2016-ENST-0014

EDITE - ED 130

Doctorat ParisTech

THÈSE

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « INFORMATIQUE et RÉSEAUX »

présentée et soutenue publiquement par

Xiaohu Wu

le 16 février 2016

Techniques for Scheduling and Pricing in Cloud Computing

Directeur de thèse : **Patrick Loiseau**

Jury

M. Pietro Michiardi, Professeur, Eurecom, Sophia-Antipolis, France	Président & Examinateur
M. Denis Trystram, Professeur, Grenoble INP - Ensimag, France	Rapporteur
M. Klaus Jansen, Professeur, University of Kiel, Germany	Rapporteur
Mme. Johanne Cohen, Directeur de Recherche, CNRS, France	Examinateur

T H È S E

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

Dissertation

In Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
from Telecom ParisTech

Specialization : Computer Science

Xiaohu Wu

Techniques for Scheduling and Pricing in Cloud Computing

Successfully defended the 16th of February 2016 before a committee composed of :

President of the Jury		
Professor	Pietro Michiardi	Eurecom, Sophia Antipolis
Reviewers		
Professor	Denis Trystram	Grenoble Institute of Technology
Professor	Klaus Jansen	University of Kiel, Germany
Examiners		
Professor	Pietro Michiardi	Eurecom, Sophia Antipolis
Professor	Johanne Cohen	French National Center for Scientific Research
Thesis Supervisor		
Professor	Patrick Loiseau	Eurecom, Sophia Antipolis

Thèse

présentée pour l'obtention du grade de

Docteur de Télécom ParisTech

Spécialité : l'informatique

Xiaohu Wu

Techniques d'ordonnancement et de tarification dans le Cloud Computing

Soutenance de thèse prévue le 16 fevrier 2016 devant le jury composé de :

Président du jury		
Professeur	Pietro Michiardi	Eurecom, Sophia Antipolis
Rapporteurs		
Professeur	Denis Trystram	Grenoble INP - Ensimag
Professeur	Klaus Jansen	University of Kiel, Germany
Examinateurs		
Professeur	Pietro Michiardi	Eurecom, Sophia Antipolis
Directeur de Recherche	Johanne Cohen	Centre National de la Recherche Scientifique
Directeur de thèse		
Professeur	Patrick Loiseau	Eurecom, Sophia Antipolis

Abstract

Cloud computing is a model for enabling ubiquitous and on-demand access to a shared pool of configurable computing resources, and it has become the main paradigm for computing, storage and many other applications. This thesis addresses the problem of scheduling and pricing in cloud computing.

The first part of this thesis address the related scheduling problems. Given the capacity of a cloud, an important objective is to improve its resource utilization or efficiency so as to allow more tenants to be served, increasing the social welfare or its revenue. Due to the ubiquity of batch data processing in cloud computing, we consider a foundamental model in which a given set of batch tasks is scheduled on multiple identical machines and each task is specified by a value, a workload, a deadline and a parallelism bound. Within the parallelism bound, the speedup is linear and the number of machines allocated to a task can vary over time without affecting its workload, i.e., the speedup is linear. For this model, we first give two core results : the definition of an optimal state under which multiple machines could be utilized by a set of tasks with hard deadlines, and, an algorithm achieving such a state.

The optimal utilization state plays a key role in the design and analysis of scheduling algorithms (i) when several typical objectives are considered, such as social welfare maximization, machine minimization, and minimizing the maximum weighted completion time, and, (ii) when the algorithmic design techniques such as greedy and dynamic programming are applied to the social welfare maximization problem. As a result, we give four new or improved algorithms for the above problems. In addition, we also introduce a new class of tasks : the speedup is linear when a task is allocated a small number of machines, while the speedup decreases when it is allocated more tasks ; this class is an extension of the above task model. Further, we propose scheduling algorithms for makespan minimization or social welfare maximization.

In the second part of this thesis, from the point of view of users, we consider cost-efficient use of the computing resources from public clouds. Cloud providers offer on-demand and spot instances (virtual machines) to users. The former are sold at a fixed price and are always available. The latter are cheaper but sold through dynamic pricing ; hence, their availability is uncertain over time. As a result, a hybrid use of spot and on-demand instances is needed to guarantee the quality of service. Furthermore, a user may also have a limited amount of instances, called self-owned instances. In the instance allocation process, two underlying theoretical questions are well addressed : to be cost-efficient, what properties should be kept in the policy for allocating self-owned instances, and what policy can maximize the utilization of spot instances after self-owned instances are used up, thus escaping unnecessary consumption of costly on-demand instances. Based on this, we propose (near-)optimal parametric policies to allocating different types of instances among jobs ; further, we use the technique of online learning to infer the optimal configuration parameters, including one paramter characterizing the availability of spot instances. The effectiveness of the proposed policies is also validated by extensive simulations.

Acknowledgements

My supervisor Patrick Loiseau has been the one who first introduced me to the pricing problem in the cloud computing, which represents one core of this thesis, and I would like to thank him for having given me the opportunity and freedom to work on this very interesting problem. I think that academic research becomes fun only when you have the chance to work together with talented persones. I have had that chance and each of them has taught me a particular aspect of our work and for that I would like to thank them : Michela Chessa, Vijay Kamble, and Yifan Pi. I would also like to thank especially Professor Walrand for having welcomed me at the EECS department inside the UC Berkeley and having spent time to discuss with me some interesting problems on scheduling and pricing.

I would like to thank all the Eurecom's administration staff and especially to Audrey Ratier for taking care of the bureaucratic problems and making it easier for me. Your patience was always appreciated by me and I owe you my gratitudes.

I would like to thank Prof. Denis Trystram at Grenoble Institute of Technology, and Prof. Klaus Jansen at Kiel University who reviewed my thesis and gave me constructive comments. I would also like to thank Prof. Pietro Michiardi at Eurecom, and Dr. Johanne Cohen at CNRS who participated in my doctoral defense.

Finally, I would like to thank my friends and my family as they make life, exiting, fun, bright and full of trumpets and because without them there would be no point in working on this thesis.

Contents

Abstract i
Acknowledgements iii
Contents v
List of Figures ix
List of Tables xi

1 Overview **1**
1.1 Background 1
1.2 An Overview of the Thesis 2
1.3 Our Contributions 3

I Theoretical Foundation for Cloud Scheduling **5**

2 Scheduling in Cloud Computing **7**
2.1 Background 7
2.2 Outline of Results 8
2.3 Related Work 11
2.4 Problem Description 12

3 Optimal Schedule **15**
3.1 Optimal Resource Utilization State 15
3.2 Scheduling Algorithm 19
3.2.1 Overview of LDF($\mathcal{S}$) 20
3.2.2 Phase 1 24
3.2.3 Phase 2 29

4 Scheduling with Various Objectives **35**
4.1 Greedy Algorithm 35
4.1.1 The Executing Process of Greedy 36
4.1.2 A New Algorithmic Analysis 37
4.1.3 Optimal Algorithm Design 41
4.2 Dynamic Programming 48
4.3 Machine Minimization 50
4.4 Minimizing Maximum Weighted Completion Time 51

5 Other Scheduling Problems 53
5.1 Introduction 53
5.1.1 Background and Motivations 53
5.1.2 Algorithmic Results 56
5.2 Related Work 56
5.2.1 Makespan Minimization 56
5.2.2 A Unified Speedup Function 58
5.2.3 Social Welfare Maximization 59
5.3 Problem Description 59
5.4 Scheduling for Utilization 61
5.4.1 Parameter Identification and Task Classification 61
5.4.2 Example 65
5.4.3 Generic Schedule 68
5.5 Optimizing Objectives 72
5.5.1 Makespan 72
5.5.2 Scheduling to Maximize the Total Value 74

II Cost-Optimal Use of Cloud Computing 79

6 Introduction 81
6.1 Background 81
6.2 Challenges and Results 83
6.3 Related Work 84

7 Model, Assumption, and Objectives 87
7.1 Pricing Models in the Cloud 87
7.2 Jobs to be Processed 88
7.3 Operational Rules for Allocating Instances to Jobs 89
7.3.1 On-demand and spot instances 89
7.3.2 Self-owned instances 91
7.4 Scheduling Objectives 91
7.4.1 Proposed principles for cost efficiency 91
7.4.2 Decision variables 92

8 The Design of Near-Optimal Policies 95
8.1 Self-owned Instances 95
8.1.1 Challenge 95
8.1.2 Policy Design 96
8.1.3 Explanation 99
8.2 Spot and On-demand Instances 101
8.2.1 First phase 101
8.2.2 Second phase 105
8.3 Scheduling Framework 109
8.4 The Application of Online Learning 110

9 Evaluation **113**
9.1 Simulation Setups 113
9.2 Results 115

10 Concluding Remarks **119**
10.1 Conclusions 119
10.2 Outlook of the Future 120

A Résumé en français **133**
A Introduction 133
B Un aperçu de la thèse et des principaux résultats 134
B.1 Partie I: vue d'ensemble et résultats 134
B.2 Partie II: vue d'ensemble et résultats 137
C Partie I: l'ordonnancement de tâches malléables 139
C.1 Motivation 139
C.2 L'état optimal de utilisation machine 140
C.3 Algorithme glouton 142
D Partie I: l'ordonnancement de tâches (δ, k)-monotonic 143
D.1 Minimisation de makespan 144
D.2 Maximisation de bien-être social 145
E Partie II: utilisation rentable des clouds publics 145
E.1 Défis 145
E.2 Notre solution 146
F Remarques finales 148
F.1 Conclusion 148
F.2 Perspectives d'avenir 149

List of Figures

3.1 The green areas denote the maximum demand of T_i that need or could be processed in $[\tau_{L-m}+1, \tau_L]$. 16

3.2 Derivation from the definition $\lambda_m(\mathcal{S})$ to $\lambda_m^C(\mathcal{S})$. 17

3.3 The resource allocation state of T_i and $\mathcal{S}'$ respectively upon completion of Fully-Utilize(i), Fully-Allocate(i), and AllocateRLM(i, 1, t_2+1) where $L = m = 3$: on the C machines, the blue area denotes the allocation to $\mathcal{S}'$ that satisfies Properties C.1 and C.2 before executing Allocate-B(i) while the green area denotes the allocation to T_i. 22

