay. The same allocation may have different discounted accumulative reward. Hence, we try to maximize the expected discounted accumulative reward:

$$E[\Psi(\theta)] = E\left[\sum_{i=0}^{\infty} \gamma^i r_{\theta+i}\right]. \tag{2}$$

In order to maximize Eq. (2), we need to get the discounted accumulative reward of each action, which is denoted by Q-function $Q(t_i, b_{ij})$ in Eq. (3).

$$\begin{aligned} Q(t_i, b_{ij}) &= E[R(t_i, b_{ij}) + \gamma V^{\pi}(\Gamma(t_i, b_{ij}))] \\ &= E[R(t_i, b_{ij})] + \gamma \sum_{t' \in S} P(t' \mid t_i, b_{ij}) V^{\pi}(t'), \end{aligned} \tag{3}$$

where $t' = \Gamma(t_i, b_{ij})$ and $P(t' \mid t_i, b_{ij}) \in PD_S$ is state transition probability. $V^{\pi}(t_i)$ is V-function, which denotes the expected reward under certain state when following the optimal scheduling scheme π:

$$V^{\pi}(t_i) = \max_j Q(t_i, b_{ij}). \tag{4}$$

The optimal scheduling scheme is the allocation which has the maximal Q:

$$\pi(t_i) = \arg\max_j Q(t_i, b_{ij}). \tag{5}$$

In the above equations, the scheduler considers two factors when making decisions. One is the load on each processing unit, i.e., the immediate reward of individual allocations. The other is the future allocations, i.e., the V-function.

Eqs. (3)–(5) define a typical dynamic programming problem. Though, to solve it we need to know $P(t' \mid t_i, b_{ij})$ and $R(t_i, b_{ij})$, which are related to task arrival and execution processes. If knowing task arrival and task execution, the dynamic programming method can be used to compute the optimal scheme π, making the very decisions that maximize the expected reward of Eq. (2). But the knowledge of task arrival and task execution is hard to obtain. To build their models is one solution to get such knowledge, but it is a nontrivial job. Q-learning provides a method which implicitly learns and gradually adapts to the uncertainties.

In this section, we attempt to use Q-learning [14], one method of model-free reinforcement learning (RL), to solve the MDP problem. It is an online and unsupervised machine learning method. It works by learning some value functions that give the expected reward of taking each action in any state. It does not need any training samples except the historical allocations. There is no need to model task arrival and task execution.

According to Q-learning, two value functions given in Eq. (6) are defined. $\hat{Q}(t_i, b_{ij})$ denotes the estimated expected reward of certain allocation, while $\hat{V}(t_i)$ denotes the estimated expected reward for some task type.

Eq. (6) gives the lth learning rule:

$$\begin{aligned} \hat{Q}_l(t_i, b_{ij}) \;&\leftarrow\; (1-\alpha_l)\hat{Q}_{l-1}(t_i, b_{ij}) \\ &\quad + \alpha_l[R(t_i, b_{ij}) + \gamma \hat{V}_{l-1}(\Gamma(t_i, b_{ij}))], \\ \hat{V}_l(t_i) &\leftarrow \max_j \hat{Q}_l(t_i, b_{ij}), \end{aligned} \tag{6}$$

where l denotes the iteration, and α_l $(0 \le \alpha_l \le 1)$ is the learning rate for the lth iteration. The immediate reward $R(t_i, b_{ij})$ is the reciprocal of the interval from arrival of a task until its completion at PU_j.

Q-learning is an error and attempt approach. This approach works correctly under the condition that every state is visited sufficiently. The ε-greedy exploration policy is taken. The exploration probability p_e is preset. The possible allocations are selected randomly by p_e; otherwise, the allocation satisfying Eq. (5) is selected. This exploration policy also functions to avoid falling in a local optimum.

Convergence is a necessary attribute of learning algorithm. According to Ref. [28], the Q-function $\hat{Q}$ in Eq. (6) can converge to Q in Eq. (3) if the learning rate is defined by Eq. (7):

$$\alpha_l = \frac{1}{1 + visits_l(t_i, b_{ij})}, \tag{7}$$

where $visits_l(t_i, b_{ij})$ is the number of visits to pair (t_i, b_{ij}) during l iterations.

A complete description of our reinforcement learning algorithm is given below.

The learning-based algorithm does not need any information like the probability ρ_i that a task is of certain type, λ of the task arrival process, and μ_{ij} of task execution.

Algorithm 1 is an online learning algorithm. It learns the allocation scheme after a task is allocated and executed. This leads to a drawback: tasks used for training before its convergence may be allocated to inappropriate processing units and users have to pay for the worse performance.

5.2. Performance analysis and deployment

(A) Communication cost

The immediate reward $R(t_i, b_{ij})$ is collected from a processing unit when a task is completed. It is just a numerical value. There is no need to get the loads on all processing units each time a task is scheduled. Hence, this does not bring any burden on communication with such a little data transmission.

(B) Scheduling overhead

In Algorithm 1, the learning and updating steps in each loop only take a constant number of algebraic calculations. The scheduling overhead mainly exists in the action selection of Step (2). The time complexity of Algorithm 1 is $O(n)$ for a single task.

