Computing and Informatics, Vol. 36, 2017, 1207–1234, doi: 10.4149/cai_2017_5_1207



# ON-LINE LOAD BALANCING WITH TASK BUFFER

Jiayin Wei, Daoyun Xu, Yongbin Qin, Ruizhang Huang

*College of Computer Science and Technology*
*Guizhou University, China*
*e-mail:* weijiayin05@sina.com,
{dyxu, cse.ybqin, cse.rzhuang}@gzu.edu.cn

**Abstract.** On-line load balancing is one of the most important problems for applications with resource allocation. It aims to assign tasks to suitable machines and balance the load among all of the machines, where the tasks need to be assigned to a machine upon arrival. In practice, tasks are not always required to be assigned to machines immediately. In this paper, we propose a novel on-line load balancing model with task buffer, where the buffer can temporarily store tasks as many as possible. Three algorithms, namely $LPTCP1_\alpha$, $LPTCP2_\alpha$, and $LPTCP3_\beta$, are proposed based on the *Longest Processing Time* ($LPT$) algorithm and a variety of planarization algorithms. The planarization algorithms are proposed for reducing the difference among each element in a set. Experimental results show that our proposed algorithms can effectively solve the on-line load balancing problem and have good performance in large scale experiments.

**Keywords:** On-line schedule, load balancing, task buffer, planarization processing

## 1 INTRODUCTION

Load balancing is a fundamental problem in distributing and communication systems. Without proper scheduling and resource allocation, tasks accumulate at each machine unevenly which resulting in an unbalanced system. In extreme cases, some machines are overloaded while some are starved. Unbalanced systems can cause serious problems: System resources cannot be fully utilized. System operates inefficiently which results in low productivity. Therefore, the study of the load balancing problem is valuable and significant. In the recent years, researchers have developed a number of algorithms to solve this problem, among which max-min task schedul-

ing [1], evolutionary meta-heuristic [2, 3, 4], bio-inspired search algorithm [5], ant colony algorithm [6, 7], extremal optimization [8], and algorithmic mechanism design [9, 10] are the most discussed.

In general, load balancing can be classified in two categories: off-line and on-line. In the off-line setting, the complete knowledge of the entire input is available. The off-line load balancing problem is NP-hard [11]. In the on-line setting [12, 13], there are $m$ parallel machines with speeds $s_1$, $s_2$, ..., $s_m$ (if all machines have the same speed, i.e. $s_1 = s_2 = \ldots = s_m$, they are called identical machines) and $n$ independent tasks with weight $w_1$, $w_2$, ..., $w_n$. Machine $i$ needs $p_i(j) = w_j/s_i$ time when task $j$ is processed on it. A task is required to be assigned to exactly one of the machines and this allocation is not allowed to be changed. All tasks must be assigned with no admission control. Let $L_i(t-1)$ denote the load of machine $i$ at time $t-1$. If task $j$ is assigned to machine $i$ at time $t$, the load of machine $i$ is increased by $p_i(j)$, calculated as follows:

$$L_k(t) = \begin{cases} L_k(t-1) + p_i(j) & \text{if } k = i, \\ L_k(t-1) & \text{o.w.} \end{cases} \tag{1}$$

Similarly, if task $j$ departs from machine $i$ at time $t$, the load on this machine is decreased by $p_i(j)$. Note that for identical machines, $\forall i, j : p_i(j) = w_j$. The common objective for the load balancing problem is to minimize the maximum load. However, some other objectives have been considered, e.g., the maximum flow time, the minimum machine completion time, the total completion time, the sum of weighted tardiness, etc.

The competitive ratio is often used to measure the performance of an on-line algorithm. Given a set of instances $\mathcal{I}$, for an algorithm $A$ and each instance $I \in \mathcal{I}$, let $c^A(I)$ denote the objective function value produced by $A$ and let $c^*(I)$ denote the optimal value in an off-line version. Then the competitive ratio of $A$ is defined as the smallest number $r \geq 1$ such that for any $I$, $c^A(I) \leq r \cdot c^*(I)$.

In this paper, we consider a new variant of on-line load balancing problem with task buffer The remainder of this paper is structured as follows. Section 2 provides a detailed related work. Section 3 provides the definition of the problem of on-line load balancing and the existing algorithms for the general version. Section 4 provides the concept of flatting set. Section 5 explains some algorithms for solving this new problem of on-line load balancing with task buffer. Section 6 details the experimental results. Finally, the conclusions of this work are then given in Section 7.

## 2 RELATED WORK

Many on-line load balancing models have been studied in the literature. Even for the most basic case of identical machines, there is a gap between the best known lower and upper bounds of the competitive ratio, in particular, 1.880 [14] and 1.9201 [15], respectively. Divakaran and Saks [16] considered an on-line variant of scheduling

with setup times and release times. They proposed a $O(1)$-competitive online algorithm for minimizing the maximum flow time if tasks arrive over time at one single machine. Liu and Lu [17] presented an online algorithm for two parallel machines with release dates and delivery times which has a competitive ratio of $\left(1+\sqrt{5}\right)/2 \approx 1.618$.

Semi-online scheduling, which assumes that some partial additional information is known in advance, was first introduced by Liu et al. [18]. They considered a semi-online problem with ordinal data on parallel machines, where task processing times are not known but the order of task by processing time is known. He et al. [19] investigated preemptive semi-online scheduling problems on $m$ identical parallel machines, where the total size of all tasks is known in advance. They presented an optimal semi-online algorithm with the objective to minimize the maximum machine completion time $C_{max}$, i.e., minimize the *makespan*, which achieves competitive ratio to 1. For the objective of maximizing the minimum machine completion time $C_{min}$, they demonstrated that the competitive ratio of any semi-online algorithm is at least $\frac{2m-3}{m-1}$ for any $m > 2$. Luo and Xu [20] considered the online hierarchical scheduling problem on two parallel machines with bounded processing times. The processing times are bounded by an interval $[1, \alpha]$. For the objective of minimizing the $C_{max}$, they proved that no algorithm can have a competitive ratio less than $1 + \alpha$.

Adding lookahead is a common practice to improve the performance of algorithm for on-line problems. However, lookahead alone is not sufficient. The paradigm of on-line reordering is more powerful than lookahead alone and has received a lot of attention [21, 22]. Englert et al. [22] presented an algorithm, for online minimum makespan scheduling, with a buffer of size $O(m)$ which achieves a competitive ratio of $\frac{W_{-1}\left(-1/e^2\right)}{(1+W_{-1}(-1/e^2))} \approx 1.4659$. In this model, at each time, the reordering buffer always contains the first $k$ tasks of the input sequence that have not been assigned so far.

We present an extensive study on the power of online reordering for the on-line load balancing problem. In our model, each task enters into a buffer when it arrives. When some machines finish processing all the tasks assigned, appropriate tasks are selected from the buffer and allocated to those machines. Different from the model in [22], the size of the buffer is determined by the number of the tasks that have not been assigned after their arrival. The task buffer needs to be flatted (i.e., reduce the difference among each task by some technology) before tasks are selected to be allocated to some machines. The assignment of task only occurs when there are machines available rather than right after tasks arrive. Three algorithms for the on-line load balancing, namely $LPTCP1_\alpha$, $LPTCP2_\alpha$, and $LPTCP3_\beta$, are designed based on the two new $\alpha$-planarization algorithms, a $\beta$-planarization algorithm in [23], and the *Longest Processing Time* ($LPT$) algorithm in [24]. Experimental results show that all three algorithms generate good solutions. The proposed algorithms show the robust performance with large scale experiments.

## 3 ON-LINE LOAD BALANCING AND EXISTING ALGORITHMS

In this section, the definition of general on-line load balancing problem and the problem of on-line load balancing with task buffer are firstly introduced. Algorithms for the general on-line load balancing problem are then discussed. Note that there are no related algorithms for the problem of on-line load balancing with task buffer.

### 3.1 Problem Definition

The general on-line load balancing problem has been extensively studied. It requires a task to be assigned right after it arrives to exactly one of the machines. However, tasks may not be allocated immediately in real cases. Considering the case of resources allocation in computer system, each task will be allocated after corresponding resources are available. The above problem widely exists in the areas of computing and communications. Therefore, we design an on-line load balancing model with task buffer, where the buffer can temporarily store tasks as many as possible. Each task enters into the task buffer when it arrives. Tasks in this buffer will be allocated according to the status of machines. We name it the problem of on-line load balancing with task buffer, and describe it in detail in the rest of the subsection.

Suppose there are $m$ identical machines, $n$ independent tasks and an unlimited sized task buffer (in fact, the size of this task buffer is not more than $n$), a task $j$ arrives at time $r_j$ with processing time $p_j$. The processing time $p_j$ is the time task $j$ takes to reach its completion. Because the primary role of the task buffer is to temporarily store tasks for later access, we name it *Cache*. Each task enters into the *Cache*, rather than be assigned immediately, when it arrives. Different from the traditional model, the load of a machine $i$ is estimated as the completion time of the last task on the machine $i$, denoted as $L_i$. When machines finish processing all tasks assigned, appropriate tasks are selected from the *Cache* and allocated to them. The allocation is not allowed to be changed. The objective function is to find a schedule to minimize the maximum load (*makespan*), which is defined as $C_{\max} = \max_{i \in m} \{L_i\}$.

### 3.2 Approaches for the General On-Line Load Balancing Problem

Extensive work has been done on the general on-line load balancing problem. The *Greedy* algorithm, the *Semi-Greedy* algorithm, and the *LPT* algorithm are the most classical algorithms.

R. L. Graham [25] presented a *Greedy* algorithm for solving the problem of on-line load balancing. The idea is to allocate each task to the machine with the current minimum load. The *Greedy* algorithm leads to a $(2 - \frac{1}{m})$-competitive solution in $O(mn)$ time. Good solution can be obtained in the early stage of the algorithm. However the quality of the solution decreases in later stage, because of the lack of knowledge about future incoming tasks. Therefore, Hart and Shogan proposed

a *Semi-Greedy* algorithm in [26]. At each iteration, the choice of to which machine an arriving task can be allocated is determined by a semi-greedy strategy. All candidate machines in a candidate list $CL$ are ordered based on a greedy function. A sublist of the candidate machines, which includes the first $r$ machines, in the ordered candidate list $CL$, is selected. This sublist is denoted as the *Restricted Candidate List* (*RCL*). The *Semi-Greedy* algorithm then randomly selects a machine from the *RCL*. It is easy to verify that if $r = 1$, the *Semi-Greedy* algorithm behaves like a *Greedy* algorithm. If $r = |CL|$, the *Semi-Greedy* algorithm becomes a random algorithm. The basic idea of the $LPT$ algorithm, proposed by R. L. Graham in [24], is to firstly sort the processing times of tasks by descending order, and assign tasks in turns to the machine with the lowest load. The $LPT$ algorithm is a $\left(\frac{4}{3} - \frac{1}{3m}\right)$-approximation algorithm with running time $O(n \log n)$.