3.4 Relationship among Algorithms: for $A \rightarrow B$, the blue and green arrows denote the relations that the algorithm A calls B, and, the algorithm B is executed upon completion of A. 33

5.1 Task Classification 65

5.2 Basic Scheduling Structure 67

6.1 Motivation of using IaaS clouds. 82

6.2 Our Question. 82

7.1 On-demand price: users are charged on an hourly basis. 87

7.2 Spot price: a user bid a price b for an instance at time 0 and can use it until time t. 88

7.3 Illustration of the process of allocating resource to j where $\delta_j = 4$, d_j equals 3 hours, $L = 5$ minutes and $z_j = 132$: the area between two red lines illustrates the workload processed at every allocation and the height of green (resp. yellow) areas denote the number of spot (resp. on-demand) instances. 90

8.1 The Challenge in Cost-Effectively Utilizing Self-owned Instances. 96

8.2 As x ranges in $[0, 1)$, the function $g_j(x)$ for jobs respectively with different flexibility to utilize spot instances. 99

8.3 As the (relative) deadline d_j increases from 12 to 48, the function $g_j(\beta_0)$ decreases respectively under $\beta_0 = \frac{31}{64}, \frac{5}{16}, \frac{1}{16}$, where $z_j = 240$, $\delta_j = 20$, and $Len = 12$. 100

8.4 Illustration of Proposition 22 in the case that $\nu(z_j, d_j) < (\kappa_0 - 1) \cdot \delta_j$ and $\kappa_2(z_j, d_j) < \kappa_3$. 104
8.5 The second phase of allocation where $i = i_j$: the orange area denotes the available space in the second phase; the green and yellow areas denote the workload processed at the i_j-th allocation by spot instances and on-demand instances that are utilized for an hour. 105
8.6 Illustration for Proposition 23: the yellow area denotes the allocation of on-demand instances to j. 106

9.1 The utilization of self-owned instances under different values of β_0. 117

List of Tables

2.1 Symbol Interpretation 14

3.1 Main Notation for Chapter 3 20

4.1 Main Notation for Chapter 4 38

5.1 The value of $\mu(\delta)$ when the value of δ is in different ranges. . . . 55
5.2 The values of β_1 and β_2 when δ is in different ranges. 72

7.1 Main Notation 92

9.1 Performance Improvements for Spot and On-Demand Instances . 115
9.2 The Optimal β under a Bid Price b 115
9.3 Performance Improvement for Self-Owned Instances 116
9.4 The Instance Utilization of the Proposed Policy under Cost-Optimal β_0 116
9.5 The Instance Utilization of the Intuitive Policy 116
9.6 Performance Improvement for Three Types of Instances 117
9.7 Performance Improvement under Online Learning 117

Chapter 1

Overview

There are three principal means of acquiring knowledge available to us: observation of nature, reflection, and experimentation. Observation collects facts; reflection combines them; experimentation verifies the result of that combination. Our observation of nature must be diligent, our reflection profound, and our experiments exact. We rarely see these three means combined; and for this reason, creative geniuses are not common.

Denis Diderot

1.1 Background

This thesis addresses the related problems of scheduling and pricing in cloud computing. Cloud computing is a model for enabling ubiquitous, on-demand access to a shared pool of configurable computing resources. Cloud computing and storage solutions provide users with various capabilities to store and process their data in third-party data centers. It relies on the sharing of resources to achieve coherence and economies of scale, similar to a utility (like the electricity grid) over a network. Hence, it focuses on maximizing the effectiveness of the shared resources. On the other hand, cloud computing allows users to avoid upfront infrastructure costs and enables IT to more rapidly adjust resources to meet fluctuating and unpredictable business demand. It has the advantages of high computing power, cheap cost of services, high performance, scalability, accessibility as well as availability and reduce the burden for users to deploy and manage these computing infrastructure. Cloud providers mostly use a fixed usage-based pricing, sometimes in association with a spot market to sell remaining resources through auctions. This can lead to higher charges if users do not adapt to the cloud pricing model. On the other hand, the design of appropriate scheduling and pricing schemes for cloud resources is central to the cloud computing field since utility maximization is often one of the most important concerns for cloud providers.

In the scheduling aspect, the following questions are always worth concerning. Given the fixed capacity of a cloud and the scheduling objectives such as satisfying the deadline requirement, how could the cloud serves as many tenants as possible or maximizes the sum of values of jobs completed by the deadlines, therefore increasing the revenue of cloud providers? In scheduling theory, these questions correspond to the machine minimization and social welfare maximization problems. Given the cloud capacity, how could the cloud optimize the scheduling objectives such as minimizing the maximum lateness of jobs? These scheduling problems are the key to the revenue management of cloud providers and the improvement of quality of service of users, and constitute the main questions to be addressed in this thesis. In the pricing aspect, the problem that concerns us is that, given the current pricing models such as the one in Amazon EC2, how could an organization can acquire computing resources from the cloud in a cost-effective way.

1.2 An Overview of the Thesis

In this section, we briefly introduce the structure of this thesis.

Part I. In the first part of this thesis, we lay some theoretical foundation on the problem of scheduling in cloud computing.

— Chapter 2

In this chapter, we introduce the importance of scheduling problems in cloud computing and the model of tasks to be considered in this scenario.

— Chapter 3

In this chapter, we identify the optimal resource utilization state of multiple machines on which a set of tasks with hard deadlines is to be scheduled; further, we propose a scheduling algorithm that can achieve such an optimal state.

— Chapter 4

The results in Chapter 3 provide a conceptual tool to enable proposing new analysis and design of algorithms and improving existing algorithms under various scheduling objectives. In particular, we obtain the following algorithmic results:

(i) an improved and optimal greedy algorithm for social welfare maximization (i.e., maximizing the sum of values of tasks completed by their

deadlines),

(ii) the first exact dynamic programming algorithm for social welfare maximization with a pseudo-polynomial time complexity,

(iii) an exact algorithm for machine minimization (i.e., minimizing the number of machines needed to produce a feasible schedule of a set of tasks such that each task is completed by its deadline),

(iv) an improved algorithm with the objective of minimizing the maximum weighted completion time.

— Chapter 5

In this chapter, we introduce a variant of the above type of tasks. Through algorithmic analysis, we propose scheduling algorithms respectively for makespan minimization and for maximizing the sum of values of tasks completed by a deadline.

Part II. In the second part of this thesis, we consider how to acquire computing resource from the cloud in a cost-effective way.

— Chapter 6

In this chapter, we introduce the pricing models in the current cloud market such as the ones in Amazon Elastic Cloud Compute (EC2); two purchase options are considered: cheap spot and costly on-demand instances (virtual machines). Users may also have their own instances, referred to as self-owned instances.

— Chapter 7

In this chapter, we propose (near-)optimal parametric policies to allocating different types of instances among jobs [1]. Facing the dynamic of spot instance's prices and the unknown statistics of job's characteristics, an online learning approach is applied to estimate the cost-optimal configuration parameters of policies.

— Chapter 8

In this chapter, we evaluate the effectiveness of the proposed parametric policies through extensive simulations, in particular that they achieve a cost reduction of up to 64.51% when spot and on-demand instances are considered and of up to 43.74% when self-owned instances are considered, compared to previously proposed or intuitive policies.

1.3 Our Contributions

We further elaborate the main contributions of this thesis. In the first part, we consider two types of tasks. The first type of tasks in Chapters 2-4 is assumed to be malleable and work-preserving, i.e., the number of machines assigned to a task can vary within a parallelism bound during the execution (which brings the

1. In this thesis, we use "jobs" and "tasks" interchangeably.

operation of preempting the execution of a task), and the task's workload does not increase with the number of machines assigned to it. The main contrition of this thesis for scheduling this type of tasks has been pointed out when we introduce Chapters 3-4 above.

In Chapter 5, motivated by recent benchmark studies, we introduce the notion of (δ, k)-*monotonic tasks* under which tasks are moldable and for every task T_j that is assigned p processors, we have that

(i) when p is small and ranges in $[1, \delta]$, its workload $D_{j,p}$ remains constant and the speedup is linear;

(ii) when p is large and ranges in $[\delta+1, k]$, the workload $D_{j,p}$ is non-decreasing as p increases while its execution time first decreases and then even begins to increase once p exceeds some threshold;

(i) the maximum number of processors that is allowed to assigned to T_j is k.

The (δ, k)-monotonic tasks are the second type of tasks we considered in this thesis; here, "moldable" means that a task can be allocated a flexible number of machines in $[1, k]$; however once specified before the task's execution this number cannot change throughout its execution. For the second type of tasks, we propose a scheduling algorithm to minimize the makespan, whose performance depends on the value of δ; in realistic scenarios, δ can ranges from 5 to 64 and its approximation ratio approximately ranges from $\frac{4}{3}$ to $\frac{11}{10}$; here, given any input of tasks, if the approximation ratio of an algorithm is ρ, the makespan that this algorithm achieves is always no greater than ρ times the makespan of an optimal algorithm. As a by-product, we also propose a scheduling algorithm to maximize the sum of values of tasks completed by a deadline.

In the second part of this thesis, we consider the problem of cost-effectively utilize self-owned instances and the instances from public clouds. In the instance allocation process, two underlying theoretical questions are well addressed: to be cost-efficient, what properties should be kept in the policy for allocating self-owned instances, and what policy can maximize the utilization of spot instances after self-owned instances are used up, thus escaping unnecessary consumption of costly on-demand instances. Based on this, we propose (near-)optimal parametric policies to allocating different types of instances among jobs. The effectiveness of the proposed policies is also validated by extensive simulations. In this thesis, the jobs to be processed are assumed to be independent malleable jobs. It is worth noting that the two theoretical questions will also be a key to the extension of the results of this thesis to the case for jobs with precedence constraints.