Algorithm 1 Reinforcement learning based algorithm

Input: task type of the arrived tasks
Output: $\pi(t_i)$
(1) Initialization
  for any $t_i \in S$, any $b_{ij} \in B$ **do**
    $Q_0(t_i, b_{ij}) = 0$;
    $visits_1(t_i, b_{ij}) = 0$
    $\alpha_1(t_i) = 1$
  end for
(2) Action selection for task t_i
  if Explore with probability p_e **then**
    Select allocation b_{ij} randomly by uniform distribution
  else
    Select allocation b_{ij} by the currently learned scheme $\pi(t_i) = \arg\max_j Q(t_i, b_{ij})$
  end if
(3) Learning
  Update(Q):
  $\hat{Q}_l(t_i, b_{ij}) = (1 - \alpha_l)\hat{Q}_{l-1}(t_i, b_{ij}) + \alpha_l[R(t_i, b_{ij}) + \gamma \hat{V}_{l-1}(\Gamma(t_i, b_{ij}))]$
  Update(V):
  $\hat{V}_l(t_i) = \max_j \hat{Q}_l(t_i, b_{ij})$
(4) Updating
  $visits_{l+1}(t_i, b_{ij}) = visits_l(t_i, b_{ij}) + 1$
  $\alpha_{l+1} = \frac{1}{1+visits_{l+1}(t_i, b_{ij})}$
  goto (2)

In fact, this can reach $O(\log n)$ by increasing space complexity, e.g., the MaxHeap. The heap is maintained in Step (4).

(C) Deployment

When the task arrival rate is not too high, the scheduler is capable of updating and learning while scheduling. However, if tasks arrive in a large batch, synchronizing updating and learning with scheduling becomes a bottleneck. There are two ways of optimization during deployment.

- Delaying updating and learning—If quite busy, the scheduler can update and learn later, though the negative influence is slower convergence.
- Separating learner from scheduler—The scheduler is only responsible for task assignment and another dedicated server is used to update and learn.

After convergence, the learner can stop learning for a while. Then the scheduler maps tasks to processing units of maximal Q-value. If there exists a Maxheap, this map can be done in $O(1)$ time. The learner resumes learning in a periodic time, in case of any changes in task arrival and execution. If the stopping probability is $\eta \in (0, 1]$, scheduling complexity is $O(\eta \cdot 1 + (1 - \eta) \cdot \log n)$. By intermittent learning, the scheduling overhead is approximately $O(1)$ and scheduling efficiency is further improved.

6. Experiments and results

In this section, we will validate convergence, robustness, and performance of the online learning-based scheduling algorithm.

The parameters related to learning are set as follows in our experiments. They are an empirical setting.

- The exploring probability is $p_e = 0.2$, in which an action is selected at random. Exploration scheme is used to avoid the local optimum and visit all states sufficiently. The larger, the slower it converges.
- The discount factor is $\gamma = 0.9$, which is the coefficient in accumulative discounted reward. The closer to 0, the more weight over the immediate reward; otherwise, the more weight over the future task allocations.

Table 3
Service rate matrix (tasks/hour).

Unit	t_1	t_2	t_3	t_4	t_5
P_1	60	50	40	45	55
P_2	30	35	30	40	40
P_3	15	55	50	35	45

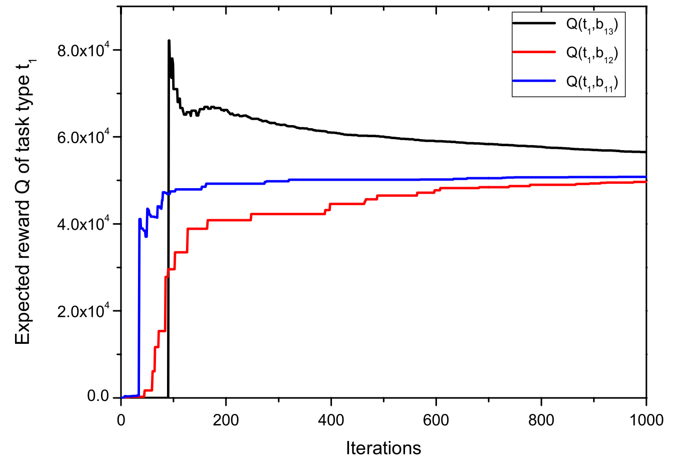


Fig. 3. Convergence of learning-based scheduling.

All of our experiments are simulated on a single PC. In the future work, we intend to transplant them onto a real cluster based on the supercomputing center to be built in our university.

6.1. Convergence

We simulate a distributed system with three processing units and a scheduler. Five types of tasks $t_1, t_2, \ldots, t_5$ arrive at the scheduler stochastically with probabilities 0.4, 0.2, 0.1, 0.1, 0.2. Tasks arrive as a Poisson flow, with execution time following an exponential distribution. Simply, we assume P_1, P_2, P_3 can execute 50, 35 and 40 tasks per hour on average. To embody different execution time for each type, we assign different service rates to each type. The service rate matrix is given in Table 3.

Convergence is important for learning-based algorithms. Our algorithm learns the flow of 1000 tasks with the arrival rate $\lambda = 80$ tasks per hour. Fig. 3 shows result of the convergence experiment. The curves of different colors denote the different expected rewards $Q(t_1, b_{1j}), j = 1, 2, 3$, in allocating tasks of type t_1 to the three processing units respectively. The oscillation tends to shrink as the learning goes on. After learning about 500 times, the Q-function converges. When focusing on the first hundred of iterations, we observe that most tasks are assigned to P_1. Later, a jam happens at P_1. So the tasks begin to be assigned to other processing units.