**Example 1.** Consider an instance of on-line load balancing problem with 4 machines and 9 tasks (i.e. $m = 4$, $n = 9$). The release time $r_j$ and processing time $p_j$ of each task $j$ are shown in Table 1.

| Task $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $r_j$ | 0 | 1 | 1 | 3 | 4 | 4 | 7 | 7 | 8 |
| $p_j$ | 8 | 4 | 5 | 6 | 6 | 7 | 3 | 2 | 4 |

Table 1. The release time and processing time of each task

The solution of the on-line load balancing problem is composed by an allocation scheme $X$ and the corresponding *makespan* $C_{max}$. The element of $X$ is $(T_j, m_i, t)$, it means that task $T_j$ is processed on machine $m_i$ at time $t$.

With the *Greedy* algorithm on Example 1, we can get a solution $X_{Greedy} = \{(1,1,0),\ (2,2,1),\ (3,3,1),\ (4,4,3),\ (5,2,5),\ (6,3,6),\ (7,1,8),\ (8,4,9),\ (9,1,11)\}$, $C_{\max} = 15$. The solution produced by the *Semi-Greedy* algorithm with $r = 2$ may be $X_{Semi\text{-}Greedy} = \{(1,1,0),\ (2,2,1),\ (3,3,1),\ (4,4,3),\ (5,2,5),\ (6,3,6),\ (7,4,9),\ (8,1,8), (9,1,10)\}$, $C'_{\max} = 14$. The solution obtained by the $LPT$ algorithm is $X_{LPT} = \{(1,1,0),\ (2,3,1),\ (3,2,1),\ (4,4,3),\ (5,2,6),\ (6,3,5),\ (7,1,8),\ (8,4,9),\ (9,1,11)\}$, $C''_{\max} = 15$. The results are shown in Figure 1.

It is obvious that there are big differences among the maximum loads obtained by the *Greedy*, the *Semi-Greedy*, and the $LPT$ algorithms. The reason is that the machine allocation decision of task is made without the consideration of two key factors. The first factor is the release and the process time of future arriving tasks. The second factor is the busy or idle status of machines. The above algorithms all allocate each task by finding the current local minimum makespan, which may not result in good solutions.

In this paper, we propose a new model of the on-line load balancing problem with task buffer, the task buffer is denoted as *Cache*. In our model, each task enters into the *Cache* when it arrives. When some machines finish processing all tasks assigned, appropriate tasks are selected from the *Cache* and allocated to them. This allocation

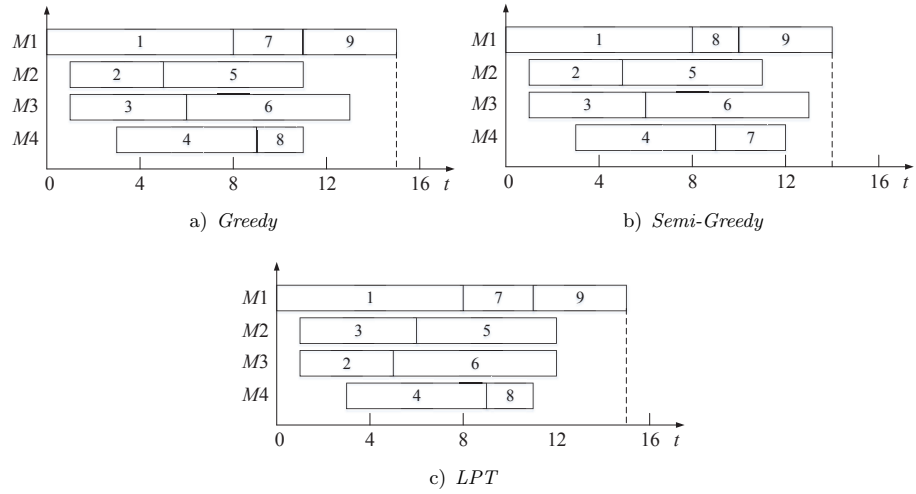



Figure 1. The results of Example 1 obtained by the *Greedy*, the *Semi-Greedy*, and the *LPT* algorithms

is not allowed to be changed. As suggested by [23, 27], when the difference between the processing times of each task is small enough, the assignment of each task will be simpler and better scheduling results will be achieved. We design three planarization algorithms on the *Cache*, for properly assigning a task in the *Cache* to a machine. In the following sections, we will firstly introduce the concept of the flatting set and three planarization algorithms in Section 4. Three algorithms for the on-line load balancing problem with task buffer based on the planarization techniques and the basic idea of the *LPT* algorithm are then presented in Section 5. In Section 6, experimental results are described.

## 4 FLATTING SET AND PLANARIZATION ALGORITHM

### 4.1 The Concept of Flatting Set

In order to formalize the level of distinctness among each elements in a positive real number set, the concepts of $\alpha$-flatting set and $\beta$-flatting set have been proposed respectively in [27] and [23]. The specific formalized definitions are shown in Definition 1 and 2.

**Definition 1** ($\alpha$-flatting set)**.** A set $A = \{a_i | a_i \in \mathbb{R}^+, i \in [n]\}$ is a $\alpha$-flatting set, if there exists a factor $\alpha \in [0, 1)$ such that for every $i \neq j \in [n]$,

$$|a_i - a_j| \leq \alpha \cdot a_{\min}$$

where $a_{\min} = \min\{a_1, a_2, \ldots, a_n\}$.

For example, if $\alpha = 0.5$, $A = \{21, 20, 18, 24, 26\}$ is a $\alpha_{0.5}$-flatting set, while $A' = \{3, 5, 9, 6\}$ is not a $\alpha_{0.5}$-flatting set. Particularly, $|A| = 1$ is a $\alpha_0$-flatting set (i.e. $\alpha = 0$). We do not need to do any special operations. In the following section, we assume that $|A| = n \geq 2$.

**Definition 2** ($\beta$-flatting set)**.** A set $A = \{a_i | a_i \in \mathbb{R}^+, i \in [n]\}$ is a $\beta$-flatting set, if there exists a factor $\beta \geq 1$ such that for every $i \neq j \in [n]$,

$$\frac{1}{\beta} \leq \frac{a_i}{a_j} \leq \beta.$$

For example, if $\beta = 4$, $A = \{12, 9, 15, 24, 30\}$ is a $\beta_4$-flatting set, while $A' = \{4, 5, 17, 11\}$ is not a $\beta_4$-flatting set. Similarly, $|A| = 1$ is a $\beta_1$-flatting set (i.e. $\beta = 1$). We do not need to do any special operations. In the following section, we assume that $|A| = n \geq 2$.

Given a set $A = \{a_i | a_i > 0, i \in [n]\}$, let $a_{\min} = \min\{A\}$, $a_{\max} = \max\{A\}$. Whether $A$ is an $\alpha$(or $\beta$)-flatting set can be determined by the Theorem 1 (or 2).

**Theorem 1.** A set $A = \{a_i | a_i \in \mathbb{R}^+, i \in [n]\}$ is an $\alpha$-flatting set iff $a_{max} \leq (1+\alpha) \cdot a_{min}$.

**Proof.** Without loss of generality, assume that $a_1 \geq a_2 \geq \ldots \geq a_n$, then $a_{\min} = a_n$, $a_{\max} = a_1$.

If $a_{max} \leq (1+\alpha) \cdot a_{min}$, then for every $a_i$, $a_j$, $i \neq j \in [n]$ satisfies,

$$|a_i - a_j| \leq a_1 - a_n \leq (1+\alpha) \cdot a_n - a_n \leq \alpha \cdot a_n.$$

By Definition 1, $A$ is an $\alpha$-flatting set.

Conversely, let $A$ be an $\alpha$-flatting set. By the Definition 1, for every $i \neq j \in [n]$, there is a factor $\alpha \in [0, 1)$ such that $|a_i - a_j| \leq \alpha \cdot a_{\min} = \alpha \cdot a_n$, thus $a_{max} - a_{min} = a_1 - a_n \leq \alpha \cdot a_n$, i.e. $a_{max} \leq (1+\alpha) \cdot a_{min}$. □

**Corollary 1.** A set $A = \{a_i | a_i \in \mathbb{R}^+, i \in [n]\}$ is an $\alpha$-flatting set, then $1 \leq \frac{a_{max}}{a_{min}} < 2$.

**Proof.** The proof is immediate since $\alpha \in [0, 1)$ and $a_{min} \leq a_{max} \leq (1+\alpha) \cdot a_{min}$. □

**Theorem 2.** A set $A = \{a_i | a_i \in \mathbb{R}^+, i \in [n]\}$ is a $\beta$-flatting set iff $a_{\max} \leq \beta \cdot a_{\min}$.

**Proof.** Assuming that, w.l.o.g., $a_1 \geq a_2 \geq \ldots \geq a_n$, then $a_{\min} = a_n$, $a_{\max} = a_1$.

If $a_{\max} \leq \beta \cdot a_{\min}$, then for every $a_i$, $a_j$, $i \neq j \in [n]$ satisfies,

$$\frac{a_i}{a_j} \leq \frac{a_1}{a_n} = \frac{a_{\max}}{a_{\min}} \leq \beta.$$

By Definition 2, $A$ is a $\beta$-flatting set.

Conversely, if $A$ is a $\beta$-flatting set, by the Definition 2 we have $a_{\max} = a_1 \leq \beta \cdot a_n = \beta \cdot a_{\min}$. □

By Theorem 1 and 2, we know that $\alpha$-flat and $\beta$-flat are consistent when $\beta \in [1, 2)$.

### 4.2 Planarization Algorithm and Its Efficiency

A none-$\alpha$ (or $\beta$)-flatting set $A$ can be converted to a corresponding $\alpha$ (or $\beta$)-flatting set by using planarization techniques. According to the $\alpha$-planarization algorithm in [27] and the $\beta$-planarization algorithm in [23], the general framework of a planarization Algorithm ($PA$) is depicted in Algorithm 1.