Part I

Theoretical Foundation for Cloud Scheduling

Chapter 2

Scheduling in Cloud Computing

The definition of terms is the beginning of wisdom.

Socrates

2.1 Background

Cloud computing has become the norm for a wide range of applications and batch processing acts as one of the most significant computing paradigms [1]. Applications such as web search index update, monte carlo simulations and big-data analytics require executing a new type of parallel tasks on clusters, termed *malleable tasks*. Two basic features of malleable tasks are about *workload* and *parallelism bound*. There are multiple machines, and, throughout the execution, the number of machines assigned to a task can vary over time within the parallelism bound but its workload keeps constant regardless of the number of used machines [2], [3]; here, the offline batch scheduling problems with independent tasks are considered. Based on this, many efforts are further devoted to its online version [4]–[6] and its extension in which each task contains several subtasks with precedence constraints [7], [8]. In practice, for better efficiency, companies such as IBM have integrated these smart scheduling algorithms for various time metrics into their batch processing platforms [8], [9].

In scheduling theory, the above task model can be viewed as an extension of the classic model of preemptive tasks that can only be executed on one machine, i.e., their parallelism bound is one. In the past, the related problems of scheduling preemptive tasks on a single machine have been extensively studied [10], [11]. When each task has to be completed by some deadline, the previous results from the special single machine case have already implied that the state of optimally utilizing machines plays a key role in the design and analysis of scheduling algorithms for several objectives [11]. In particular, the famous EDF (Earliest Deadline First) rule can achieve the optimal resource utilization state, that is, given a set of tasks, if there is a feasible schedule of these tasks with deadlines on a machine, the EDF rule can always produce a feasible schedule.

Many applications of the EDF rule have been found to design, e.g., (i) a schedule that achieves the optimal resource utilization state where each task is additionally associated with a release time [12], (ii) an exact algorithm for minimizing the maximum task lateness (i.e., task's completion time minus due date) [13], and (iii) an exact algorithm for tasks with deadlines and release times to minimize the total weighted number of tardy tasks (i.e., tasks missing their deadlines) [14]. Moreover, the EDF rule is also fundamental in real-time systems where the scheduling feasibility is analyzed [15].

Similarly, we believed that, a schedule that achieves such an optimal resource utilization state is also fundamental in scheduling malleable tasks and can benefit the design and analysis of scheduling algorithms (i) under various objectives, or (ii) when different algorithmic design techniques such as greedy and dynamic programming are applied for the same objective. Here, the underlying intuition is that, if the resource utilization state is not optimal in an algorithm, its performance can be improved by utilizing the machines optimally, allowing more tasks to be completed or reducing the overall completion times of tasks. All these considerations motivated us to develop a corresponding theoretical framework in this thesis.

We note that, before our work, Jain *et al.* considered scheduling independent malleable tasks to maximize the social welfare (i.e., the sum of values of tasks completed by their deadlines) and proposed a greedy algorithm that achieves a performance guarantee $\frac{C-k}{C} \cdot \frac{s-1}{s}$ [3]; here, C is the number of machines, k is the maximum parallelism bound of all tasks, s is the minimum slackness of all tasks where each task's slackness is defined to be the ratio of its deadline to its minimum execution time (when a task is always allocated the maximum number of machines throughout the execution). Intuitively, s characterizes the resource allocation urgency or flexibility, e.g., $s = 1$ means that, in order not to miss the deadline, a task has to always utilize the maximum number of machines it could utilize from the beginning to its deadline. k is a system parameter and is assumed to be finite [16].

2.2 Outline of Results

In the following, we summarize the main results of this thesis for scheduling malleable tasks [17], [18].

Core result (Chapter 3). Our core result is to identify a sufficient and necessary condition under which a set of independent malleable tasks could be all completed by their deadlines on C machines, referred to as boundary condition.

In particular, by understanding the basic constraints when scheduling malleable tasks, we identify and define a state in which C machines can be said to be optimally utilized by a set of tasks with deadlines in terms of resource utilization. Then, we propose an optimal scheduling algorithm LDF($\mathcal{S}$) (Latest Deadline First) that achieves such an optimal state. The LDF($\mathcal{S}$) algorithm has a polynomial time complexity of $\mathcal{O}(n^2)$ and is different from the EDF algorithm that gives an optimal schedule in the single-machine case. Here, the maximum deadline of tasks is assumed to be finitely bounded.

Applications (Chapter 4). The above core results have several applications to propose new or improved algorithmic design and analysis for different scheduling objectives. The scheduling objectives that are separately addressed in this thesis include:

(a) *social welfare maximization:* maximize the sum of values of tasks completed by their deadlines;

(b) *machine minimization:* minimize the number of machines needed to produce a feasible schedule for a set of tasks such that each task is completed by its deadline;

(c) *maximum weighted completion time minimization:* minimize the maximum weighted completion time of tasks.

Here, the first and third objectives have been studied in [2], [3], [8]. The second objective that involves the optimal utilization of machines has been considered for other types of tasks [19] but we are the first to consider it for malleable tasks. After applying the core results above, we obtain the following algorithmic results:

(i) an improved greedy algorithm with a performance guarantee $\frac{s-1}{s}$ for social welfare maximization with a time complexity $\mathcal{O}(n^2)$;

(ii) the first exact dynamic programming algorithm for social welfare maximization with a pseudo-polynomial time complexity $\mathcal{O}\left(\max\left\{n \cdot d^L \cdot C^L, n^2\right\}\right)$, where L is the number of deadlines, D and d are the maximum workload and deadline of tasks;

(iii) the first exact algorithm for machine minimization with a time complexity $\mathcal{O}(n^2, L \cdot n \cdot \log n)$;

(iv) a polynomial time $(1+\epsilon)$-approximation algorithm for maximum weighted completion time minimization.

The greedy algorithm of [3] and ours represent a class of greedy algorithms. In this class, the tasks are considered in the decreasing order of their marginal values (i.e., the ratio of a task's value to its size); then, if the task being considered could be fully completed by its deadline under the currently idle machines over time, it will be accepted and fully allocated according to a certain allocation algorithm; otherwise, it will be rejected. We further show that

— for social welfare maximization, $\frac{s-1}{s}$ is the best possible performance guarantee that this class of greedy algorithms could achieve.

— as a result, the proposed greedy algorithm of this thesis is the best possible among this class of greedy algorithms.

The second algorithm for social welfare maximization can work efficiently when L is small since its time complexity is exponential in L. However, this may be reasonable in a machine scheduling context. In scenarios like [7], tasks are often scheduled periodically, e.g., on an hourly or daily basis, and many tasks have a relatively soft deadline (e.g., finishing after four hours instead of three will not trigger a financial penalty). Then, the scheduler can negotiate with the tasks and select an appropriate set of deadlines $\{\tau_1, \tau_2, \cdots, \tau_L\}$, thereafter rounding the deadline of a task down to the closest τ_i ($1 \leq i \leq L$). By reducing L, this could permit to use the dynamic programming (DP) algorithm rather than GreedyRLM in the case where the slackness s is close to 1. With s close to 1, the approximation ratio of GreedyRLM approaches 0 and possibly little social welfare is obtained by adopting GreedyRLM while the DP algorithm can still obtain the almost optimal social welfare.

Technical Difference. The second algorithm can be viewed as an extension of the pseudo-polynomial time exact algorithm in the single machine case [10] that is also designed via the dynamic programming procedure. However, before our work, how to enable this extension to malleable tasks was not clear as indicated in [2], [3]. This is mainly due to the lack of a notion of the optimal state of machines being utilized by malleable tasks with deadlines and the lack of an algorithm that achieves such a state. In contrast, the optimal state in the single machine case can be achieved by the EDF algorithm. The core results of this thesis are the enabler of the application of a DP procedure to the scenario of malleable tasks.

The way of applying the core results to a greedy algorithm is less obvious and in the single machine case there is no corresponding algorithm to hint its role in algorithmic design. For the above class of greedy algorithms, we manage to give a new algorithm analysis, figuring out what resource allocation features of tasks can benefit and determine the performance of such a greedy algorithm. This analysis is an extended analysis of the greedy algorithm for the knapsack problem where some items are chosen to be packed into a knapsack and each item is defined by its size and value [20]; it does not rely on the dual-fitting technique on which the algorithm in [3] is built. Here, the problem could be viewed as an extension of the knapsack problem where each item has two additional dimensions of constraints: a time window in which an item could be placed, i.e., a deadline, and the maximum width of this window, i.e., a parallelism bound. Two of the most important algorithms in the knapsack problem are either based on the DP technique or of greedy type that also considers items by their marginal values [20]; we give in this thesis their counterparts in the scenario of malleable tasks.

In the construction of the greedy and optimal scheduling algorithms, we are inspired by the algorithm in [3]. After our definition of the optimal resource utilization state and a new analysis of the above class of greedy algorithms, we found that the greedy algorithm in [3] could achieve such an optimal state from the maximum deadline of tasks d to some earlier time slot t. However, this is achieved by guaranteeing the existence of a time slot t' earlier than t such that the number of available machines at t' is $\geq k$, which leads a suboptimal utilization of resources in some time interval. In our algorithm, we only require t' to be such that the number of available machines at t' is ≥ 1, which finally leads to an optimal resource utilization. More details could be found in the remarks of Chapter 3.2.

The above third and fourth algorithms are obtained by respectively applying the above core results to a binary search procedure and the related results in [8].

2.3 Related Work

Now, we introduce the related works.