6.2. Robustness

Our previous work [31] represents a class of scheduling approaches which achieve adaptability by explicitly modeling task arrival and execution processes. To demonstrate robustness of learning approach, we use the work [31] as a delegate to compare with the learning-based algorithm.

The approach in [31] models task arrival as a Poisson flow and execution as an exponential distribution. So M/M/1 model in queueing theory is used to formulate the scheduling problem. As per the additive property of Poisson distribution, the aggregated

flow is still a Poisson flow. So the arrival process of each local queue is also a Poisson flow. The arrival rate λ_j on PU_j can be defined as:

$$\lambda_j = \sum_{i=1}^{m} \lambda_{ij} = \sum_{i=1}^{m} \rho_{ij}\lambda = \sum_{i=1}^{m} Pr_{ij}\rho_i\lambda, \tag{8}$$

where λ_{ij} is the rate that tasks of type t_i arrive at PU_j, ρ_{ij} is the probability that a task in PU_j's local queue is of type t_i, and ρ_i is the probability that a task in the arrival queue is of type t_i.

The service rate μ_j of PU_j can be computed as the weighted average of μ_{ij}:

$$\mu_j = \sum_{i=1}^{m} \frac{\lambda_{ij}}{\lambda_j}\mu_{ij} \tag{9}$$

where μ_{ij} denotes the service rate for task type t_i on processing unit PU_j.

If the scheduler can allocate tasks fast enough, i.e., $\mu \gg \lambda$, the arrival queue is empty. The task processing time depends only on the waiting time and the staying time in the local queues. Based on the results from queueing theory [8], we have the expected response time below.

$$\begin{aligned} RT_{exp} &= \sum_j \frac{\lambda_j}{\lambda} \cdot \frac{1}{\mu_j - \lambda_j} \\ &= \sum_j \frac{\lambda_j}{\lambda} \cdot \frac{1}{\sum_i \frac{\lambda_{ij}}{\lambda_j}\mu_{ij} - \sum_i Pr_{ij}\rho_i\lambda}. \end{aligned} \tag{10}$$

Scheduling aims to find out the allocation probability Pr^*_{ij} that minimize the expected response time. As a result, it is defined as a non-linear programming problem.

$$\begin{aligned} &\min_{Pr_{ij}} RT_{avg} = \min_{Pr_{ij}} \sum_j \frac{\lambda_j}{\lambda} \cdot \frac{1}{\sum_i \frac{\lambda_{ij}}{\lambda_j}\mu_{ij} - \sum_i Pr_{ij}\rho_i\lambda}, \\ &\text{s.t.} \quad 0 \le Pr_{ij} \le 1, \\ &\qquad \sum_j Pr_{ij} = 1. \end{aligned} \tag{11}$$

In the experiments, we use function "*fmincon*" in MATLAB toolbox to solve it. In this approach, the allocation scheme Pr^*_{ij} is computed before scheduling, and then the scheduler only needs to follow the scheme. Hence, the scheduling overhead is very small. If the distributed system is not very large and tasks arrive gently, the constraint $\mu \gg \lambda$ is satisfied. Algorithm 2 gives its pseudo-code.

Algorithm 2 Non-linear programming based algorithm

Input: $m, n, \lambda, \rho_i, \mu_{ij}$

Output: Pr_{ij}

1: Initialize $Pr_{ij} = \frac{1}{n}, \forall i, j$
2: Define objective function RT_{exp} by Eqs. (8)–(11)
3: $Pr_{ij} = fmincon(RT_{exp})$ %using subspace trust region method for non-programming problem

The time complexity of the subspace trust region method used in *fmincon* is $O((mn)^3 \cdot N)$, where N is the number of iterations. Compared with Algorithm 1, Algorithm 2 needs parameters of task arrival and execution. It is hard to estimate these parameters. In addition, Algorithm 2 assumes task arrival of Poisson distribution and task execution time of exponential distribution. They are rigid limitations.

In the rest, we will examine the performance under different forms of task arrival. The arrival interval between successive tasks follows such four distributions as exponential, constant, uniform, and chi-square. Their expectations keep the same as reciprocal of the arrival rate, i.e., $1/\lambda$. Wherein the constant one means that the arrival interval between two tasks is a constant of $1/\lambda$. Keep the other settings as in the convergence experiments.

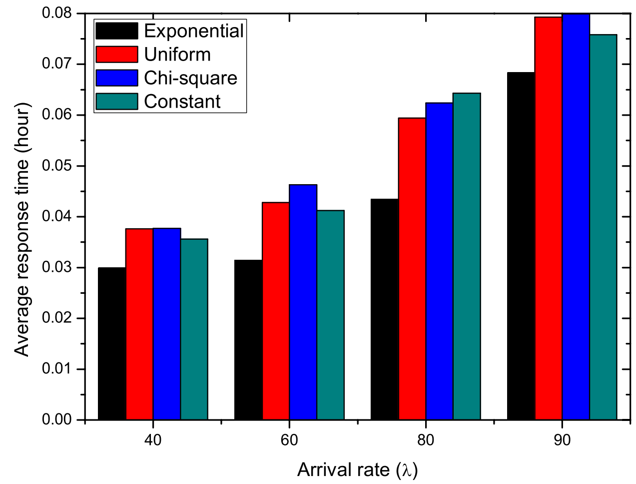


Fig. 4. Robustness in task arrival for Algorithm 2.