**Algorithm 1:** *Planarization Algorithm, PA*

**Input:** $A = \{a_1, a_2, \ldots, a_n\}$, $\mathcal{P}$.
**Output:** A flat sequence $B$.

1 $n \leftarrow |A|$ ;
2 **for** $i \leftarrow 1$ **to** $n$ **do**
3 $\quad c(i).Index \leftarrow i$, $c(i).v \leftarrow a_i$, $b(i) \leftarrow \{c(i)\}$, $b(i).v \leftarrow a_i$;
4 $\quad \mathbb{B} \leftarrow \{b(1), b(2), \ldots, b(n)\}$;
5 **Sorting:** $B \leftarrow \{b(\pi(1)), b(\pi(2)), \ldots, b(\pi(n))\}$ where $b(\pi(i)) \in \mathbb{B}$ and $\forall 1 \leq i < j \leq n, b(\pi(i)).v \geq b(\pi(j)).v$ ;
6 **while** $b(\pi(1)).v / b(\pi(n)).v > \mathcal{P}$ **do**
7 $\quad$ Flatting $B$ by a planarization procedure and the result is $B' = \{b'_i\}$ ;
8 $\quad n \leftarrow |B'|$ ;
9 $\quad$ **Sorting:** $B \leftarrow \{b(\pi(1)), b(\pi(2)), \ldots, b(\pi(n))\}$ where $b(\pi(i)) \in B'$ and $\forall 1 \leq i < j \leq n, b(\pi(i)).v \geq b(\pi(j)).v$ ;
10 **return** $B$;

The input parameter $\mathcal{P}$ in Algorithm 1 is the target degree of flatness. The step 1 is used to get the size of $A$. The steps 2 to 4 are used to construct a set $\mathbb{B} = \{b(1), b(2), \ldots, b(n)\}$, where $b(i)$ is a set of $c(i)$, $c(i).Index$ is the index of $a_i$ in $A$, $c(i).v$ is the value of $a_i$, $b(i).v$ is the sum of all element in it. The step 5 is used to sort $\mathbb{B}$ in descending order by $b(i).v$, the result is $B$. The step 6 is a while loop, it will run until $B$ is a flatting set. The step 7 is flatting $B$ by a planarization procedure and the result is $B'$. The step 9 is used to sort $B'$ in descending order by $b(i).v$ and the result is $B$. It is worth noting that the purpose of steps 5 and 9 is to ensure that the set $B$, which is the input of a planarization procedure or the result of Algorithm 1, is ordered.

Of all the processing steps of Algorithm 1, the step 7 is the most critical. In step 7, some elements in the set are combined into a new element by a planarization procedure. The performance of the Algorithm 1 is closely related to the planarization procedure adopted. In [27], the $PA_\alpha$ was proposed which combines the smallest and the second smallest elements into a new element. This procedure is straightforward but it is not efficient.

Taking the $PA_\beta$ proposed in [23] as a reference, we design two new $\alpha$-planarization procedures. The first one, namely *APBM*, is designed taking the mean value of all elements into consideration. The efficiency of the *APBM* is improved by making the new element generated in each loop as close as possible to the mean value of all the elements. The reason is that the purpose of flatting is to reduce the gap among the elements of the set. A detail description of the *APBM* is shown in Procedure APBM.

**Procedure 1:** APBM($B$)

1 $B' \leftarrow \{b(\pi(1))\}$, $i \leftarrow 2$, $l \leftarrow n$, $b_{avg} \leftarrow \left(\sum_{i=1}^{n} b(\pi(i)).v\right)/n$ ;
2 **while** $l \geq i$ **do**
3  **if** $l = i$ **then return** $B' \leftarrow B' \cup \{b(\pi(i))\}$ ;
4  **if** $b(\pi(i)) \geq b_{avg}$ **then**
5  $B' \leftarrow B' \cup \{b(\pi(i))\}$, $i = i + 1$ ;
6  **else**
7  Let $b'(\pi(i)) \leftarrow \{b(\pi(i)), b(\pi(i')), \ldots, b(\pi(l))\}$ and $b'(\pi(i)).v \leftarrow b(\pi(i)).v + \sum_{k=i'}^{l} b(\pi(k)).v$, where $i' \leftarrow \arg\max_{j>i} \left\{ \left( b(\pi(i)).v + \sum_{k=j}^{l} b(\pi(k)).v \right) \geq b_{avg} \right\}$ (if there is no such $i'$ then let $i' \leftarrow i + 1$) ;
8  Let $i \leftarrow i + 1$, $l \leftarrow i' - 1$, $B' \leftarrow B' \cup \{b'(\pi(i))\}$ ;
9 **return** $B'$ ;

In Procedure APBM, the APBM($B$) means that the input parameter of the Procedure APBM is $B$. $b_{avg}$ is the mean value of all elements in $B$. The variable $i$ ($l$) is a forward (backward) scanning variable. The while loop, from step 2 to 8, will always be executed until $l < i$.

The second $\alpha$-planarization procedure, namely *APBT*, is designed based on the Theorem 1. It is easy to see that an $\alpha$-flatting set $A$ satisfies $a_{max} \leq (1 + \alpha) \cdot a_{min}$. The efficiency of the *APBT* is improved by making the new element generated in each loop satisfying $a_{max} \leq (1 + \alpha) \cdot a_{min}$. A detail description of the *APBT* is shown in Procedure APBT.

The APBT($B$, $\mathcal{P}$) in Procedure APBT means that the input parameters of the Procedure APBT are $B$ and $\mathcal{P}$. $\mathcal{P}$ is the target degree of flatness. $b_{\max}$ is the large value of all elements in $B$, i.e. $b(\pi(1)).v$. Similarly to the Procedure APBM, the variable $i$ ($l$) is a forward (backward) scanning variable. The while loop, from step 2 to 9, will always be executed until $l < i$. The condition $b(\pi(i)) \geq \frac{b_{\max}}{1+\mathcal{P}}$, in the step 5, is designed based on the Theorem 1.

The time complexity of both the Procedure APBM and APBT is $O(n)$, because only one element in the set is processed at a time. The sort operation of the Algorithm 1 is in $O(n \log n)$ time complexity when heap sort is used. Algorithm 1 needs to be conducted in $O(\log n)$ iteration at most. Therefore the time complexity of the

**Procedure 2:** APBT($B$, $\mathcal{P}$)

$B' \leftarrow \{b(\pi(1))\}$, $i \leftarrow 2$, $l \leftarrow n$, $b_{\max} \leftarrow b(\pi(1)).v$ ;
**while** $l \geq i$ **do**
  **if** $l = i$ **then return** $B' \leftarrow B' \cup \{b(\pi(i))\}$ ;
  ;
  **if** $b(\pi(i)) \geq \frac{b_{\max}}{1+\mathcal{P}}$ **then**
    $B' \leftarrow B' \cup \{b(\pi(i))\}$, $i = i + 1$ ;
  **else**
    Let $b'(\pi(i)) \leftarrow \{b(\pi(i)), b(\pi(i')), \ldots, b(\pi(l))\}$ and
    $b'(\pi(i)).v \leftarrow b(\pi(i)).v + \sum_{k=i'}^{l} b(\pi(k)).v$, where
    $i' \leftarrow \arg\max_{j>i} \left\{ \left( b(\pi(i)).v + \sum_{k=j}^{l} b(\pi(k)).v \right) \geq \frac{b_{\max}}{1+\alpha} \right\}$ (if there is no such $i'$ then let $i' \leftarrow i + 1$) ;
    Let $i \leftarrow i + 1$, $l \leftarrow i' - 1$, $B' \leftarrow B' \cup \{b'(\pi(i))\}$ ;
**return** $B'$ ;

Algorithm 1 is $O(n(\log n)^2)$. When Algorithm 1 use the Procedure APBM (APBT) as the planarization procedure, it is denoted as $PA1_\alpha$ ($PA2_\alpha$).

For the $\beta$-Flatting, we use the $PA_\beta$ proposed in [23], namely $BPA$ in this paper. The main idea is to repeat the process of generating a new set $A'$ by merging $a_i$ and $a_{n+1-i}$ into $a'_i$ until $A'$ is a $\beta$-flatting set. The time complexity of the $BPA$ is $O(n(\log n)^2)$ [23].

## 5 ALGORITHMS FOR THE ON-LINE LOAD BALANCING WITH TASK BUFFER

On-line load balancing with task buffer is a new problem. All existing algorithms for the general on-line load balancing problem cannot be used directly. In this section, we modify the *Greedy*, *Semi-Greedy*, and $LPT$ algorithms for solving this new problem. Three algorithms are then designed based on the $LPT$ algorithm and the planarization algorithms $PA1_\alpha$, $PA2_\alpha$, and $PA_\beta$.

### 5.1 General Approaches

Note that, by definition of the on-line load balancing with task buffer, an unlimited task buffer is employed for task allocation. Tasks in the buffer are assigned only when there are available machines. In order to modify the *Greedy*, the *Semi-Greedy*, and the $LPT$ algorithm for solving this problem, a task buffer denoted by *Cache* and a list of currently available machines denoted by $AM$ are employed. Each task enters into the *Cache* when it arrives at the system. The operation of task allocation is conducted when the *Cache* and $AM$ are both nonempty.

We design an algorithm, namely the *Greedy with Cache* (*GreedyC*), to solve the problem of on-line load balancing with task buffer based on the *Greedy* algorithm. In each iteration, set $k = \min\{|Cache|, |AM|\}$ as the number of tasks which need to be allocated in this time. Assign the first $k$ tasks in the *Cache* to the corresponding available machines with the minimum load and update the loads of these machines by

$$L_i = t + p_j \tag{2}$$

where $t$ is the current time, $p_j$ is the processing time of the task $j$. $L_i$ is the load of machine $i$ which is allocated to the task $j$. The detail description of the *GreedyC* algorithm is shown in Algorithm 2.

**Algorithm 2:** *Greedy with Cache, GreedyC*

**Input:** Task set $T = \{(r_j, p_j) | j \in [n]\}$, machine count $m$.
**Output:** Maskspan $C_{max}$, task list $TL$ where $TL(i)$ is the task list assigned to machine $i$.

1 $\forall i \in [m]$, $L(i) \leftarrow 0$, $TL(i) \leftarrow \emptyset$ ;
2 $t \leftarrow 0$, $Cache \leftarrow \emptyset$, $U \leftarrow$ sort $T$ in ascending order of $r_j$ ;
3 **while** $Cache \neq \emptyset$ **or** $U \neq \emptyset$ **do**
4 $\quad Cache \leftarrow Cache \cup \{(r_j, p_j) | (r_j, p_j) \in U \wedge r_j = t\}$, $AM \leftarrow \{i | L(i) \leq t \wedge i \in [m]\}$ ;
5 $\quad k \leftarrow \min\{|Cache|, |AM|\}$ ;
6 $\quad$ **for** $l \leftarrow 1$ **to** $k$ **do**
7 $\quad\quad mid \leftarrow \arg\min_{i \in AM}\{L(i)\}$ ;
8 $\quad\quad TL(mid) \leftarrow TL(mid) \cup Cache(l)$, $L(mid) \leftarrow t + Cache(l, 2)$ ;
9 $\quad\quad Cache \leftarrow Cache \backslash \{Cache(l)\}$ ;
10 $\quad t \leftarrow t + 1$ ;
11 **return** $TL$ *and* $C_{\max} = \max_{i \in [m]}\{L(i)\}$;

**Remark 1.** The notation $Cache(l, 2)$ in the step 8 of Algorithm 2 refers to the second element of $Cache(l)$, i.e., the processing time of the task $Cache(l)$. The structure of *Cache* is shown in Figure 2.