In this thesis, we consider the basic scenario of offline scheduling independent malleable tasks [2], [3]; its variants have also been considered so far [4], [5], [7]. In these previous works, the linear programming approach is applied to design and analyze algorithms where the objective of maximizing the social welfare is addressed [1]. In [2], Jain *et al.* proposed an algorithm with an approximation ratio $\frac{1}{\alpha}$ via *deterministic rounding of linear programming* where $\alpha = \left(1 + \frac{C}{C-k}\right) \cdot (1+\epsilon)$. Subsequently, Jain *et al.* [3] proposed a greedy algorithm GreedyRTL and used the *dual-fitting technique* to derive an approximation ratio $\frac{C-k}{C} \cdot \frac{s-1}{s}$. In [7], Bodik *et al.* considered an extension of the basic scenario of this thesis, i.e., DAG-structured malleable tasks, and, based on *randomized rounding of linear programming*, they proposed an algorithm with an expected approximation ratio $\frac{1}{\alpha(\lambda)}$ for every $\lambda > 0$, where $\alpha(\lambda) = \frac{1}{\lambda} \cdot e^{-\frac{1}{\lambda}} \cdot \left[1 - e^{-\frac{(1-1/\lambda)C-k}{2\omega\kappa} \cdot \ln\lambda \cdot (1-\frac{\kappa}{C})}\right]$. The online version of our scenario is considered in [4], [5] where independent tasks arrive over time. Again based on the *dual-fitting technique*, two weighted greedy algorithms are proposed respectively for non-committed and committed scheduling: the first algorithm achieves a competitive ratio $\frac{1}{\beta(s)}$ where $\beta(s) = 2 + \mathcal{O}\left(\frac{1}{(\sqrt[3]{s}-1)^2}\right)$ and $s > 1$ [3]; the second algorithm achieves a competitive ratio $\frac{1}{\beta'(s)}$ where $\beta'(s) = \frac{\beta(s\cdot\omega\cdot(1-\omega))}{\omega\cdot(1-\omega)}$, $\omega \in (0,1)$, and $s > \frac{1}{\omega\cdot(1-\omega)}$.

Methodologically, the works [2]–[5], [7] formulated their problem as an Integer Programming (IP) and relax the IP to a relaxed linear programming (LP). The techniques in [2], [7] require to solve the LP to obtain a fractional optimal solution where a task might be partially executed; then they manage to round the fractional solution to an integer solution of the IP that corresponds to an approximate solution to their original problem. In [3]–[5], the dual fitting technique is applied and it first finds the dual of the LP and then construct a

1. We refer readers to [11], [21] for more details on the general techniques to design scheduling algorithms.

feasible algorithmic solution X to the dual in some greedy way. This solution corresponds to a feasible solution Y to their original problems, and, due to the weak duality, the value of the dual under the solution X (expressed in the form of the value under Y multiplied by a parameter $\alpha \geq 1$) will be an upper bound of the optimal value of the IP, i.e., the optimal value that can be achieved in the original problem. Hence, the approximation ratio of the algorithm involved in the dual is $\frac{1}{\alpha}$; it is a lower bound of the ratio of the actual value obtained by the algorithm to the optimal value.

In addition, Nagarajan *et al.* [8] considered DAG-structured malleable tasks and proposed two algorithms with approximation ratios of 6 and 2 respectively for the objectives of minimizing the total weighted completion time and the maximum weighted lateness of tasks. Nagarajan *et al.* showed that *optimally scheduling deadline-sensitive malleable tasks in terms of resource utilization is a key to the solutions to scheduling for their objectives.* In particular, seeking a schedule for DAG tasks can be transformed into seeking a schedule for tasks with simpler chain-precedence constraints; then whenever there is a feasible schedule to complete a set of tasks by their deadlines, Nagarajan *et al.* proposed a non-optimal algorithm where each task is completed by at most 2 times its deadline and give two procedures to obtain near-optimal completion times of tasks in terms of the above two objectives.

2.4 Problem Description

Now, we formally describe the task model and the objectives of scheduling tasks on machines.

There are C machines and a set of n tasks, denoted by $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$. Every task T_i is specified by several characteristics: (1) *value* v_i, (2) *demand* (or *workload*) D_i, (3) *deadline* d_i, and (4) *parallelism bound* k_i. The time horizon is divided into d consecutive time slots: $\{1, 2, \cdots, d\}$, where d is the maximum deadline of tasks, i.e., $d = \max_{T_i \in \mathcal{T}}\{d_i\}$, and assumed to be finitely bounded; each slot contains a fixed number of minutes. A task T_i can only utilize the machines located in time interval $[1, d_i]$. The parallelism bound k_i limits that T_i can be simultaneously executed on at most k_i machines. Let k be the maximum parallelism bound of tasks, i.e., $k = \max_{T_i \in \mathcal{T}}\{k_i\}$; here, k_i is a configuration parameter of tasks and k is assumed to be finite [16]. An *allocation* of machines to a task T_i is a function

$$y_i : [1, d_i] \rightarrow \{0, 1, 2, \cdots, k_i\},$$

where $y_i(t)$ is the number of machines allocated to T_i at a slot $t \in [1, d_i]$. In this model, $D_i, d_i \in \mathcal{Z}^+$ for all $T_i \in \mathcal{T}$. Denote by $W(t) = \sum_{T_i \in \mathcal{T}} y_i(t)$ the system's workload at a slot t (i.e., the number of allocated machines at t), and by $\overline{W}(t) = C - W(t)$ its complementary (i.e., the number of idle machines at t). We say that slot t is *fully utilized* if $\overline{W}(t) = 0$, and is *not fully utilized* if $\overline{W}(t) > 0$.

Given the model above, the following three scheduling objectives are considered separately:

— *The first objective* is social welfare maximization and it aims to choose an a subset $\mathcal{S} \subseteq \mathcal{T}$ and produce a feasible schedule of $\mathcal{S}$ on C machines so as to maximize the social welfare $\sum_{T_i \in \mathcal{S}} v_i$ (i.e., the sum of values of tasks completed by deadlines); here, the value v_i of a task T_i is gained if and only if it is *fully allocated* by the deadline, i.e., $\sum_{t \le d_i} y_i(t) \ge D_i$, and partial execution of a task yields no value.

— *The second objective* is machine minimization; it ignores the values of tasks and aims to find the minimum number of machines needed to produce *a feasible schedule* of all tasks $\mathcal{T}$ on C machines where the parallelism and deadline constraints are satisfied.

— *The third objective* is minimizing the maximum weighted lateness of all tasks, i.e., $\max_{T_i \in \mathcal{T}}\{v_i \cdot (t_i - d_i)\}$, where t_i is the completion time of T_i.

In this thesis, we aim to propose good scheduling algorithms. Depending on the objective of an algorithm, its performance is indicated by the makespan, the social welfare, or the maximum weighted lateness of all tasks that it achieves. Given any input of tasks, if an algorithm always produces a schedule of tasks on m machines with the optimal performance, it is optimal. If the optimal algorithm cannot be obtained, the algorithm's quality could be measured by a performance ratio, i.e., the ratio of the proposed algorithm's performance to the performance of an ideally optimal algorithm (that is unknown to us); this ratio is usually referred to as the approximation ratio [21]. Formally, we denote by $A(\mathcal{T})$ and $OPT(\mathcal{T})$ respectively the performance of an algorithm and the optimal one, and an algorithm is called a ρ-approximation algorithm if there exists a value ρ such that, for an arbitrary set $\mathcal{T}$, (i) when a minimization problem is considered,

$$\frac{A(\mathcal{T})}{OPT(\mathcal{T})} \le \rho \ (\rho \ge 1),$$

and (ii) when a maximization problem is considered,

$$\frac{A(\mathcal{T})}{OPT(\mathcal{T})} \ge \rho \ (0 \le \rho \le 1).$$

In this thesis, our final goal is to propose scheduling algorithms that achieve approximation ratios as close to 1 as possible.

Furthermore, we denote by $[l]$ and $[l]^+$ the sets of integers $\{0, 1, \cdots, l\}$ and $\{1, 2, \cdots, l\}$ respectively. Let $len_i = \left\lceil \frac{D_i}{k_i} \right\rceil$ and it could be viewed as the (minimum) execution time of T_i when it always utilizes k_i machines throughout the execution. The deadlines of all tasks of $\mathcal{T}$ constitute a set $\{\tau_1, \tau_2, \cdots, \tau_L\}$, where $L \le n$, $\tau_1, \tau_2, \cdots, \tau_L \in \mathcal{Z}^+$, and $\tau_1 < \tau_2 < \cdots < \tau_L = d$; in other words, given any task $T_i \in \mathcal{T}$, we have $d_i \in \{\tau_1, \tau_2, \cdots, \tau_L\}$. Let $\mathcal{D}_i = \{T_{i,1}, T_{i,2}, \cdots, T_{i,n_i}\}$ denote the set of tasks with deadline τ_i ($i \in [L]^+$), where $|\mathcal{D}_i| = n_i$ and $\sum_{i=1}^{L} n_i = n$. We assume that the demand of each task is an integer. Let $D = \max_{T_i \in \mathcal{T}}\{D_i\}$ be the demand of the largest task. The notation of this section is used in Chapters 3 and 4 and summarized in Table 2.1. Throughout Chapters 3 and 4, we use i, j, m, l, or m' as subscripts to index the element of different sets such as tasks and use t or $\bar{t}$ to index a time slot.

Table 2.1 – Symbol Interpretation

Notation	Explanation
C	the total number of machines
$\mathcal{T}$	a set of tasks to be scheduled on C machines
T_i	a task in $\mathcal{T}$
D_i, d_i, v_i	the workload, deadline, and value of a task T_i
k_i	the parallelism bound of T_i, i.e., the maximum number of machines that can be allocated to and utilized by T_i simultaneously
$y_i(t)$	the number of machines allocated to T_i at a slot t where $y_i(t) \in \{0, 1, \cdots, k_i\}$ and set all $y_i(t)$ to 0 initially
$W(t)$	the total number of machines that are allocated out to the tasks at t, i.e., $W(t) = \sum_{T_i \in \mathcal{T}} y_i(t)$
$\overline{W}(t)$	the total number of machines idle at t, i,e., $\overline{W}(t) = C - W(t)$
len_i	the (minimum) execution time of a task T_i when it always utilizes k_i machines throughout the execution, i.e., $len_i = \left\lceil \frac{D_i}{k_i} \right\rceil$
d, D	the maximum deadline and workload of all tasks of $\mathcal{T}$, i.e., $d = \max_{T_i \in \mathcal{T}} d_i$ and $D = \max_{T_i \in \mathcal{T}} D_i$
$\{\tau_1, \cdots, \tau_L\}$	the set of the deadlines d_i of all tasks T_i of $\mathcal{T}$, where $0 = \tau_0 < \tau_1 < \cdots < \tau_L = d$
$\mathcal{D}_i$	all the tasks $\{T_{i,1}, T_{i,2}, \cdots, T_{i,n_i}\}$ of $\mathcal{T}$ that have a deadline τ_i, $1 \leq i \leq L$
$[l]$	a set of integers $\{0, 1, \cdots, l\}$
$[l]^+$	a set of integers $\{1, 2, \cdots, l\}$

Chapter 3

Optimal Schedule

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan Turing

In this chapter, we identify a state under which C machines can be said to be optimally utilized by a set of tasks. We then propose a scheduling algorithm that achieves such an optimal state.