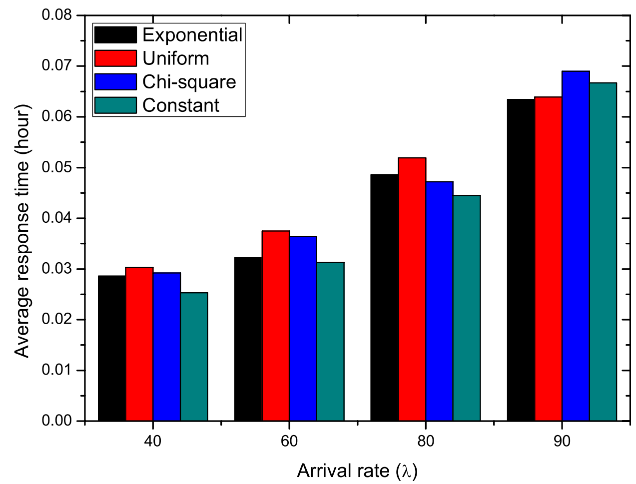


Fig. 5. Robustness in task arrival for Algorithm 1.

Fig. 4 shows the average response time over 100 tasks under different arrival rates. The average response time of exponential arrival is the lowest, because it is completely in accordance with the assumption. However, the other distributions result in longer response time.

In Fig. 5, the robustness of the learning based algorithm is analyzed. The average response time is measured over 100 tasks after learning converges. As the task arrival rate increases, the average response time increases too. In contrast to Fig. 4, the average response time of the learning-based algorithm is relatively stable. When the arrival rate is 80, they are about 0.0481, 0.0518, 0.0478, and 0.0472 respectively under the four distributions. But the performance of Algorithm 2 decreases when the distribution is not exponential. In summary, the learning algorithm is more capable of adapting to the task arrival process.

In the following, we investigate robustness when the estimation on execution time has errors. The exact service rates of three processing units are provided in Table 3. Two experiments with arrival rate 60 and 80 are committed under Poisson arrival process. If some errors in estimation happen, Figs. 6 and 7 show the changes in average response time. The scales on x-axis stand for the correct,

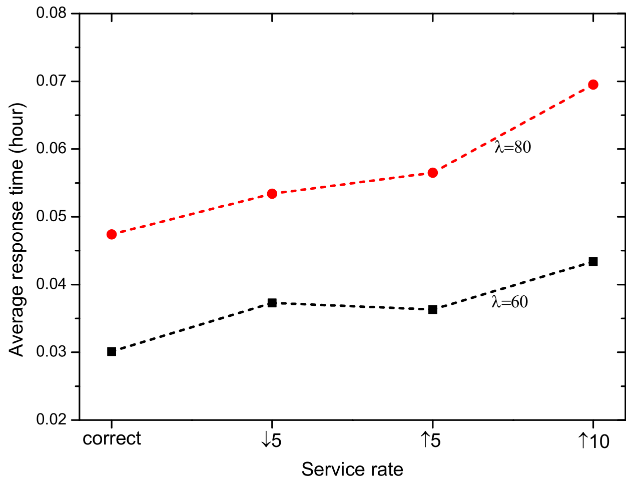


Fig. 6. Robustness in task execution for Algorithm 2.

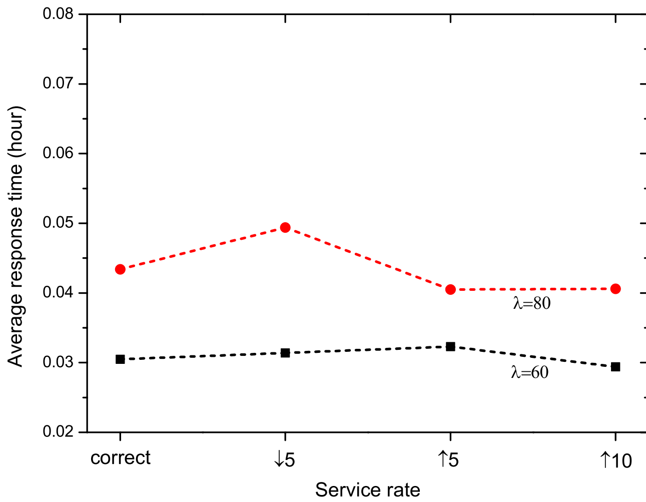


Fig. 7. Robustness in task execution for Algorithm 1.

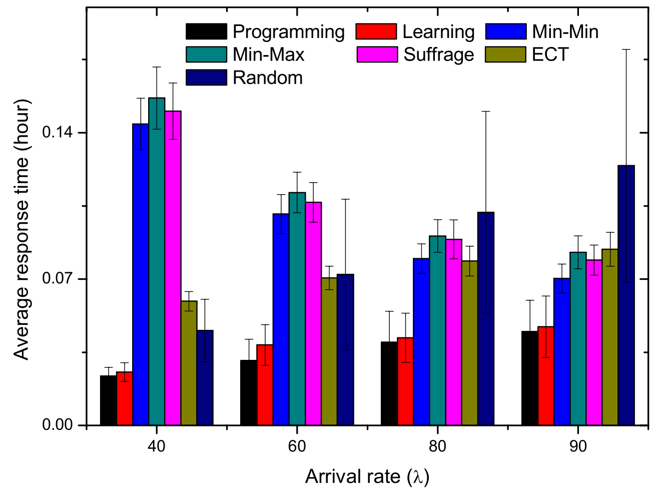


Fig. 8. Performance comparison under Poisson arrival.