Similarly to the *GreedyC* algorithm, we modify the *Semi-Greedy* algorithm for solving the problem of on-line load balancing with task buffer, named as the *Semi-Greedy with Cache* (*Semi-GreedyC*) algorithm. In each iteration, for each task $j$ of the first $k$ tasks in the *Cache*, assign it to a machine $i$ which is randomly selected from the sublist of machines formed by the first $r$ available machines. Update the load of machine $i$ by Equation (2). The detail description of the *Semi-GreedyC* algorithm is shown in Algorithm 3.

**Remark 2.** The operation $rand(1, min\{r, |AM|\})$ in the step 7 of Algorithm 3 refers to choosing a number in the integer interval $[1, min\{r, |AM|\}]$ at random.

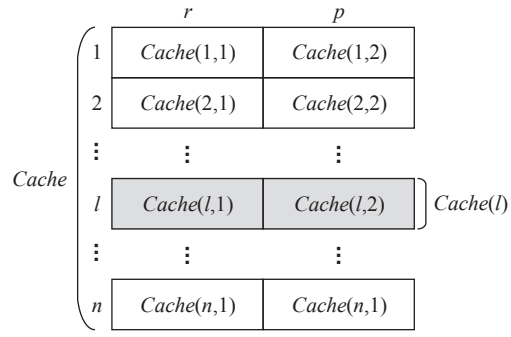



Figure 2. The structure of *Cache*

**Algorithm 3:** *Semi-Greedy with Cache, Semi-GreedyC*

**Input:** Task set $T = \{(r_j, p_j) \,|j \in [n]\}$, machine count $m$, the parameter of restricted candidate list $r(1 \leq r \leq m)$.

**Output:** Maskspan $C_{max}$, task list $TL$ where $TL(i)$ is the task list assigned to machine $i$.

```
1  ∀i ∈ [m], L(i) ← 0, TL(i) ← ∅ ;
2  t ← 0, Cache ← ∅, U ← sort T in ascending order of r_j ;
3  while Cache ≠ ∅ or U ≠ ∅ do
4      Cache ← Cache ∪ {(r_j, p_j) | (r_j, p_j) ∈ U ∧ r_j = t},
         AM ← {i|L(i) ≤ t ∧ i ∈ [m]} ;
5      k ← min {|Cache|, |AM|} ;
6      for l ← 1 to k do
7          mid ← rand(1, min{r, |AM|}) ;
8          TL(AM(mid)) ← TL(AM(mid)) ∪ Cache(l),
             L(AM(mid)) ← t + Cache(l, 2) ;
9          Cache ← Cache \ {Cache(l)}, AM ← AM \ {mid} ;
10     t ← t + 1 ;
11 return TL and C_max = max_{i∈[m]} {L(i)};
```

Similarly, we modify the *LPT* algorithm to solve the problem of on-line load balancing with task buffer, named as the *LPT with Cache* (*LPTC*) algorithm. In each iteration, sort the processing times of tasks in the *Cache* by descending order. The result is named as *CacheTmp*. Assign the first $k$ tasks in the *CacheTmp* to the corresponding available machines with the minimum load. Update the loads of these machines by Equation (2). The detail description of the *LPTC* algorithm is shown in Algorithm 4.

**Algorithm 4:** *LPT with Cache, LPTC*

**Input:** Task set $T = \{(r_j, p_j) \,|j \in [n]\}$, machine count $m$.
**Output:** Maskspan $C_{max}$, task list $TL$ where $TL(i)$ is the task list assigned to machine $i$.

1 $\forall i \in [m]$, $L(i) \leftarrow 0$, $TL(i) \leftarrow \emptyset$ ;
2 $t \leftarrow 0$, $Cache \leftarrow \emptyset$, $U \leftarrow$ sort $T$ in ascending order of $r_j$ ;
3 **while** $Cache \neq \emptyset$ **or** $U \neq \emptyset$ **do**
4 $\quad Cache \leftarrow Cache \cup \{(r_j, p_j) \,|\, (r_j, p_j) \in U \wedge r_j = t\}$, $AM \leftarrow \{i | L(i) \leq t \wedge i \in [m]\}$ ;
5 $\quad k \leftarrow \min\{|Cache|, |AM|\}$ ;
6 $\quad$ **if** $k > 0$ **then**
7 $\quad\quad CacheTmp \leftarrow$ sort $Cache$ in descending order of $p_j$ ;
8 $\quad\quad$ **for** $l \leftarrow 1$ **to** $k$ **do**
9 $\quad\quad\quad mid \leftarrow \arg\min_{i \in AM}\{L(i)\}$ ;
10 $\quad\quad\quad TL(mid) \leftarrow TL(mid) \cup CacheTmp(l)$, $L(mid) \leftarrow t + CacheTmp(l, 2)$ ;
11 $\quad\quad\quad Cache \leftarrow Cache \backslash \{CacheTmp(l)\}$ ;
12 $\quad t \leftarrow t + 1$ ;
13 **return** $TL$ *and* $C_{\max} = \max_{i \in [m]}\{L(i)\}$;

### 5.2 Three More Efficient Algorithms

One problem of the *GreedyC*, *Semi-GreedyC*, and *LPTC* algorithms is that all these algorithms do not take the full usage of the task information in the *Cache* in to consideration. Three algorithms, namely $LPTCP1_\alpha$, $LPTCP2_\alpha$, and $LPTCP3_\beta$, are designed based on the *LPTC* algorithm. A varity of planarization algorithms are employed to increase the usage of the task information in the *Cache*, which improve the algorithm efficiency in return.

We improve the *LPTC* algorithm by using the planarization algorithms proposed in the Section 4.2, named as the *LPT with Cache based on PA* (*LPTCP*). When the number of tasks in the *Cache* is not more than the number of available machines $|AM|$, back to the *LPTC* algorithm. Otherwise, use the Algorithm 1 with a parameter of $|AM|$ on the *Cache* to get a quasi-flatting set *CacheP* with the size no less than $|AM|$. Assign the first task of each element in the *CacheP* to the corresponding available machines. The *LPTCP* algorithm is depicted in Algorithm 5.

*CacheP* in Algorithm 5 in the step 13 of Algorithm 5 is a task sequence obtained by the *PA*. The element of *CacheP* is a virtual task which is composed by one or more tasks of *Cache*. The processing time of $CacheP(l)$ is the sum of the processing times of all tasks in it. $CacheP(l, 1)$ is the first task with the longest processing time of $CacheP(l)$, and $CacheP(l, 1).p$ is the processing time of $CacheP(l, 1)$.

**Algorithm 5:** *LPT with Cache based on PA, LPTCP*

**Input:** Task set $T = \{(r_j, p_j) \,|j \in [n]\}$, machine count $m$, flat parameter $\mathcal{P}$.

**Output:** Maskspan $C_{max}$, task list $TL$ where $TL(i)$ is the task list assigned to machine $i$.

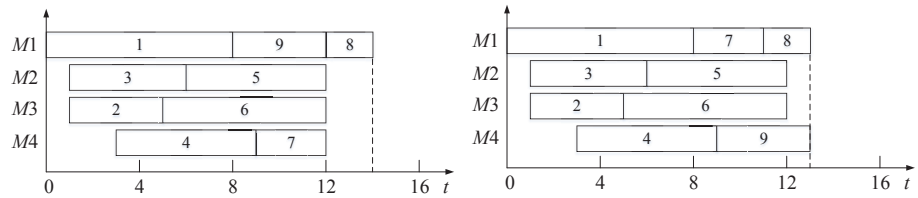
1 $\forall i \in [m]$, $L(i) \leftarrow 0$, $TL(i) \leftarrow \emptyset$ ;
2 $t \leftarrow 0$, $Cache \leftarrow \emptyset$, $U \leftarrow$ sort $T$ in ascending order of $r_j$ ;
3 **while** $Cache \neq \emptyset$ **or** $U \neq \emptyset$ **do**
4 $\quad Cache \leftarrow Cache \cup \{(r_j, p_j) \,|\, (r_j, p_j) \in U \wedge r_j = t\}$, $AM \leftarrow \{i | L(i) \leq t \wedge i \in [m]\}$ ;
5 $\quad k \leftarrow \min\{|Cache|, |AM|\}$ ;
6 $\quad$ **if** $|Cache| > 0$ **and** $|AM| > 0$ **then**
7 $\quad\quad$ **if** $|Cache| \leq |AM|$ **then**
8 $\quad\quad\quad CacheTmp \leftarrow$ sort $Cache$ in descending order of $p_j$ ;
9 $\quad\quad\quad$ **for** $l \leftarrow 1$ **to** $|CacheTmp|$ **do**
10 $\quad\quad\quad\quad TL(AM(l)) \leftarrow TL(AM(l)) \cup CacheTmp(l)$, $L(AM(l)) \leftarrow t + CacheTmp(l, 2)$ ;
11 $\quad\quad\quad Cache \leftarrow \emptyset$ ;
12 $\quad\quad$ **else**
13 $\quad\quad\quad CacheP \leftarrow PA\,(Cache, \mathcal{P}, |AM|)$ ;
14 $\quad\quad\quad$ **for** $l \leftarrow 1$ **to** $|AM|$ **do**
15 $\quad\quad\quad\quad TL(AM(l)) \leftarrow TL(AM(l)) \cup CacheP(l, 1)$, $L(AM(l)) \leftarrow t + CacheP(l, 1).p$ ;
16 $\quad\quad\quad\quad Cache \leftarrow Cache \backslash \{CacheP(l, 1)\}$ ;
17 $\quad t \leftarrow t + 1$ ;
18 **return** $TL$ *and* $C_{\max} = \max_{i \in [m]} \{L(i)\}$;

By replacing the general $PA$ in the step 13 of Algorithm 5 by the $PA1_\alpha$, $PA2_\alpha$, and $PA_\beta$ in Section 4.2, we can get three algorithms respectively namely $LPTCP1_\alpha$, $LPTCP2_\alpha$ and $LPTCP3_\beta$.

**Example 2.** Consider an instance of on-line load balancing problem with task buffer, the data is the same as Example 1.