3.1 Optimal Resource Utilization State

All tasks are denoted by a set $\mathcal{T}$; given any $T_i \in \mathcal{T}$, its deadline $d_i \in \{\tau_1, \tau_2, \cdots, \tau_L\}$ where $\tau_1 < \tau_2 < \cdots < \tau_L$. We denote by $\mathcal{S} \subseteq \mathcal{T}$ an arbitrary subset of $\mathcal{T}$; all tasks of $\mathcal{S}$ with a deadline τ_l are denoted by $\mathcal{S}_l$ where $l \in [L]^+$. In this section, we define the maximum amount of workload of $\mathcal{S}$ that could be processed in a fixed time interval $[\tau_m + 1, \tau_L]$ on C machines for all $m \in [L-1]$, where $\tau_L = d$, i.e., the maximum deadline of tasks.

Firstly, we define the maximum amount of resource, denoted by $\boldsymbol{\lambda_m(\mathcal{S})}$, that could be utilized by $\mathcal{S}$ in $[\tau_{L-m} + 1, \tau_L]$ in an idealized case where there is an indefinite number of machines, i.e., $C = \infty$, for all $m \in [L]^+$. Trivially, we set $\tau_0 = 0$.

To define this, we clarify the maximum amount of resource that an individual task T_i can utilize in $[\tau_{L-m} + 1, \tau_L]$. The deadline and parallelism constraints imply that:

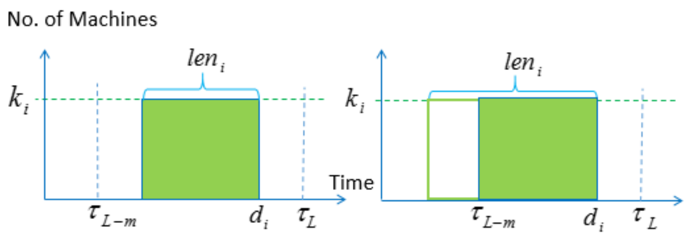

Figure 3.1 – The green areas denote the maximum demand of T_i that need or could be processed in $[\tau_{L-m}+1, \tau_L]$.

— T_i can only utilize the machines in $[1, d_i]$,

— T_i can only utilize at most k_i machines simultaneously at every slot.

The tasks with $d_i \leq \tau_{L-m}$ cannot be executed in the interval $[\tau_{L-m}+1, \tau_L]$. Let us consider a task T_i with $d_i \in [\tau_{L-m}+1, \tau_L]$. The number of slots available in $[\tau_{L-m}+1, d_i]$ is $d_i - \tau_{L-m}$ in the discrete case, and, also recall that len_i the (minimum) execution time of T_i when it always utilizes the maximum number k_i of machines throughout the execution. In the interval $[\tau_{L-m}+1, \tau_L]$, the maximum resource that is available for processing T_i is $(d_i - \tau_{L-m}) \cdot k_i$ and, as illustrated in Fig. 3.1, we have that

— in the case that $len_i \leq d_i - \tau_{L-m}$, the maximum demand of T_i that needs to be processed in $[\tau_{L-m}+1, \tau_L]$ is D_i, i.e., the green area in the left subfigure; here, we have $(d_i - \tau_{L-m}) \cdot k_i \geq len_i \cdot k_i \geq D_i$.

— in the case that $len_i > d_i - \tau_{L-m}$, the maximum demand of T_i that can be processed in $[\tau_{L-m}+1, \tau_L]$ is $(d_i - \tau_{L-m}) \cdot k_i$, i.e., the green area in the right subfigure; here, we have $(d_i - \tau_{L-m}) \cdot k_i < len_i \cdot k_i$, leading to $(d_i - \tau_{L-m}) \cdot k_i < D_i$ since $len_i = \left\lceil \frac{D_i}{k_i} \right\rceil$, and d_i, k_i, τ_{L-m} are integers.

As a consequence of the observation above, we make the following definition.

Definition 1. *Given an arbitrary task $T_i \in \mathcal{S}$, we denote by $\beta_{i,m}$ the maximum workload of T_i that could be processed in $[\tau_{L-m}+1, \tau_L]$ and it is defined as follows:*

— $\beta_{i,m} \leftarrow 0$, *if $d_i \leq \tau_{L-m}$ (i.e., $T_i \in \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_{L-m}$);*

— *if $d_i \geq \tau_{L-m}+1$ (i.e., $T_i \in \mathcal{S}_{L-m+1} \cup \cdots \cup \mathcal{S}_L$), as illustrated in Fig. 3.1,*

  — $\beta_{i,m} \leftarrow D_i$, *if $len_i \leq d_i - \tau_{L-m}$;*

  — $\beta_{i,m} \leftarrow k_i \cdot (d_i - \tau_{L-m})$, *otherwise.*

$\lambda_m(\mathcal{S})$ equals the sum of the maximum workload of every task in $\mathcal{S}$ that could be executed in $[\tau_{L-m}+1, \tau_L]$ and is defined as follows.

Definition 2. *Initially, set $\lambda_m(\mathcal{S})$ to zero for all $m \in [L]^+$. In the case where $C = \infty$ (i.e., the capacity constraint is ignored), $\lambda_m(\mathcal{S})$ is defined as follows:*

$$\lambda_m(\mathcal{S}) \leftarrow \lambda_m(\mathcal{S}) + \beta_{i,m}, \textit{ for every task } T_i \in \mathcal{S}.$$

Secondly, built on Definition 2, we move to the case where C is finite and define the maximum amount of resource that can be utilized by $\mathcal{S}$ on C machines in every $[\tau_{L-m}+1, \tau_L]$, denoted by $\boldsymbol{\lambda_m^C(\mathcal{S})}$, where $m \in [L]^+$.

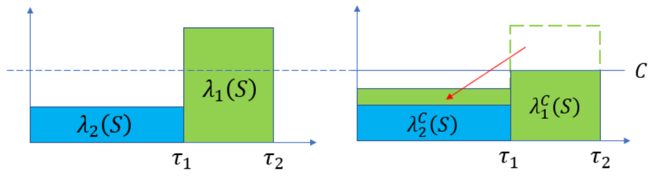

Figure 3.2 – Derivation from the definition $\lambda_m(\mathcal{S})$ to $\lambda_m^C(\mathcal{S})$.

To help readers grasp the intuition in the process of deriving $\lambda_m^C(\mathcal{S})$ from $\lambda_m(\mathcal{S})$, we first illustrate this process in the case where $L=2$ with the support of Fig. 3.2. Fig. 3.2 (left) illustrates the parameter $\lambda_m(\mathcal{S})$ in Definition 2, where the green area denotes $\lambda_1(\mathcal{S})$ and the green and blue areas together denote $\lambda_2(\mathcal{S})$. As illustrated in Fig. 3.2 (right), due to the capacity constraint that C is finite, we have that

(i) $C \cdot (\tau_2 - \tau_1)$ is the maximum possible workload that could be processed in $[\tau_1+1, \tau_2]$ due to the capacity constraint, and $\lambda_1(\mathcal{S})$ is the maximum available workload of $\mathcal{S}$ that needs to be processed in $[\tau_1+1, \tau_2]$ due to the deadline and parallelism constraints. As a result, on C machines, the maximum workload $\lambda_1^C(\mathcal{S})$ of $\mathcal{S}$ that can be processed in $[\tau_1+1, \tau_2]$ is the size of the green area in $[\tau_1+1, \tau_2]$, i.e.,

$$\lambda_1^C(\mathcal{S}) = \min\{C \cdot (\tau_2 - \tau_1), \lambda_1(\mathcal{S})\} = C \cdot (\tau_2 - \tau_1).$$

(ii) After $\lambda_1^C(\mathcal{S})$ workload of $\mathcal{S}$ has been processed in $[\tau_1+1, \tau_2]$, the remaining workload of $\mathcal{S}$ that needs to be processed in $[1, \tau_1]$ is $\lambda_2(\mathcal{S}) - \lambda_1^C(\mathcal{S})$; the maximum workload that could be processed in $[1, \tau_1]$ is $C \cdot \tau_1$ due to the capacity constraint, with $\min\{C \cdot (\tau_1 - \tau_0),\ \lambda_2(\mathcal{S}) - \lambda_1^C(\mathcal{S})\}$ being the maximum remaining workload of $\mathcal{S}$ that could be processed in $[1, \tau_1]$. As a result, $\lambda_2^C(\mathcal{S})$ is defined as follows:

$$\begin{aligned}\lambda_2^C(\mathcal{S}) &= \lambda_1^C(\mathcal{S}) + \min\left\{C \cdot (\tau_1 - \tau_0),\ \ \lambda_2(\mathcal{S}) - \lambda_1^C(\mathcal{S})\right\} \\ &= \min\left\{C \cdot (\tau_2 - \tau_0),\ \ \lambda_2(\mathcal{S})\right\} \\ &= \lambda_2(\mathcal{S}),\end{aligned}$$

i.e., the size of all colored areas in $[\tau_0+1, \tau_2]$.

Generalizing the above process, we derived a recursive definition of $\lambda_m^C(\mathcal{S})$.