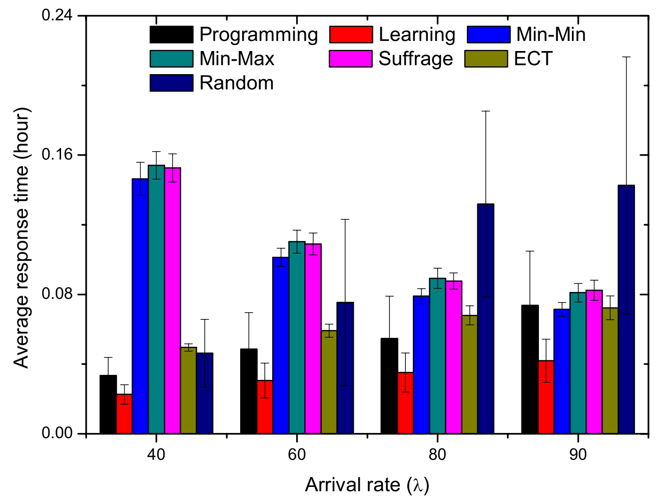


Fig. 9. Performance comparison under random arrival.

5-underestimated, 5-overestimated, and 10-overestimated service rates. n-overestimated or underestimated mean that the estimated service rates for each type are higher or lower than the rates in Table 3 by n. The average response time in Fig. 6 increases a lot, while there are some fluctuation in Fig. 7 because Algorithm 1 needs no estimation.

In conclusion, learning-based algorithm is more robust in task arrival and execution variation.

6.3. Performance analysis

The following experiments compare our algorithms with some classical scheduling ones. In this group of experiments, the performance of our algorithms, Min–Min, Min–Max, Suffrage, and a naive ECT based online algorithm is analyzed. These algorithms are designed for optimizing Makespan, and use completion time for heuristic rules. But completion time of each task only considers waiting time in the local queue and execution time, not including waiting time in the arrival queue and scheduling time. So for a fair comparison, we use processing time, which includes all the staying time in both queues, to replace completion time.

Average response time is the main index for evaluation. However, in a dynamic environment, the stochastic property causes an oscillated or unstable average response time. Different task flows may have different performance. It is expected that the scheduling algorithm can generate an "always-good" scheme under any task arrival process. In other words, the variance of average response time ought to be small for a stable scheduling algorithm. Consequently our performance analysis takes the average response time into account as well as its variance.

In the experiments, the average response time is measured over 100 tasks after learning converges. To simulate stochastic environment, we generate 50 task flows by random number generator which share the same statistical parameters like expectation. For example, flows of Poisson arrival have the same arrival rate. The variance is computed over these flows. The batch size for Min–Min, Min–Max, and Suffrage is 10 tasks. The performance is first investigated on a small-scale distributed system, with the setting similar to the above experiment, and then on a large-scale distributed system.

6.3.1. Small-scale distributed system

Fig. 8 compares the performance of our algorithms with other four classic algorithms under the condition that tasks arrive in Poisson distribution. Fig. 9 shows the performance when task arrival follows a uniform distribution. In these figures, we also used the random policy as a comparison baseline, in which tasks are assigned to all processing units with equal probability.

First, our algorithm outperforms the others in terms of the average response time. Because the assumptions are satisfied in

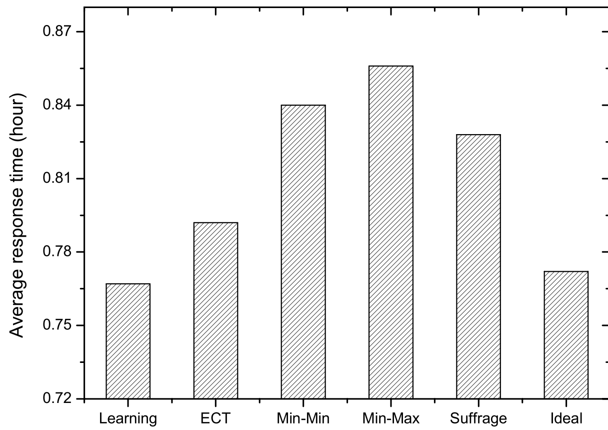


Fig. 10. Performance comparison of LPC cluster applications (140 CPUs).

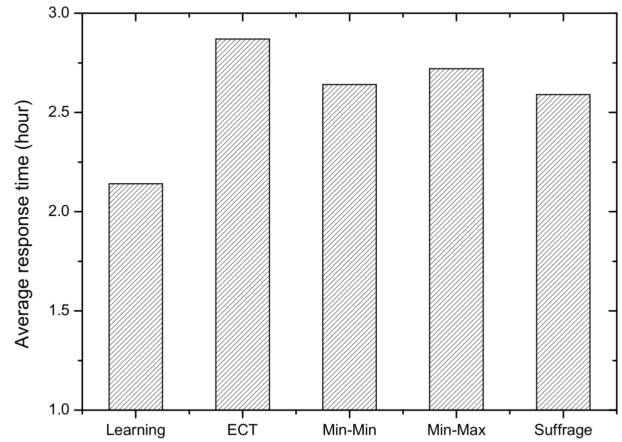


Fig. 11. Performance comparison of LPC cluster applications (15 CPUs).