With the $LPTC$ algorithm on Example 2, we can get a solution $X_{LPTC} = \{(1,1,0),\ (2,3,1),\ (3,2,1),\ (4,4,3),\ (5,2,6),\ (6,3,5),\ (7,4,9),\ (8,1,12),\ (9,1,8)\}$, $C_{\max} = 14$. While by the $LPTCP1_{\alpha=0.4}$ algorithm, we can get a solution $X_{LPTCP1_{\alpha=0.4}} = \{(1,1,0),\ (2,3,1),\ (3,2,1),\ (4,4,3),\ (5,2,6),\ (6,3,5),\ (7,1,8),\ (8,1,11),(9,4,9)\}$, $C'_{\max} = 13$. The results of this example are shown in Figure 3.

From Figure 3, we can see that the difference between the load of each machine is smaller than the traditional *Greedy*, *Semi-Greedy*, and $LPT$ algorithms depicted




a) *LPTC* b) $LPTCP1_{\alpha=0.4}$

Figure 3. The results obtained by *LPTC* and $LPTCP1_{\alpha=0.4}$

in Figure 1. Furthermore, the $LPTCP1_{\alpha=0.4}$ algorithm performs best of all the algorithms.

## 6 EXPERIMENTAL RESULTS

To show the correctness and effectiveness of the above proposed algorithms, following simulations are performed.

As we have mentioned in Section 5, all existing algorithms for the general on-line load balancing problem cannot be used directly to solve the problem of on-line load balancing with task buffer. So in the following simulations, we have not compare with them. We implement the *Greedy*, *GreedyC*, $Semi\text{-}Greedy_r$, $Semi\text{-}GreedyC_r$, *LPT*, *LPTC*, $LPTCP1_\alpha$, $LPTCP2_\alpha$, and $LPTCP3_\beta$ algorithms in Matlab R2014a and run them on a PC with 3.2 GHz Intel CPU and 2 GB RAM. Because there is no standard test experimental dataset for the problem of on-line load balancing with task buffer, we use 100 randomly generated instances as the testing dataset. The method of generating instances is similar to [13], specifically as follows:

- 10 instances were generated for each pair of task and machine size of $20 \times 5$, $50 \times 5$, $100 \times 5$, $20 \times 10$, $50 \times 10$, $100 \times 10$, $200 \times 10$, $100 \times 20$, $200 \times 20$ and $500 \times 20$, respectively,
- processing time $p_j \in \{1, \ldots, 99\}$ is randomly assigned to each task $j \in [100]$,
- release time $r_j \in \left[1, \left\lceil (\sum_{j=1}^{n} p_j)/(2.5 \times m) \right\rceil\right]$ is randomly assigned to each task $j \in [100]$.

In order to determine the parameter values for the $Semi\text{-}Greedy_r$, $Semi\text{-}GreedyC_r$, $LPTCP1_\alpha$, $LPTCP2_\alpha$, and $LPTCP3_\beta$ algorithms, we run these algorithms respectively with different parameter settings on the above 10 instances with the size of $100 \times 20$, and compare the average makespan on each parameter value to find the appropriate parameter setting.

Take the $LPTCP2_\alpha$ algorithm for example, in Figure 4 the trend of the makespan values and running time values are depicted by varying the value of $\alpha$. From Figure 4, we can see that when $\alpha = 0.06$, 0.16, 0.22, and 0.57, the average makespans are 254.7,

254.8, 254.8, and 256.2, the running times are 0.007088, 0.006875, 0.006617, and 0.004629, respectively. The parameter selection criteria is the makespan efficiency privileged over the running time, because the objective of the problem of on-line load balancing with task buffer is make-span. Therefore, the appropriate value of $\alpha$ is set to 0.22 for the $LPTCP2_\alpha$ algorithm.

Similarly, $r$ is set to $0.3n$ for the $Semi\text{-}Greedy_r$ and $Semi\text{-}GreedyC_r$ algorithms. $\alpha$ is set to 0.44 for the $LPTCP1_\alpha$ algorithm. $\beta$ is set to 1.21 for the $LPTCP3_\beta$ algorithm.

**Remark 3.** The method used to determine the parameter values may not be good, because it only uses the 10 middle scale instances in the generated testing dataset with the size of $100 \times 20$. If the parameter can be adaptively selected according to the information of each instance, the result may be better.

For each pair of instances, we compare each algorithm according to the average relative percent deviation ($ARPD$) of the makespans, the number of instances with minimum load, and the average running time. The $ARPD$ for the $k \in [10]$ pair instances is calculated as follows:

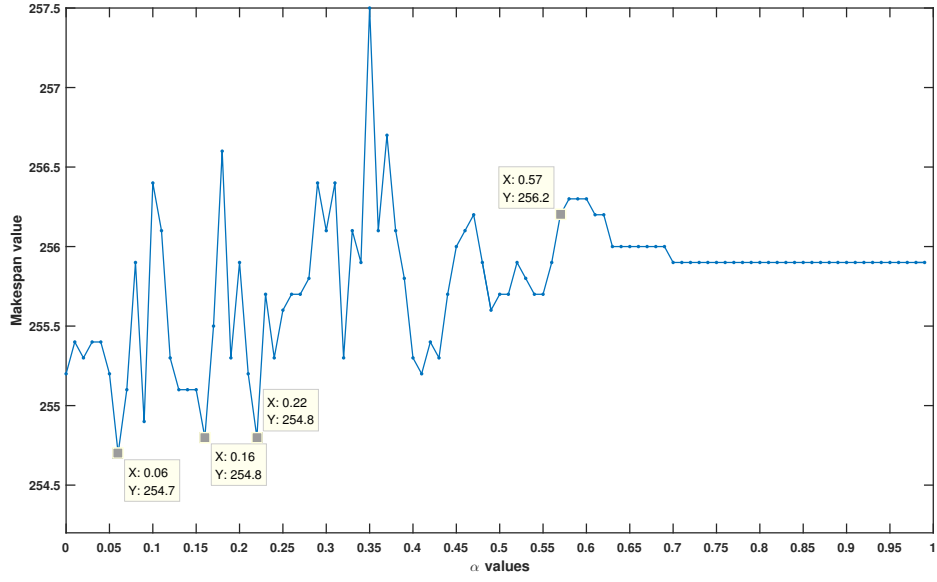
$$ARPD_k = \sum_{j=(k-1)\times 10+1}^{k\times 10} \left( \frac{(L_{j,i} - L_j^*) \times 100}{L_j^*} \right) /10 \tag{3}$$

where $L_{j,i}$ is the makespan of instance $j$ obtained by algorithm $i \in S$, $S = \{Greedy, GreedyC, Semi\text{-}Greedy_{r=0.3n}, Semi\text{-}GreedyC_{r=0.3n}, LPT, LPTC, LPTCP1_\alpha, LPTCP2_\alpha, LPTCP3_\beta\}$, $L_j^* = \min\{L_{j,i}\}$. Experimental results are shown in Tables 2 to 4. From the experimental results, the following conclusions can be drawn.

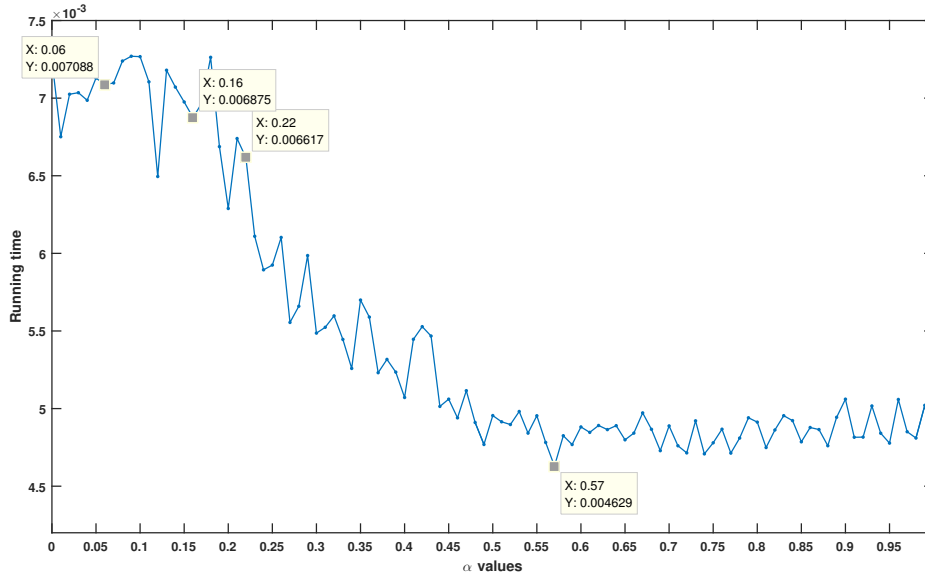
| Algorithms \ Scale of instances | $20\times5$ | $50\times5$ | $100\times5$ | $20\times10$ | $50\times10$ | $100\times10$ | $200\times10$ | $100\times20$ | $200\times20$ | $500\times20$ | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Greedy* | 10.9699 | 5.9238 | 2.8575 | 15.5002 | 13.8477 | 7.5810 | 4.2491 | 16.6895 | 8.3355 | 3.4412 | 8.9395 |
| *GreedyC* | 10.9699 | 5.9238 | 2.8575 | 15.5002 | 13.8477 | 7.5810 | 4.2491 | 16.6895 | 8.3355 | 3.4412 | 8.9395 |
| $Semi\text{-}Greedy_{r=0.3n}$ | *13.7129* | *8.2237* | *3.1356* | *22.7205* | *14.3397* | *8.3331* | *4.8502* | *20.4622* | *9.9668* | *4.2616* | ***11.0006*** |
| $Semi\text{-}GreedyC_{r=0.3n}$ | 10.9699 | 5.9238 | 2.8575 | 15.5002 | 13.8477 | 7.5810 | 4.2491 | 16.6895 | 8.3355 | 3.4412 | 8.9395 |
| *LPT* | 10.3302 | 5.7193 | 2.9720 | 14.2901 | 14.0386 | 7.3798 | 4.1546 | 16.5322 | 8.0858 | 3.2805 | **8.6783** |
| *LPTC* | 0.0000 | 0.2430 | 0.1037 | 0.0000 | 0.8864 | 0.2921 | 0.0607 | 0.7135 | 0.2123 | 0.0237 | **0.2535** |
| $LPTCP1_{\alpha=0.44}$ | 0.0000 | 0.1540 | 0.0960 | 1.8182 | 0.5333 | 0.2135 | 0.0306 | 0.6684 | 0.1755 | 0.0314 | **0.3721** |
| $LPTCP2_{\alpha=0.22}$ | 0.0000 | 0.2992 | 0.0655 | 0.1695 | 0.5560 | 0.1726 | 0.0407 | 0.2816 | 0.0992 | 0.0155 | **0.1700** |
| $LPTCP3_{\beta=1.21}$ | 0.0000 | 0.1429 | 0.0397 | 0.1695 | 0.5916 | 0.2604 | 0.0091 | 0.6323 | 0.1736 | 0.0000 | **0.2019** |