Definition 3. *In the case where C is finite (i.e., with the capacity constraint), for all $m \in [L]$, the maximum amount of resource $\lambda_m^C(\mathcal{S})$ that could be utilized by $\mathcal{S}$ in $[\tau_{L-m}+1, \tau_L]$ is defined by the following recursive procedure:*

— *set $\lambda_0^C(\mathcal{S})$ to zero trivially;*

— *set $\lambda_m^C(\mathcal{S})$ to the sum of $\lambda_{m-1}^C(\mathcal{S})$ and* $\min\left\{\lambda_m(\mathcal{S}) - \lambda_{m-1}^C(\mathcal{S}),\ C \cdot (\tau_{L-m+1} - \tau_{L-m})\right\}$.

Now, we give some further explanation of Definition 3 where the amount of resource allocated to $\mathcal{S}$ in some slot interval equals the amount of the workload of $\mathcal{S}$ processed in this interval. Recall that $\lambda_m(\mathcal{S})$ denotes the maximum workload $\mathcal{S}$ that could be processed in $[\tau_{L-m}+1, \tau_L]$ in the case of considering the parallelism and deadline constraints but ignoring the capacity constraint. With all these three constraints considered, when the maximum workload of $\mathcal{S}$ that can be processed in $[\tau_{L-m+1}+1, \tau_L]$ is assumed to be $\lambda_{m-1}^C(\mathcal{S})$, we have that

— if $\lambda_m(\mathcal{S}) - \lambda_{m-1}^C(\mathcal{S}) > C \cdot (\tau_{L-m+1} - \tau_{L-m})$, the C machines are unable to process the remaining $\lambda_m(\mathcal{S}) - \lambda_{m-1}^C(\mathcal{S})$ workload in $[\tau_{L-m}+1, \tau_{L-m+1}]$ and the maximum workload that can be processed here is $C \cdot (\tau_{L-m+1} - \tau_{L-m})$, leading to $\lambda_m^C(\mathcal{S}) = \lambda_{m-1}^C(\mathcal{S}) + C \cdot (\tau_{L-m+1} - \tau_{L-m})$.

— otherwise, the C machines are able to process the remaining $\lambda_m(\mathcal{S}) - \lambda_{m-1}^C(\mathcal{S})$ workload in $[\tau_{L-m}+1, \tau_{L-m+1}]$, leading to $\lambda_m^C(\mathcal{S}) = \lambda_m(\mathcal{S})$.

We finally state our definition that formalizes the concept of optimal utilization of C machines by a set $\mathcal{S}$ of malleable tasks with deadlines:

Definition 4 (Optimal Resource Utilization State)**.** *We say that C machines are optimally utilized by a set of tasks $\mathcal{S}$, if, for all $m \in [L]^+$, $\mathcal{S}$ utilizes $\lambda_m^C(\mathcal{S})$ resources in $[\tau_{L-m}+1, d]$ on C machines.*

We define $\mu_m^C(\mathcal{S}) = \sum_{T_i \in \mathcal{S}} D_i - \lambda_{L-m}^C(\mathcal{S})$ as the remaining (minimum) workload of $\mathcal{S}$ that needs to be processed after $\mathcal{S}$ has maximally utilized C machines in $[\tau_m + 1, \tau_L]$ for all $m \in [L-1]$.

Lemma 1 (Boundary Condition)**.** *If there exists a feasible schedule of $\mathcal{S}$ on C machines, the following inequality holds for all $m \in [L-1]$:*

$$\mu_m^C(\mathcal{S}) \leq C \cdot \tau_m,$$

which is referred to as **boundary condition**.

Proof. Recall the definition of $\lambda_{L-m}^C(\mathcal{S})$ in Definition 3. After $\mathcal{S}$ has maximally utilized the machines in $[\tau_m+1, d]$ and been allocated the maximum amount of resource, i.e., $\lambda_{L-m}^C(\mathcal{S})$, if there exists a feasible schedule for $\mathcal{S}$, the total amount of the remaining demands of $\mathcal{S}$ to be processed should be no more than the capacity $C \cdot \tau_m$ in $[1, \tau_m]$. □

Lemma 2. *Let $\hat{\mathcal{S}}$ denote an arbitrary subset of $\mathcal{S}$, i.e., $\hat{\mathcal{S}} \subseteq \mathcal{S}$. If $\mathcal{S}$ satisfies the boundary condition, then, $\hat{\mathcal{S}}$ also satisfies the boundary condition.*

Proof. Let us consider an arbitrary allocation to $\mathcal{S}$ such that for all $m \in [L]^+$ the total allocation of $\mathcal{S}$ in $[\tau_{L-m}+1, \tau_L]$ is $\lambda_m^C(\mathcal{S})$; we let $\hat{\lambda}_m^C\left(\hat{\mathcal{S}}\right)$ denote the total allocation of $\hat{\mathcal{S}}$ in $[\tau_{L-m}+1, \tau_L]$ and define $\hat{\mu}_{L-m}^C\left(\hat{\mathcal{S}}\right) = \sum_{T_i \in \mathcal{S}} D_i - \hat{\lambda}_m^C\left(\hat{\mathcal{S}}\right)$, where $\hat{\mathcal{S}} \subseteq \mathcal{S}$.

Since $\mathcal{S}$ satisfies the boundary condition, we have $\mu_0^C(\mathcal{S}) = \sum_{T_i \in \mathcal{S}} D_i - \lambda_L^C(\mathcal{S}) \leq 0$, leading to $\lambda_L^C(\mathcal{S}) = \sum_{T_i \in \mathcal{S}} D_i$; so, in the allocation to $\mathcal{S}$, all workload of $\mathcal{S}$ will be processed in $[1, \tau_L]$ and we further have that (i) $\mu_{L-m}^C(\mathcal{S}) = \sum_{T_i \in \mathcal{S}} D_i - \lambda_m^C(\mathcal{S}) = \lambda_L^C(\mathcal{S}) - \lambda_m^C(\mathcal{S})$ and the value of $\mu_{L-m}^C(\mathcal{S})$ denotes the total allocation of $\mathcal{S}$ in $[1, \tau_{L-m}]$ and (ii) similarly, the value of $\hat{\mu}_{L-m}^C(\hat{\mathcal{S}})$ denotes the total allocation of $\hat{\mathcal{S}}$ in $[1, \tau_{L-m}]$, which leads to that $\hat{\mu}_{L-m}^C\left(\hat{\mathcal{S}}\right) \leq \mu_{L-m}^C(\mathcal{S}) \leq C \cdot \tau_{L-m}$ since $\hat{\mathcal{S}} \subseteq \mathcal{S}$. Since $\lambda_m^C\left(\hat{\mathcal{S}}\right)$ denotes the maximum workload of $\mathcal{S}$ that could be processed in $[\tau_{L-m}+1, \tau_L]$, we have that $\mu_{L-m}^C\left(\hat{\mathcal{S}}\right) = \sum_{T_i \in \hat{\mathcal{S}}} D_i - \lambda_m^C\left(\hat{\mathcal{S}}\right) < \hat{\mu}_{L-m}^C\left(\hat{\mathcal{S}}\right) \leq C \cdot \tau_{L-m}$; the lemma thus holds. □

As described in Definition 1, given $m \in [L]^+$ and a task $T_i \in \mathcal{T}$, the maximum workload of T_i that could be processed in $[\tau_{L-m}+1, \tau_L]$ is $\beta_{i,m}$. In the rest of this thesis, when we say that $[\tau_{L-m}+1, \tau_L]$ *is optimally utilized by* T_i, it means that $\beta_{i,m}$ workload of T_i is processed in $[\tau_{L-m}+1, \tau_L]$. When we say that $[\tau_{L-m}+1, \tau_L]$ *is optimally utilized by* $\mathcal{S}$, it means that $\lambda_m^C(\mathcal{S})$ workload of $\mathcal{S}$ is processed in $[\tau_{L-m}+1, d]$ on C machines.

Lemma 3. *Let $\hat{\mathcal{S}} \subseteq \mathcal{S}$, and $T_i \in \mathcal{S} - \hat{\mathcal{S}}$. If $\lambda_m^C\left(\hat{\mathcal{S}}\right)$ workload of $\hat{\mathcal{S}}$ and $\beta_{i,m}$ workload of T_i are simultaneously processed in $[\tau_{L-m}+1, \tau_L]$ on C machines, we have that $[\tau_{L-m}+1, \tau_L]$ is optimally utilized by $\hat{\mathcal{S}} \cup \{T_i\}$.*

Proof. It suffices to show that $\lambda_m^C\left(\hat{\mathcal{S}} \cup \{T_i\}\right) \leq \lambda_m^C\left(\hat{\mathcal{S}}\right) + \beta_{i,m}$. Let us consider an allocation to $\hat{\mathcal{S}} \cup \{T_i\}$ such that its $\lambda_m^C\left(\hat{\mathcal{S}} \cup \{T_i\}\right)$ workload is processed on C machines in $[\tau_{L-m}+1, d]$; we denote by α_1 and α_2 the workload of T_i and the workload of $\hat{\mathcal{S}}$ that are simultaneously processed in $[\tau_{L-m}+1, d]$; here, we have $\beta_{i,m} \geq \alpha_1$. According to the meaning of $\lambda_m^C\left(\hat{\mathcal{S}} \cup \{T_i\}\right)$, we have that, after α_1 workload of T_i has already been allocated on C machines in $[\tau_{L-m}+1, d]$ in advance, the maximum workload of $\hat{\mathcal{S}}$ that could be processed in $[\tau_{L-m}+1, d]$ is α_2; in contrast, $\lambda_m^C\left(\hat{\mathcal{S}}\right)$ is the maximum workload of $\hat{\mathcal{S}}$ that could be processed on C machines in $[\tau_{L-m}+1, d]$ under the case that the C machines are idle and assigned no workload in $[\tau_{L-m}+1, d]$ in advance. Hence, we have $\lambda_m^C\left(\hat{\mathcal{S}}\right) \geq \alpha_2$. Finally, we have $\lambda_m^C\left(\hat{\mathcal{S}} \cup \{T_i\}\right) = \alpha_1 + \alpha_2 \leq \lambda_m^C\left(\hat{\mathcal{S}}\right) + \beta_{i,m}$. □

Besides Table 2.1, the additional notation to be used in this chapter is summarized in Table 3.1.