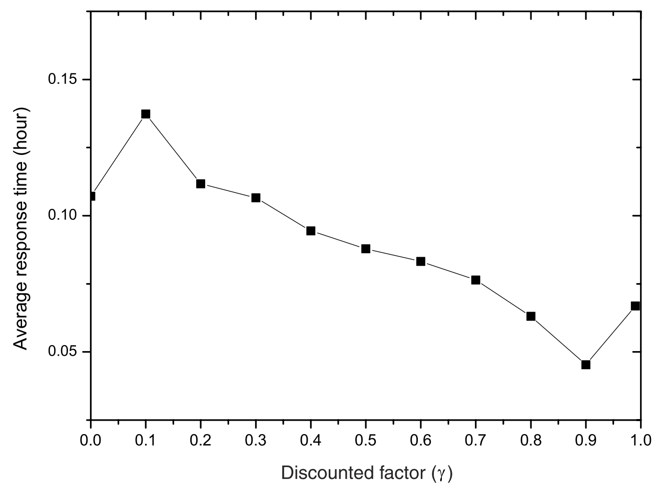


Fig. 12. Average response time under varied discounted factor.

Fig. 8, the non-linear programming based algorithm generates the better allocation probabilities and has the shortest response time. The performance of our learning-based algorithm is close to it. However, in Fig. 9, the performance of the non-linear programming-based algorithm drops a lot and is worse than the learning-based algorithm.

Second, the average response time of our algorithms and ECT increases as the task arrival rate increases. However, the three batch-mode algorithms have the opposite trend. The reason is that batch-mode scheduling spends time in waiting for a batch of tasks to arrive. It takes more time when tasks arrive sparsely, so that the response time is longer for lower arrival rate.

Finally, different from traditional performance comparison, both figures show the variance of average response time. Our algorithm has greater variance than other algorithms. Perhaps this has something to do with the probabilistic scheduling and the random exploration policy in the learning-based algorithm. The other algorithms may not be as good as they look, because they need to correctly predict task execution time before scheduling. In practice, there often exist errors in prediction. In this case, the variance may get greater.

6.3.2. Large-scale distributed system

In the section, we simulate a real cluster, i.e., LPC (Laboratoire de Physique Corpusculaire, CNRS-IN2P3, Clermont-Ferrand, France) [21]. There are mainly two types of tasks, i.e., Dteam jobs and Biomed jobs. During 163 days, 108,651 jobs from which 56,799 Dteam jobs and 45,523 Biomed jobs are received. The other 6329 jobs are rare. For simplicity, our experiments only consider Dteam and Biomed jobs. Dteam jobs are mainly short monitoring jobs (execution time per Dteam jobs is 0.1 h) while Biomed jobs are CPU intensive jobs (execution time per Biomed jobs is 1.5 h). According to this, for the tested large-scale distributed system, the task arrival rate is 28/h. In LPC, jobs are submitted by bursts. To simulate this, we assume that such arrival mode repeats every 6 h (one third tasks arrive continuously at the rate of 66/h while the rest tasks at 9/h). There are 140 homogeneous CPUs. Dteam and Biomed are the two task types. They arrive by the probability of 0.52 and 0.48 respectively. The service rates for Dteam jobs and Biomed jobs are 10/h and 0.67/h. Figs. 10 and 11 show the average response time over 163 days.

Because the task arrival process is hybrid, this experiment does not compare the performance of Algorithm 2. The "ideal" response time in Fig. 10 is obtained under the condition that there are sufficient CPUs. Thus, an idle CPU can be assigned to a task immediately after it arrives. We use it as a baseline. Its value is $0.52 \times 0.1 + 0.48 \times 1.5 = 0.772$. The closer to it, the better the algorithm is. Under this distributed environment, the cluster is relatively idle. So the three batch-mode algorithms have lower performance for the reason mentioned in the small-scale experiment. Compared with ECT, our learning-based algorithm is better, because it could reserve resources for bursting task arrival.

In the above experiment, there are so many processing units that the system keeps idle. In order to examine the performance under heavy loaded environment, we reduce CPUs in LPC from 140 to 15 in the simulation. Fig. 11 shows the 15-CPU LPC responds to each job in about 2.2 h using learning-based algorithm, about one hour faster than other algorithms.

6.4. Parameter setting

In the learning-based algorithm, there are two parameters, i.e., the exploration probability and the discounted factor. This section provides the empirical data for their settings. The experiment environment uses the small-scale distributed system.

Fig. 12 gives the average response time under varied discounted factor when tasks arrive by Poisson distribution with arrival rate $\lambda = 80$. When $\gamma = 0$, the algorithm only takes the immediate reward into account and becomes the ECT algorithm. As γ increases, the future allocations are considered and the average response time decreases. From this figure, there exists a minimum between 0.8 and 0.99. For simplicity, 0.9 is selected as the value of discounted factor.

The parameter of exploration probability functions to guarantee sufficient traversal of all the states. If it is too big, the learning

Table 4
The influence of exploration probability.

p_e	0.1	0.15	0.2	0.25	0.3	0.4
ART	0.0562	0.0483	0.0419	0.0451	0.0507	0.0726
CI	425	482	547	677	829	∞

algorithm cannot converge. If it is too small, the states cannot be visited sufficiently, and the Q-function converges to the wrong value or the local optimum. Under the same settings, Table 4 shows the results of convergence and average response time with varied exploration probability.