Table 2. The average relative percent deviation ($ARPD$)

1. In terms of experimental results of the average relative percent deviation ($ARPD$), our proposed algorithms, namely $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$, generally achieve better performances than all other algorithms. The trend of the $ARPD$ on the number of tasks equal to 100 is shown in the Figure 5. From Figure 5, we can see that when the amount of task is a fixed number, the more number of machines, the larger value of $ARPD$. However, the impacts on the $LPTC$, $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms are relatively tiny.




a) The trend of the makespan




b) The trend of the running time

Figure 4. The trends of the makespan and running time of the $LPTCP2_\alpha$ algorithm

| Algorithms \ Scale of instances | 20 × 5 | 50 × 5 | 100 × 5 | 20 × 10 | 50 × 10 | 100 × 10 | 200 × 10 | 100 × 20 | 200 × 20 | 500 × 20 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Greedy* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *GreedyC* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *Semi-Greedy*$_{r=0.3n}$ | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *Semi-GreedyC*$_{r=0.3n}$ | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* | *0* |
| *LPT* | *0* | *0* | *0* | 1 | *0* | *0* | *0* | *0* | *0* | *0* | **1** |
| *LPTC* | 10 | 5 | 4 | 10 | 1 | 3 | 5 | 2 | 4 | 7 | **51** |
| *LPTCP1*$_{\alpha=0.44}$ | 10 | 6 | 2 | 9 | 4 | 3 | 7 | 3 | 5 | 6 | **55** |
| *LPTCP2*$_{\alpha=0.22}$ | 10 | 4 | 5 | 9 | 6 | 5 | 6 | 5 | 7 | 8 | **65** |
| *LPTCP3*$_{\beta=1.21}$ | 10 | 7 | 7 | 9 | 3 | 3 | 9 | 5 | 4 | 10 | **67** |

Table 3. The amount of instances equal to the minimum load obtained by each algorithm

| Algorithms \ Scale of instances | 20 × 5 | 50 × 5 | 100 × 5 | 20 × 10 | 50 × 10 | 100 × 10 | 200 × 10 | 100 × 20 | 200 × 20 | 500 × 20 | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Greedy* | *0.0014* | *0.0028* | 0.0063 | 0.0012 | 0.0028 | 0.0053 | *0.0104* | *0.0050* | 0.0135 | *0.0250* | 0.0074 |
| *GreedyC* | 0.0015 | 0.0031 | 0.0064 | *0.0011* | 0.0029 | 0.0058 | 0.0116 | *0.0050* | 0.0103 | 0.0270 | 0.0075 |
| *Semi-Greedy*$_{r=0.3n}$ | *0.0014* | *0.0028* | *0.0054* | 0.0012 | *0.0027* | *0.0052* | *0.0104* | 0.0051 | *0.0100* | 0.0252 | *0.0069* |
| *Semi-GreedyC*$_{r=0.3n}$ | 0.0021 | 0.0049 | 0.0101 | 0.0013 | 0.0034 | 0.0070 | 0.0142 | 0.0054 | 0.0111 | 0.0289 | 0.0088 |
| *LPT* | 0.0022 | 0.0055 | 0.0096 | 0.0015 | 0.0042 | 0.0083 | 0.0163 | 0.0070 | 0.0143 | 0.0471 | **0.0116** |
| *LPTC* | 0.0030 | 0.0068 | 0.0135 | 0.0017 | 0.0048 | 0.0095 | 0.0187 | 0.0070 | 0.0146 | 0.0375 | **0.0117** |
| *LPTCP1*$_{\alpha=0.44}$ | 0.0017 | 0.0057 | 0.0160 | 0.0013 | 0.0052 | 0.0154 | 0.0539 | 0.0173 | 0.0569 | 0.3788 | **0.0552** |
| *LPTCP2*$_{\alpha=0.22}$ | 0.0017 | 0.0048 | 0.0123 | 0.0012 | 0.0046 | 0.0116 | 0.0390 | 0.0129 | 0.0377 | 0.2092 | **0.0335** |
| *LPTCP3*$_{\beta=1.21}$ | 0.0016 | 0.0044 | 0.0105 | 0.0012 | 0.0042 | 0.0102 | 0.0277 | 0.0104 | 0.0271 | 0.1251 | **0.0222** |

Table 4. The average running time of each algorithm (Unit: second)

The trends of the *ARPD* on different number of machines is shown in the Figure 6. From Figure 6, we can see that when the amount of machine is a fixed number, the more number of tasks, the smaller value of *ARPD*. Comprehensive analysis of Figures 5 and 6 shows that along with the increasing ratio of the number of tasks and the number of machines, the better *ARPD* can be obtained by these algorithms.

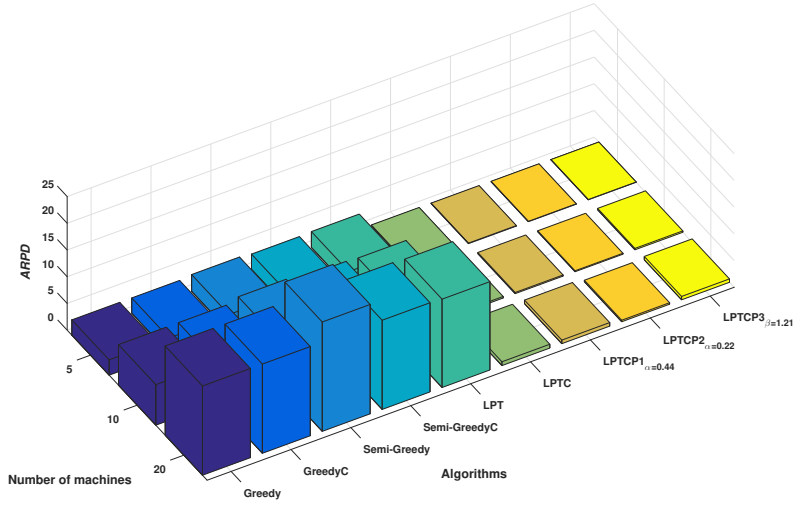
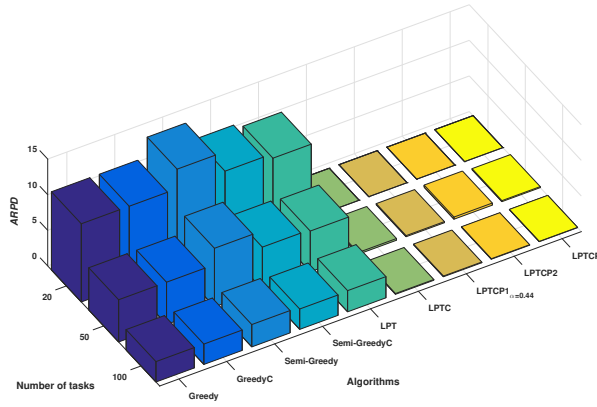
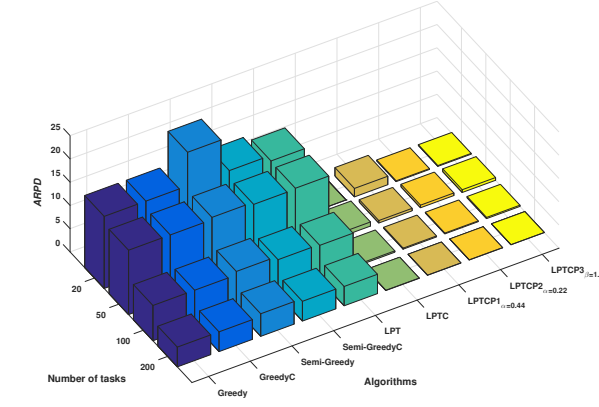


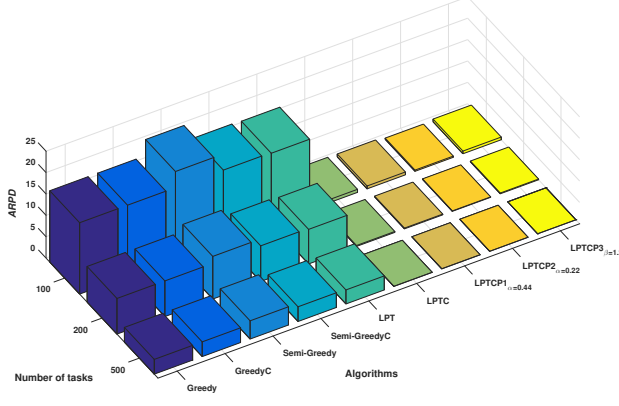

Figure 5. The trend of the *ARPD* on the number of tasks equal to 100




a) The number of machines is 5




b) The number of machines is 10




c) The number of machines is 20

Figure 6. The trends of the *ARPD* on different number of machines

Note that the performance of the *Greedy* and *GreedyC* are the same. Even though the *GreedyC* algorithm has a buffer, it allocates each task according to its release time and does not change the assignment relationship between tasks and machines. Comparing the experimental performances of $Semi\text{-}GreedyC_{r=0.3n}$ to $Semi\text{-}Greedy_{r=0.3n}$ and $LPTC$ to $LPT$, it is obvious that adding a buffer is useful. Especially, the $LPTC$ algorithm is significantly better than the $LPT$ algorithm in any case. Furthermore, the $ARPD$s obtained by $LPTC$, $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ have little differences between each other, and the $LPTCP2_{\alpha=0.22}$ algorithm achieves the best performance. Experimental performances of the $LPTC$, $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms measured by $ARPD$ are shown in the Figure 7.