3.2 Scheduling Algorithm

In this section, we show that, if $\mathcal{S}$ satisfies the boundary condition above, then, there exists an algorithm LDF($\mathcal{S}$) that produces a feasible schedule of $\mathcal{S}$, achieving the optimal resource utilization state.

Table 3.1 – Main Notation for Chapter 3

Notation	Explanation
$\mathcal{S}$	a set of tasks to be allocated by LDF($\mathcal{S}$) and $\mathcal{S} \subseteq \mathcal{T}$
$\mathcal{S}_i$	the tasks of $\mathcal{S}$ with a deadline τ_i
$\lambda_m(\mathcal{S})$	the maximum amount of resource that could be utilized by $\mathcal{S}$ in $[\tau_{L-m}+1, \tau_L]$ in an idealized case where there is an indefinite number of machines, $m \in [L]^+$
$\lambda_m^C(\mathcal{S})$	the maximum amount of resource that can be utilized by $\mathcal{S}$ on C machines in every $[\tau_{L-m}+1, \tau_L]$, $m \in [L]^+$
$\mu_m^C(\mathcal{S})$	the remaining workload of $\mathcal{S}$ that needs to be processed after $\mathcal{S}$ has optimally utilized C machines in $[\tau_m+1, \tau_L]$, i.e., $\mu_m^C(\mathcal{S}) = \sum_{T_i \in \mathcal{S}} D_j - \lambda_{L-m}^C(\mathcal{S})$, $m \in [L-1]$
T_i	a task that is being allocated by the algorithm LDF($\mathcal{S}$); the actual allocation is done by Allocate-B(i)
$\mathcal{S}'$	so far, all tasks that have been fully allocated by LDF($\mathcal{S}$) and are considered before T_i
$\mathcal{S}''$	$\mathcal{S}'' = \mathcal{S}' \cup \{T_i\}$
t_0	a turning point defined in Property C.2, with time slots respectively later than and no later than t_0 having different resource utilization state
t_1	similar to t_0, a turning point defined in Lemma 5 upon completion of Fully-Utilize(i)
t_2	similar to t_0, a turning point defined in Lemma 8 upon completion of Fully-Allocate(i)
t'	the latest time slot in $[1, \tau_m]$ with $\overline{W}(t') > 0$
t'', t'''	a time slot that satisfies some property defined and only used in Chapter 3.2.3

3.2.1 Overview of LDF($\mathcal{S}$)

Initially, for all $T_i \in \mathcal{S}$ and $t \in [1, d]$, we set the allocation $y_i(t)$ to zero; then, LDF($\mathcal{S}$) runs as follows:

1. the tasks of $\mathcal{S}$ are considered in the decreasing order of their deadlines, i.e., in the order of $\mathcal{S}_L$, $\mathcal{S}_{L-1}$, $\cdots$, $\mathcal{S}_1$, where the tasks in the same set are considered in random order;
2. for a task T_i being considered, the algorithm Allocate-B(i), presented as Algorithm 1, is called to allocate D_i resource to T_i under the deadline and parallelism constraints.

We will show that, only if $\mathcal{S}$ satisfies the boundary condition and the resource utilization satisfies some properties upon every completion of Allocate-B($\cdot$), all tasks in $\mathcal{S}$ will be fully allocated after LDF($\mathcal{S}$) ends. We first elaborate the high-level idea in the process of deriving this conclusion.

In LDF($\mathcal{S}$), when a task T_i is being considered, suppose that T_i belongs to

$\mathcal{S}_m$ and denote by $\mathcal{S}' \subseteq \mathcal{S}_L \cup \cdots \cup \mathcal{S}_m$ the tasks that have been fully allocated so far; the tasks of $\mathcal{S}'$ are considered before T_i. Here, $\mathcal{S}$ satisfies the boundary condition and so do all its subsets including $\mathcal{S}'$ and $\mathcal{S}' \cup \{T_i\}$ by Lemma 2. Before the execution of Allocate-B(i), we assume that the resource allocation state satisfies the following two properties.

Recall the optimal resource utilization state in Definition 4, and the first property is that such a state of C machines is achieved by the current allocation to $\mathcal{S}'$.

***Property* 3.2.1.** *For all $l \in [L]^+$, $\lambda_l^C(\mathcal{S}')$ workload of $\mathcal{S}'$ is processed in $[\tau_{L-l}+1, d]$ where $\lambda_l^C(\mathcal{S}')$ is defined in Definition 3.*

The second property is that a stepped-shape resource utilization state is achieved in $[1, \tau_m]$ by the current allocation to $\mathcal{S}'$.

***Property* 3.2.2.** *If there exists a slot $t \in [1, \tau_m]$ such that $\overline{W}(t) > 0$, let t_0 be the latest slot in $[1, \tau_m]$ such that $\overline{W}(t_0) > 0$; then we have $\overline{W}(1) \geq \overline{W}(2) \geq \cdots \geq \overline{W}(t_0)$.*

If Property C.1 and Property C.2 hold, we will show in Chapter 3.2.2 and 3.2.3 that, there exists an algorithm Allocate-B(i) such that upon completion of Allocate-B(i) the following two properties are satisfied:

***Property* 3.2.3.** *T_i is fully allocated.*

***Property* 3.2.4.** *The resource allocation to $\mathcal{S}' \cup \{T_i\}$ still satisfies Property C.1 and Property C.2.*

In the case that the above Allocate-B(i) exists, only if $\mathcal{S}$ satisfies the boundary condition, $\mathcal{S}$ can be fully allocated by LDF($\mathcal{S}$). The reason for this can be explained by induction. When the first task T_i in $\mathcal{S}$ is considered, $\mathcal{S}'$ is empty, and, before the execution of Allocate-B(i), Properties C.1 and C.2 holds trivially. Further, upon completion of Allocate-B(i), T_i will be fully allocated by Allocate-B(i) due to Property C.3, and Property C.4 still holds.. Then, assume that $\mathcal{S}'$ that denotes the current fully allocated tasks is nonempty and Properties C.1 and C.2 hold; the task T_i being considered by LDF($\mathcal{S}$) will still be fully allocated and Properties C.3 and C.4, upon completion of Allocate-B(i). Hence, all tasks in $\mathcal{S}$ will be finally fully allocated upon completion of LDF($\mathcal{S}$).

In the rest of this section, we will propose an algorithm Allocate-B(i) mentioned above: in the case that $\mathcal{S}$ satisfies the boundary condition, upon completion of Allocate-B(i) Properties C.3 and C.4 holds if the resource allocation state of $\mathcal{S}'$ before executing Allocate-B(i) satisfies Properties C.1 and C.2. Then, we immediately have the following proposition:

Proposition 1. *If $\mathcal{S}$ satisfies the boundary condition, LDF($\mathcal{S}$) will produce a feasible schedule of $\mathcal{S}$ on C machines.*

Overview of Allocate-B(i). Before executing Allocate-B(i), the allocation of the previously allocated tasks $\mathcal{S}'$ satisfies Properties C.1 and C.2, also illustrated

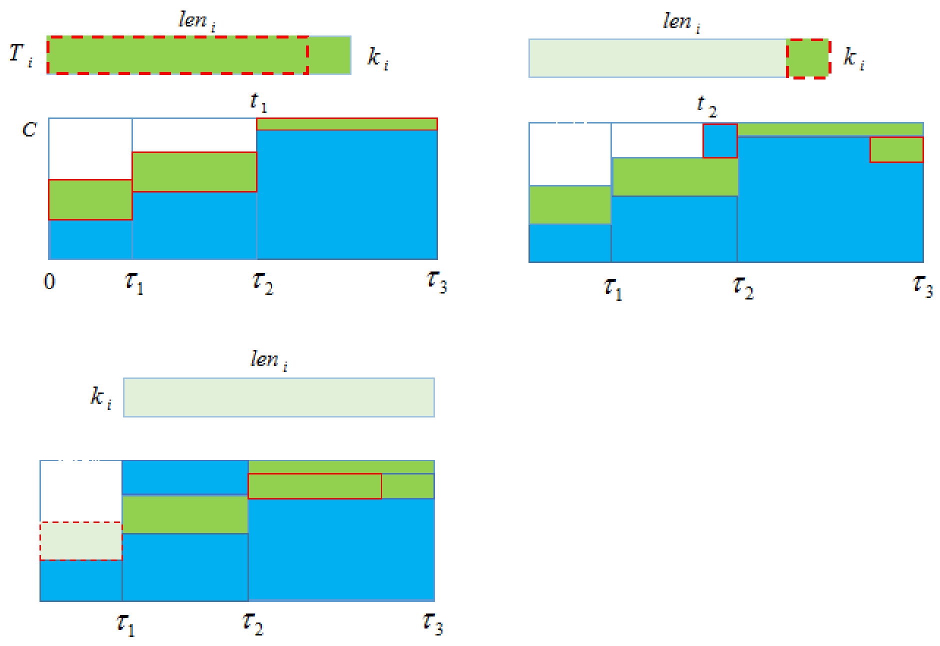

Figure 3.3 – The resource allocation state of T_i and $\mathcal{S}'$ respectively upon completion of Fully-Utilize(i), Fully-Allocate(i), and AllocateRLM(i, 1, t_2+1) where $L = m = 3$: on the C machines, the blue area denotes the allocation to $\mathcal{S}'$ that satisfies Properties C.1 and C.2 before executing Allocate-B(i) while the green area denotes the allocation to T_i.

by the blue area of the 1st subfigure of Fig 3.3; $\mathcal{S}'$, $\mathcal{S}' \cup \{T_i\}$ and $\mathcal{S}$ satisfy the boundary condition where $\mathcal{S}' \cup \{T_i\} \subseteq \mathcal{S}$ by Lemma 2. The construction of Allocate-B(i) will proceed with two phases where $d_i = \tau_m$. In the first phase, we introduce what operations are feasible to make T_i fully allocated D_i resource, which will be presented in two algorithms Fully-Utilize(i) and Fully-Allocate(i). Fully-Utilize(i) operates as follows:

(i) For every slot t from the deadline d_i towards earlier time slots, Fully-Utilize(i) makes T_i fully utilize the maximum amount of machines available at t with the parallelism constraint. Upon its completion, we have that

— the resource allocation state of T_i is such that $y_i(1) \leq y_2(t) \leq \cdots \leq y_i(d_i)$, as illustrated by the green area of the 1st subfigure of Fig. 3.3,

— the allocation state of $\mathcal{S}' \cup \{T_i\}$ satisfies Property C.2, i.e., $W(1) \leq W(2) \leq \cdots \leq W(d_i)$, as illustrated by the colored area of the 1st subfigure of Fig. 3.3.