ART denotes the average response time while *CI* is the number of iterations for convergence. The Q-function is believed to be converged when the difference is lower than 5% in the following one hundred iterations. In Table 4, the Q-function converges more slowly as p_e increases, and when $p_e = 0.4$, it does not converge. As far as the average response time is concerned, a local optimum is reached when p_e is small and the *ART* gets worse. Considering the two aspects together, its better setting is around 0.2.

7. Conclusion

There existed lots of heuristics for dynamic task scheduling. But they are sensitive to prediction of task execution or neglect possible influence on subsequent tasks. Scheduling performance could be further improved with adaptability to task arrival and execution processes. Consequently, in this paper we use MDP to model scheduling, which regards independent tasks as a flow. Then we introduce machine learning into task scheduling and propose an online Q-learning based algorithm for independent tasks. By guarded adjusting scheduling policies for a task flow, it is able to adapt to task arrival and execution patterns automatically, and reserve resources for forthcoming tasks. Learning makes this adaption without needing any prior knowledge about task arrival and execution. It also leads to robustness for dynamics in task traffic and execution. Compared with some typical heuristic algorithms, this proactive scheduling effectively decreases the average response time, achieving much better load balance.

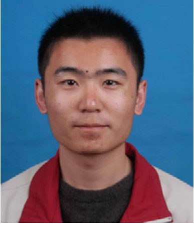
In the future, we intend to transplant our method onto a cluster in our national supercomputing center, and deploy it in the project of our network virtual laboratory.

Acknowledgments

We are very grateful to three anonymous reviewers for the comments and suggestions which greatly improved the quality of the manuscript, and the Director of the National Supercomputing Changsha Center for facilitating the experiments. This research was partially funded by the Key Program of National Natural Science Foundation of China (Grant No. 61133005), the National Natural Science Foundation of China (Grant No. 61370095), the Natural Science Foundation for Distinguished Young Scholars of Hunan (12JJ1011), and supported by the Hunan Provincial Natural Science Foundation (Grant No. 13JJ4038).