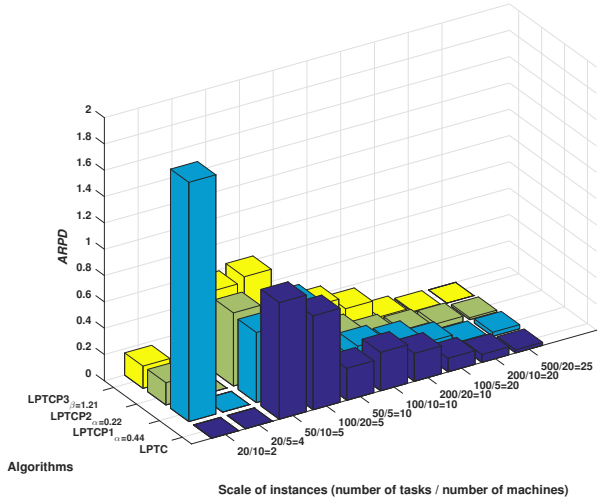
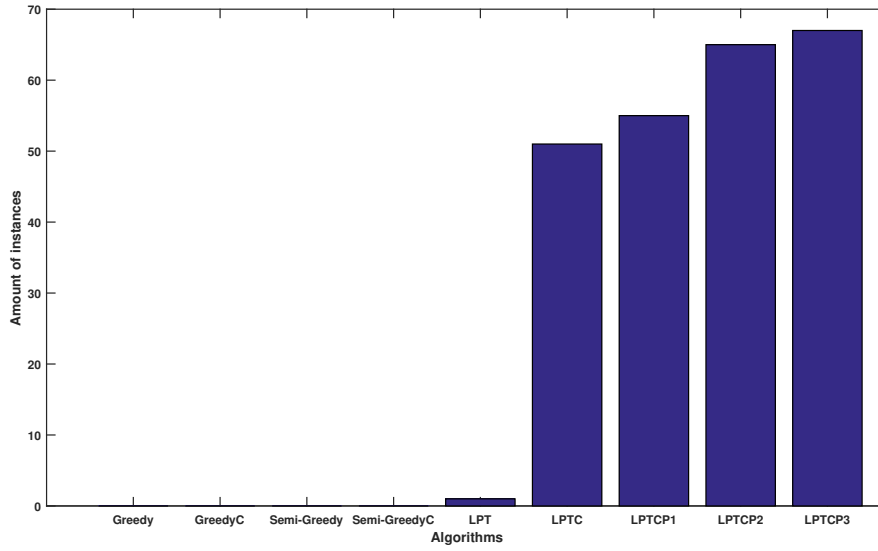
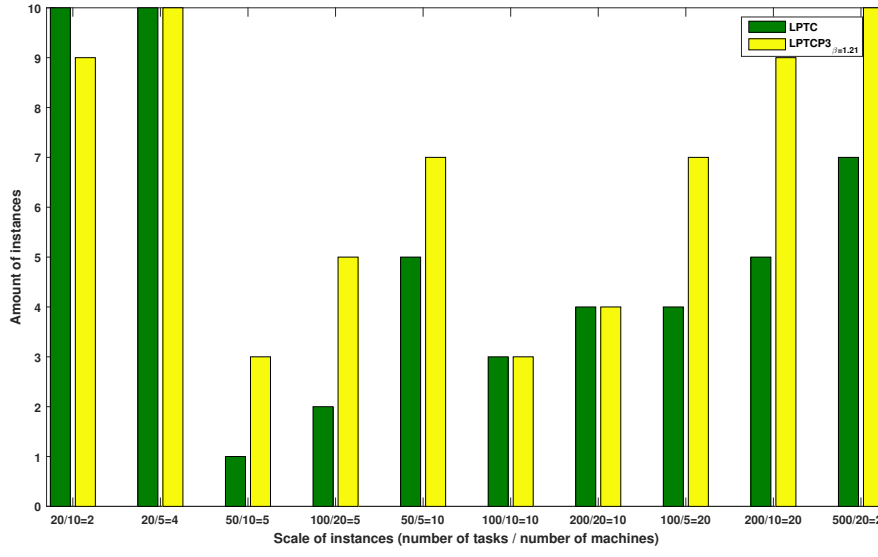



Figure 7. Experimental performances of the $LPTC$, $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms measured by $ARPD$

2. In terms of the number of instances with the minimum load, our proposed algorithms generally achieve better performance compared with all other algorithms. About 50 % of instances are with minimum load for the $LPTC$, $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms. The $LPTCP3_{\beta=1.21}$ algorithm beats the others, producing the minimum load results that is up to 67 %. Experimental results on the number of instances with minimum load of all algorithms are depicted in Figure 8. From Figure 8, we can see that along with the increasing ratio of the number of tasks and the number of machines, the number of instances with minimum load obtained by the $LPTCP3_{\beta=1.21}$ algorithm is higher than by the $LPTC$ algorithm.
3. In terms of the average running time, our proposed algorithms generally require longer running time compared with all other algorithms. The trend of the av-




a) All above algorithms




b) The $LPTC$ and $LPTCP3_{\beta=1.21}$ algorithms

Figure 8. Experimental performances of different algorithms measured by the number of instances with minimum load

erage running time on the number of tasks equal to 100 is shown in Figure 9. From Figure 9, we can see that when the amount of task is a fixed number, except for the $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms, the more number of machines, the less running time needed for processing. At the same time, the average running time of our proposed algorithms has not changed much.

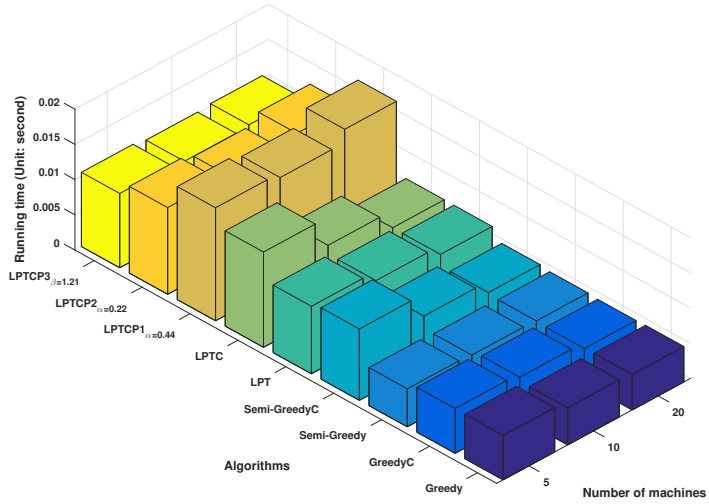
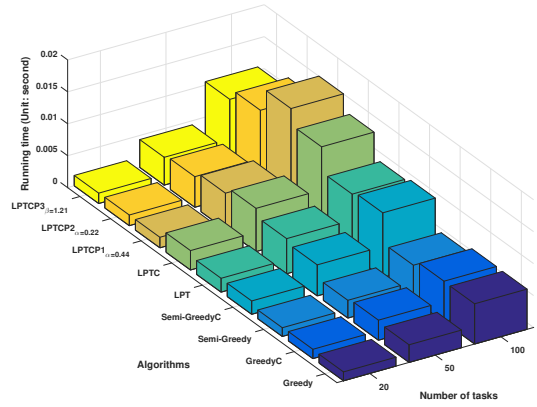



Figure 9. The trend of the average running time on the number of tasks equal to 100

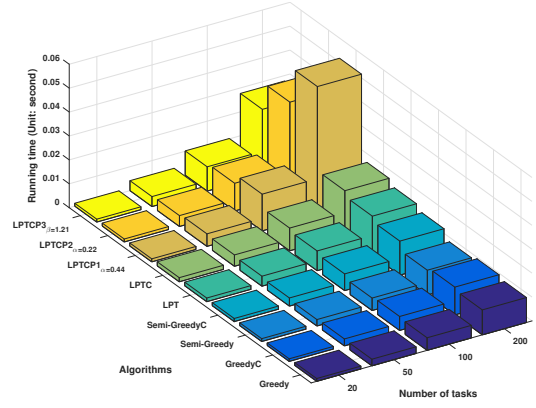
In Figure 10, the trends of the average running time on different number of machines are depicted. From Figure 10, we can see that when the amount of machine is a fixed number, the more number of tasks, the longer running time needed for processing.

Note that the times required for the *Greedy*, *GreedyC*, $Semi\text{-}Greedy_{r=0.3n}$, and $Semi\text{-}GreedyC_{r=0.3n}$ are similar to each other. Their total average running times are all smaller than 0.01 s. The total average running times of the *LPT* and *LPTC* are slightly larger than 0.01 s. The $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$ and $LPTCP3_{\beta=1.21}$ algorithms take slightly longer time for processing. Their total average running times are 0.0552 s, 0.0335 s and 0.0222 s, respectively. However, our proposed algorithms are still practical, even for the largest scale experiment instances with 500 tasks and 20 machines, their running times are less than 0.4 second.

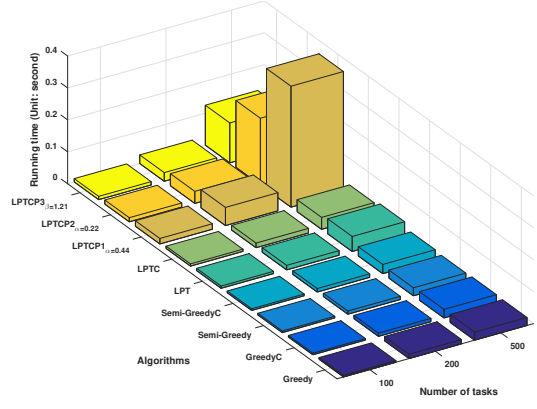
Experimental result on the average running time of the *LPTC*, $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms is depicted in Figure 11. The overall trend is the longer running time required along with the increasing ratio of the number of tasks and the number of machines. The average running times of the $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms are longer than those of the *LPTC* algorithm, because they are all designed based on the *LPTC*




a) The number of machines is 5




b) The number of machines is 10




c) The number of machines is 20

Figure 10. The trends of the average running time on different number of machines

algorithm. The main difference between our proposed algorithms is in the planarization algorithm they adopted. Although our proposed algorithms have the same time complexity, the $LPTCP3_{\beta=1.21}$ algorithm using $PA_\beta$ as the planarization algorithm requires the shortest running time. It means that $PA_\beta$ is more efficient than $PA1_\alpha$ and $PA1_\alpha$.

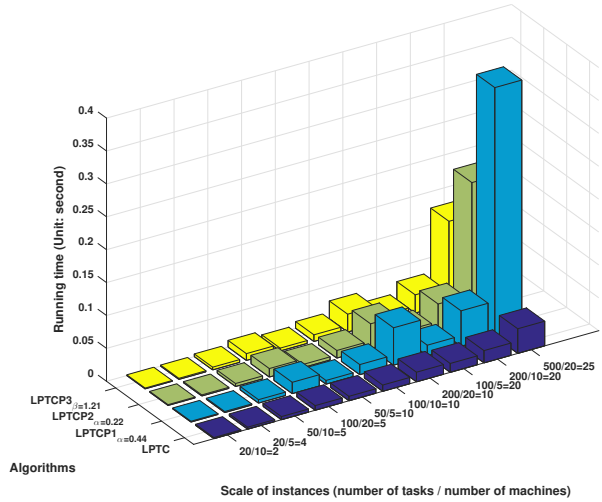



Figure 11. Experimental performances of the $LPTC$, $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms measured by the average running time

From the above analysis, the on-line load balancing with task buffer is superior to the general one. The main reason is that the buffer can defer making the decision of the task allocation until an idle machine occurs rather than the task arrives. The allocation decision can be made more reasonable with more information available. From experimental results, the $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ algorithms obviously outperform other algorithms.