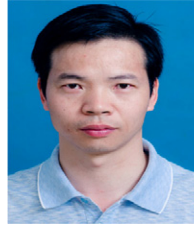
References

[1] M. Aktaruzzaman, Literature review and survey: resource discovery in computational grids, Technical Report, School of Computer Science, University of Windsor, Windsor, Ontario, Canada, 2003.
[2] F. Berman, R. Wolski, et al., Adaptive computing on the Grid using AppLeS, IEEE Trans. Parallel Distrib. Syst. 14 (4) (2003) 369–382.
[3] S.S. Chauhan, R.C. Joshi, A weighted mean time Min–Min Max–Min selective scheduling strategy for independent tasks on Grid, in: Proc. IEEE 2nd Int'l Conf. Advance Computing Conference, IACC2010, Feb. 2010, pp. 4–9, 19–20.
[4] Y. Chung, S. Ranka, Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors, in: Proc. Super-computing, Nov. 1992, pp. 512–521.
[5] S. Chuprat, S. Baruah, Scheduling divisible real-time loads on clusters with varying processor start times, in: 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Aug 2008, pp. 15–24.
[6] K. Etminani, M. Naghibzadeh, A min–min max–min selective algorithm for grid task scheduling, in: Proc. 3rd IEEE/IFIP Int'l Conf. Internet in Central Asia, Sept. 2007, pp. 1–7.
[7] L. Gong, X. Sun, E.F. Watson, Performance modeling and prediction of nondedicated network computing, IEEE Trans. Comput. 51 (9) (2002) 1041–1055.
[8] Donald Gross, John F. Shortle, James M. Thompson, Carl M. Harris, Fundamentals of Queueing Theory, fourth ed., John Wiley & Sons, Hoboken, NJ, 2008.
[9] D. Grosu, A.T. Chronopoulos, M.Y. Leung, Cooperative load balancing in distributed systems, Concurr. Comput.: Pract. Exper. 20 (16) (2008) 1953–1976.
[10] L. He, S.A. Jarvis, D.P. Spooner, D. Bacigalupo, G. Tan, G.R. Nudd, Mapping DAG-based applications to multiclusters with background workload, in: Proc. IEEE Int'l Symposium on Cluster Computing and the Grid, May 2005, pp. 855–862.
[11] S. Jang, X. Wu, V. Taylor, Using Performance Prediction to Allocate Grid Resources, in: Tech. Rep., GriPhyN, 2004.
[12] Y.C. Jiang, Z.C. Huang, The rich get richer: preferential attachment in the task allocation of cooperative networked multiagent systems with resource caching, IEEE Trans. Syst. Man Cybern. A: Syst. Humans 42 (5) (2012) 1040–1052.
[13] Y.C. Jiang, J.C. Jiang, Contextual resource negotiation-based task allocation and load balancing in complex software systems, IEEE Trans. Parallel Distrib. Syst. 20 (5) (2009) 641–653.
[14] L.P. Kaelbling, M.L. Littman, A.W. Moore, Reinforcement learning: a survey, J. Artificial Intelligence Res. 4 (1996) 237–285.
[15] S.J. Kim, J.C. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, in: Proc. Int'l Conf. Parallel Processing, vol. 2, 1988, pp. 1–8.
[16] A. Kretsis, P. Kokkinos, E.A. Varvarigos, Implementing and evaluating scheduling policies in gLite middleware, Concurr. Comput.: Pract. Exper. 25 (3) (2013) 349–366.
[17] Katia Leal, Eduardo Huedo, Ignacio M. Llorente, A decentralized model for scheduling independent tasks in Federated Grids, Future Gener. Comput. Syst. 25 (8) (2009) 840–852.
[18] J. Liou, M.A. Palis, An efficient task clustering heuristic for scheduling DAGs on multiprocessors, in: Proc. Int'l Parallel Processing Symp., 1997, pp. 152–156.
[19] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, R.F. Freund, Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems, J. Parallel Distrib. Comput. 59 (2) (1999) 107–131.
[20] Anwar Mamat, Ying Lu, Jitender Deogun, Steve Goddard, Efficient real-time divisible load scheduling, J. Parallel Distrib. Comput. 72 (2012) 1603–1616.
[21] Emmanuel Medernach, Job arrival analysis of a cluster in a grid environment, in: 11th International Workshop of Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 3834, 2005, pp. 36–61.
[22] S. Penmatsa, A.T. Chronopoulos, Game-theoretic static load balancing for distributed systems, J. Parallel Distrib. Comput. 71 (4) (2011) 537–555.
[23] X. Qin, T. Xie, An availability-aware task scheduling strategy for heterogeneous systems, IEEE Trans. Comput. 57 (2) (2008) 188–199.
[24] R. Sakellariou, H. Zhao, A low-cost rescheduling policy for efficient mapping of workflows on grid systems, J. Sci. Programming 12 (4) (2004) 253–262.
[25] R. Subrate, A.Y. Zomaya, B. landfeldt, Game-theoretic approach for load balancing in computational grids, IEEE Trans. Parallel Distrib. Syst. 19 (1) (2008) 66–76.
[26] X. Sun, M. Wu, Grid harvest service: a system for long-term, application-level task scheduling, in: Proc. the 17th International Symposium on Parallel and Distributed Processing, IPDPS'03, 2003, pp. 25–33.
[27] J.D. Ullman, NP-complete scheduling problems, J. Comput. System Sci. 10 (1975) 384–393.
[28] C. Watkins, P. Dayan, Q-learning, Mach. Learn., 8, 279–292.
[29] M. Wu, X. Sun, Self-adaptive task allocation and scheduling of meta-tasks in non-dedicated heterogeneous computing, J. High Perform. Comput. Netw. (IJHPCN) 2 (2) (2004) 186–197.
[30] F. Xhafa, L. Barolli, A. Durresi, Immediate mode scheduling of independent jobs in computational grids, in: Proc. the 21st International Conference on Advanced Networking and Applications, 2007, pp. 970–977.
[31] Zheng Xiao, Zhao Tong, Kenli Li, Probabilistic Scheduling Based on Queueing Model for Multi-user Network Applications, Computer and Information Technology, CIT, in: 2012 IEEE 12th International Conference on, 2012. pp. 224, 229.
[32] Yuming Xu, Kenli Li, Ligang He, Tung Khac Truong, A DAG scheduling scheme on heterogeneous computing systems using double molecular structure-based chemical reaction optimization, J. Parallel Distrib. Comput. 73 (9) (2013) 1306–1322.
[33] L. Yang, J.M. Schopf, I. Foster, Conservative scheduling: using predicted variance to improve scheduling decisions in dynamic environments, in: Proc. ACM/IEEE Supercomputing Conference, Nov. 2003, pp. 31–46.



Zhao Tong received his M.Sc. from Hunan Agricultural University, China, in 2010, and B.Sc. in Computer Science from Beijing Institute of Technology in 2007. He is currently a Ph.D. candidate in Hunan University, China. His research interest includes modeling and scheduling for parallel and distributed computing systems, parallel system reliability, and parallel algorithms.

Zheng Xiao received his Ph.D. in Computer Science from Fudan University, China, in 2009, and B.Sc. in Communication Engineering from Hunan University in 2003. Now he is an assistant professor in College of Information Science and Engineering of Hunan University. His research interests include parallel and distributed computing, distributed artificial intelligence, Collaborative Computing.



Kenli Li received the Ph.D. in Computer Science from Huazhong University of Science and Technology, China, in 2003, and the M.Sc. in Mathematics from Central South University, China, in 2000. He was a visiting scholar at University of Illinois at Champaign and Urbana from 2004 to 2005. Now he is a professor of Computer science and Technology at Hunan University, a senior member of CCF. His major research includes parallel computing, Grid and Cloud computing, and DNA computer.

Keqin Li received the Ph.D. in Computer Science from University of Houston, USA, in 1990, and B.Sc. in Computer Science from Tsinghua University, China, in 1985. He was the acting chair of Department of Computer Science during Spring 2004, and currently a SUNY Distinguished Professor, State University of New York, USA. He is also an Intellectual Ventures Endowed Visiting Chair Professor at the National Laboratory for Information Science and Technology, Tsinghua University. His research interests are mainly in design and analysis of algorithms, parallel and distributed computing, and computer networking, with particular interests in approximation algorithms, parallel algorithms, job scheduling, task dispatching, load balancing, performance evaluation, dynamic tree embedding, scalability analysis, parallel computing using optical interconnects, wireless networks, and optical networks.