## 7 CONCLUSIONS

In this paper, we have proposed a new model of on-line load balancing problem which has an unlimited sized task buffer, named as on-line load balancing with task buffer. We present the concepts of $\alpha$-flatting set and $\beta$-flatting set, the corresponding decision methods, and planarization techniques. Three algorithms are designed, namely $LPTCP1_\alpha$, $LPTCP2_\alpha$, and $LPTCP3_\beta$, for solving this problem based on the two new $\alpha$-planarization techniques, a $\beta$-planarization algorithm in [23], and the $LPT$ algorithm. Related simulation experiments are conducted to compare our proposed algorithms with the *Greedy*, *Semi-Greedy*, *LPT*, and their variants algorithms, namely *GreedyC*, *Semi-GreedyC*, and *LPTC*. Experimental results show

that the $LPTCP1_{\alpha=0.44}$, $LPTCP2_{\alpha=0.22}$, and $LPTCP3_{\beta=1.21}$ perform the best for the 100 instances.

An important issue for the further research topics is how to adaptively select the parameters of $\alpha$ and $\beta$. Parameter selection method used in this paper is based on solving a set of instances with the size of $100 \times 20$. This method has a strong limitation. Theoretically speaking, if it can adaptively select the appropriate parameters based on the information of each instance, it is possible to generate a better solution. In addition, the proposed model is built upon the assumption with unlimited sized task buffer, because the maximum size of the task buffer is equal to the number of tasks. Nevertheless, if we can estimate the size of the task buffer more accurately, it will make the model more realistic.

On the other hand, we paid more attention to the comparison and analysis of the numerical results, because it is very difficult to analyze the competitive ratios of our proposed algorithms. However, if the competitive ratios of our proposed algorithms are evaluated, it will contribute to the theoretical analysis of the proposed algorithms. This will be an important future research direction.

**Acknowledgement**

This work was supported by the National Nature Science Foundation of China (No. 61262006, No. 61540050, No. 61202089, No. 61462011), the National Research Foundation for Doctoral Program of Higher Education of China Project (No. 20125201120006), the Major Applied Basic Research Program of Guizhou Province (No. JZ20142001) and the Science and Technology Foundation of Guizhou Province (No. LH20147636).

## REFERENCES

[1] Mao, Y.—Chen, X.—Li, X.: Max-Min Task Scheduling Algorithm for Load Balance in Cloud Computing. Proceedings of International Conference on Computer Science and Information Technology, 2014, pp. 457–465, doi: 10.1007/978-81-322-1759-6_53.

[2] Ripoll, A.—Senar, M. A.—Cortés, A.—Luque, E.: Mapping and Dynamic Load-Balancing Strategies for Parallel Programming. Computers and Artificial Intelligence, Vol. 17, 1998, No. 5, pp. 481–491.

[3] García-Dopico, A.—Pérez, A.—Rodríguez, S.—García, M. I.: CYCLIC: A Locality-Preserving Load-Balancing Algorithm for PDES on Shared Memory Multiprocessors. Computing and Informatics, Vol. 31, 2012, No. 6, pp. 1255–1278.

[4] Cho, K.-M.—Tsai, P.-W.—Tsai, C.-W.—Yang, C.-S.: A Hybrid Meta-Heuristic Algorithm for VM Scheduling with Load Balancing in Cloud Computing. Neural Computing and Applications, Vol. 26, 2015, No. 6, pp. 1297–1309, doi: 10.1007/s00521-014-1804-9.

[5] Dasgupta, K.—Mandal, B.—Dutta, P.—Mandal, J. K.—Dam, S.: A Genetic Algorithm (GA) Based Load Balancing Strategy for Cloud Computing. Procedia Technology, Vol. 10, 2013, No. 2, pp. 340–347, doi: 10.1016/j.protcy.2013.12.369.

[6] Yildiz, M.: An Autonomous Load Balancing Framework for UCN. User-Centric Networking. Springer, 2014. pp. 241–265, doi: 10.1007/978-3-319-05218-2_12.

[7] Dam, S.—Mandal, G.—Dasgupta, K.—Dutta, P.: An Ant Colony Based Load Balancing Strategy in Cloud Computing. Advanced Computing, Networking and Informatics-Volume 2. Springer, Smart Innovation, Systems and Technologies, Vol. 28, 2014, pp. 403–413, doi: 10.1007/978-3-319-07350-7_45.

[8] De Falco, I.—Laskowski, E.—Olejnik, R.—Scafuri, U.—Tarantino, E.—Tudruj, M.: Load Balancing in Distributed Applications Based on Extremal Optimization. Applications of Evolutionary Computation (EvoApplications 2013). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7835, 2013, pp. 52–61, doi: 10.1007/978-3-642-37192-9_6.

[9] Bezek, A.—Gams, M.: Comparing a Traditional and a Multi-Agent Load-Balancing System. Computing and Informatics, Vol. 25, 2006, No. 1, pp. 17–42.

[10] Christodoulou, G.—Kovács, A.: A Deterministic Truthful PTAS for Scheduling Related Machines. SIAM Journal on Computing, Vol. 42, 2013, No. 4, pp. 1572–1595, doi: 10.1137/120866038.

[11] Garey, M. R.—Johnson, D. S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco, LA, Freeman, 1979.

[12] Azar, Y.: On-Line Load Balancing. Online Algorithms. Springer, 1998, pp. 178–195, doi: 10.1007/BFb0029569.

[13] Caramia, M.—Dell'Olmo, P.: On-Line Load Balancing. Effective Resource Management in Manufacturing Systems: Optimization Algorithms for Production Planning. Springer, 2006, pp. 35–64.

[14] Rudin III, J. F.—Chandrasekaran, R.: Improved Bounds for the Online Scheduling Problem. SIAM Journal on Computing, Vol. 32, 2003, No. 3, pp. 717–735.

[15] Fleischer, R.—Wahl, M.: Online Scheduling Revisited. Algorithms-ESA 2000. Springer, 2000, pp. 202–210, doi: 10.1007/3-540-45253-2_19.

[16] Divakaran, S.—Saks, M.: An Online Algorithm for a Problem in Scheduling with Set-Ups and Release Times. Algorithmica, Vol. 60, 2011, No. 2, pp. 301–315, doi: 10.1007/s00453-009-9337-9.

[17] Liu, P.—Lu, X.: Online Scheduling on Two Parallel Machines with Release Dates and Delivery Times. Journal of Combinatorial Optimization, Vol. 30, 2015, No. 2, pp. 347–359.

[18] Liu, W.-P.—Sidney, J. B.—Van Vliet, A.: Ordinal Algorithms for Parallel Machine Scheduling. Operations Research Letters, Vol. 18, 1996, No. 5, pp. 223–232.

[19] He, Y.—Zhou, H.—Jiang, Y. W.: Preemptive Semi-Online Algorithms for Parallel Machine Scheduling with Known Total Size. Acta Mathematica Sinica, Vol. 22, 2006, No. 2, pp. 587–594.

[20] Luo, T.—Xu, Y.: Semi-Online Hierarchical Load Balancing Problem with Bounded Processing Times. In: Gu, Q., Hell, P., Yang, B. (Eds.): Algorithmic Aspects in Infor-

mation and Management (AAIM 2014). Springer International Publishing, Lecture Notes in Computer Science, Vol. 8546, 2014, pp. 231–240.

[21] Albers, S.: New Results on Web Caching with Request Reordering. Algorithmica, Vol. 58, 2010, No. 2, pp. 461–477, doi: 10.1007/s00453-008-9276-x.

[22] Englert, M.—Özmen, D.—Westermann, M.: The Power of Reordering for Online Minimum Makespan Scheduling. SIAM Journal on Computing, Vol. 43, 2014, No. 3, pp. 1220–1237, doi: 10.1137/130919738.

[23] Wei, J.—Xu, D.—Qin, Y.—Zhou, J.: A Heuristic Algorithm for Solving the Problem of Load Balancing. 7th International Conference on Advanced Computational Intelligence, 2015, pp. 89–96.

[24] Graham, R. L.: Bounds for Certain Multiprocessing Anomalies. Bell System Technical Journal, Vol. 45, 1966, No. 9, pp. 1563–1581, doi: 10.1002/j.1538-7305.1966.tb01709.x.

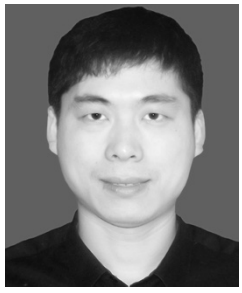
[25] Graham, R. L.: Bounds on Multiprocessing Timing Anomalies. SIAM Journal on Applied Mathematics, Vol. 17, 1969, No. 2, pp. 416–429, doi: 10.1137/0117039.

[26] Hart, J. P.—Shogan, A. W.: Semi-Greedy Heuristics: An Empirical Study. Operations Research Letters, Vol. 6, 1987, No. 3, pp. 107–114.

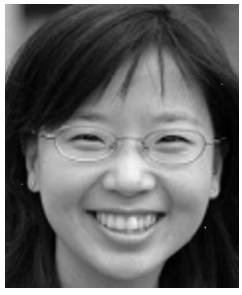
[27] Wei, J.—Qin, Y.—Xu, D.: A Scheduling Algorithm of $\alpha$ Planarization for Solving the Problem of Multiprocessor Scheduling. Computer Science, Vol. 39, 2012, No. 1, pp. 178–181.

**Jiayin Wei** received his B.Sc. degree in information and computational science from the Anhui University of Technology, China, in 2009 and the Ph.D. degree in the computer software and theory from the Guizhou University, China, in 2015. His research interests include algorithmic mechanism design, algorithm design and analysis, online algorithms and approximate algorithms.

**Daoyun Xu** is Professor of computer science and technology at the Guizhou University. He teaches several courses in computer sciences at graduate and postgraduate level at the Faculty of Computer Science and Technology. His research interests include computability and complexity of computing, algorithm design and analysis, etc.



**Yongbin Qin** received his B.Sc. degree in computer science from the Jinan University, China, in 2003 and the Mphil. and Ph.D. degrees in the computer science and technology from the Guizhou University, China, in 2007 and 2011. In 2007, he joined the Guizhou University, China as Lecturer. Since 2011, he has been Associate Professor at the Guizhou University. His research interests include intelligent computing, machine learning, and algorithm design.



**Ruizhang Huang** received her B.Sc. degree in computer science from the Nankai University, China, in 2001 and the Mphil. and Ph.D. degrees in the systems engineering and engineering management from the Chinese University of Hong Kong, Hong Kong, in 2003 and 2008. In 2007, she joined the Hong Kong Polytechnic University, Hong Kong, as Lecturer. Since 2011, she has been with the Guizhou University as Associate Professor. She is an active researcher in the area of data mining, text mining, machine learning, and information retrieval. She has published a number of papers including prestigious journals and conferences.