Afterwards, in the case that T_i has not been fully allocated yet (i.e., $D_i - \sum_{\bar{t}=1}^{d_i} y_i(\bar{t}) > 0$), we use the green area with the dotted red frame in the 2nd subfigure of Fig. 3.3 to denote the remaining workload of T_i and Fully-Allocate(i)

is proposed to make T_i fully allocated. In this case, there exists a slot not fully utilized and let t_1 denote the latest such slot in $[1, d_i]$; otherwise, T_i would have been fully allocated since $\mathcal{S}' \cup \{T_i\}$ satisfies the boundary condition. Further, we also have that

— the resource allocation state of T_i is such that $y_i(1) = \cdots = y_i(t_1) = k_i$, as illustrated by the green area of the 1st subfigure of Fig. 3.3.

Based on such resource allocation states, Fully-Allocate(i) transfers the allocations of $\mathcal{S}'$ at the slots in $[t_1 + 1, d_i]$ to the latest slots in $[1, t_1]$ that have idle machines so that more machines could be allocated to process the remaining workload of T_i without violating the parallelism constraint, and it operates as follows:

(ii) for every slot t from d_i to $t_1 + 1$, Fully-Allocate(i) transfers the allocation of $\mathcal{S}'$ at t to the latest slot $t' \in [1, t_1]$ that have not become fully utilized with idle machines, and the transfer stops until $\overline{W}(t)$ becomes $k_i - y_i(t)$ or $D_i - \sum_{\bar{t}=1}^{d_i} y_i(\bar{t})$; then, T_i is allocated $\overline{W}(t)$ extra machines at t, after which we have that either $y_i(t) = k_i$ or T_i has been fully allocated.

Upon completion of Fully-Allocate(i), T_i is fully allocated since $D_i \leq k_i \cdot d_i$; then, the resource allocation states of T_i and $\mathcal{S}'$ are illustrated by the green and blue areas of the 2nd subfigure of Fig. 3.3.

Now, the allocation state will not satisfy Property C.1 if there exists some interval $[\tau_l + 1, \tau_L]$ that is not optimally utilized by $\mathcal{S}' \cup \{T_i\}$ where $l \in [m-1]$, as illustrated by the blank area in $[\tau_1 + 1, \tau_3]$ of the 2nd subfigure of Fig. 3.3; let t_2 denote the latest slot in $[1, t_1]$ such that $\overline{W}(t_2) > 0$. So, in the second phase, an adjustment to the allocations of T_i and $\mathcal{S}'$ is needed so that Property C.1 is satisfied: partial allocation of $\mathcal{S}'$ in $[t_2 + 1, d_i]$ is transferred to the latest slots in $[1, t_2]$, aimming to transfer the allocation of T_i in $[1, t_2]$ to $[t_2 + 1, d_i]$; in particular, we propose an algorithm AllocateRLM(i, η_1, x) that operates as follows:

(iii) for every t from d_i to $t_2 + 1$, if $y_i(t) < k_i$, AllocateRLM($\cdot$) transfers the allocation of $\mathcal{S}'$ at t to the latest slot t' in $[1, t_2]$ that have not become fully utilized, and the transfer stops until $\overline{W}(t)$ becomes $k_i - y_i(t)$ or $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) = \overline{W}(t)$; then, let $\theta = \overline{W}(t)$, allocate θ extra machines to T_i at t, and equivalently reduce the allcoation of T_i at the earliest slots in $[1, t_2]$ by θ, after which we have that either $y_i(t) = k_i$ or the allocation of T_i in $[1, t' - 1]$ is zero.

Upon completion of AllocateRLM($\cdot$), Property C.1 holds, and the resource allocation states of $\mathcal{S}' \cup \{T_i\}$, $\mathcal{S}'$, and T_i are illustrated by the green and blue area, the blue area and the green area in the 3rd subfigure of Fig. 3.3.

During executig Fully-Allocate(i) and AllocateRLM($\cdot$), a core operation is transferring the allocation of $\mathcal{S}'$ from t to an earlier slot t'. The feasibility of such transfer lies in that, we can always find a task $T_{i'} \in \mathcal{S}'$ such that its allocation at t is larger than its allocation at t', i.e., $y_{i'}(t) > y_{i'}(t')$; then, we can reduce its allocation at t while increasing its allocation at t'. The existence of $T_{i'}$ can be guaranteed when (a) $y_i(t) = k_i$ and $\overline{W}(t) = 0$ and (b) $y_i(t') < k_i$ and $\overline{W}(t') > 0$, which will be explained in Lemma 7.

Algorithm 1: Allocate-B(i)

```
1 Fully-Utilize($i$);
2 Fully-Allocate($i$);
3 AllocateRLM($i$, 1, $t_2 + 1$);
```

Following the above sketch of Allocate-B(i), the details of the first and second phases will be presented in Chapter 3.2.2 and Chapter 3.2.3 respectively where Allocate-B(i) is presented as Algorithm 1.

3.2.2 Phase 1

Now, we introduce Fully-Utilize(i) and Fully-Allocate(i) formally. Before their execution, recall that we assume in the last subsection $T_i \in \mathcal{S}_m$; the allocation to the previously allocated tasks $\mathcal{S}'$ satisfies Properties C.1 and C.2, as illustrated by the blue area in the first subfigure of Fig. 3.3. The whole set of tasks $\mathcal{S}$ to be scheduled satisfies the boundary condition where $\mathcal{S}' \subsetneq \mathcal{S}$; so, $\mathcal{S}'$ and $\mathcal{S}' \cup \{T_i\}$ also satisfy the boundary condition by Lemma 2.

Fully-Utilize(i). Initially, set $y_i(t)$ to zero for all time slots, and, Fully-Utilize(i) operates as follows:

— for every time slot t from the deadline d_i to 1, set

$$y_i(t) \leftarrow \min\{k_i, \overline{W}(t), D_i - \sum\nolimits_{\bar{t}=t+1}^{d_i} y_i(\bar{t})\}. \tag{3.1}$$

Here, k_i is the parallelism bound, $\overline{W}(t)$ is the number of machine idle at t, and $D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t})$ is the remaining workload to be processed upon completion of its allocations at slots $t+1, \cdots, d_i$; specially, $\sum_{\bar{t}=d_i+1}^{d_i} y_i(\bar{t})$ is set to 0, representing the allocation to T_i is zero before Fully-Utilize(i) begins. Their minimum denotes the maximum amount of machines that T_i can or needs to utilize at t after the allocation to T_i at slots $t+1, \cdots, d_i$. Upon completion of Fully-Utilize(i), the allocation of T_i on C machines is illustrated by the green area of the first subfiture of Fig. 3.3; in this example, only partial demand of T_i is allocated, illustrated by the green area with the red dotted frame.

Before executing Fully-Utilize(i), the resource allocation to the previously allocated tasks $\mathcal{S}'$ satisfies Property C.1. Its execution does not change the previous allocation to $\mathcal{S}'$. Let

$$\mathcal{S}'' = \mathcal{S}' \cup \{T_i\}.$$

Since $d_i = \tau_m$, the workload of T_i can only be processed in $[1, \tau_m]$; the maximum workload of $\mathcal{S}''$ that could be processed in $[\tau_m + 1, \tau_L]$ still equals its counterpart when $\mathcal{S}'$ is considered. We come to the following conclusion in order to not violate the boundary condition:

Lemma 4. *Upon completion of Fully-Utilize(i), all tasks of $\mathcal{S}$ would have been fully allocated* in the case that *the total allocation to $\mathcal{S}''$ in $[1, \tau_m]$ is $C \cdot \tau_m$, i.e., $C \cdot \tau_m = \sum_{T_j \in \mathcal{S}''} \sum_{\bar{t}=1}^{\tau_m} y_j(\bar{t})$.*

Proof. Before executing Fully-Utilize(i), the resource allocation to $\mathcal{S}'$ satisfies Property C.1. Its execution does not change the previous allocation to $\mathcal{S}'$. Let $\mathcal{S}'' = \mathcal{S}' \cup \{T_i\}$. Since $d_i = \tau_m$, the workload of T_i can only be processed in $[1, \tau_m]$; the maximum workload of $\mathcal{S}''$ that could be processed in $[\tau_m + 1, \tau_L]$ still equals its counterpart when $\mathcal{S}'$ is considered, i.e., $\lambda_{L-m}^{C}(\mathcal{S}'') = \lambda_{L-m}^{C}(\mathcal{S}')$. Upon completion of Fully-Utilize(i), if the total allocation to $\mathcal{S}''$ in $[1, \tau_m]$ is $C \cdot \tau_m$, we could conclude that T_i is the last task of $\mathcal{S}$ being considered and all tasks in $\mathcal{S}$ have been fully allocated; otherwise, $\mathcal{S}'' \subsetneq \mathcal{S}$, which contradicts the fact that $\mathcal{S}$ and its subset satisfy the boundary condition, which implies that after the maximum workload of $\mathcal{S}''$ has been processed in $[\tau_m + 1, \tau_L]$, the remaining workload $\mu_m^C(\mathcal{S}'') = \mu_m^C(\mathcal{S}') + D_i \leq C \cdot \tau_m$. $\square$

Upon completion of Fully-Utilize(i), *the other case* is the total allocation to $\mathcal{S}''$ is $< C \cdot \tau_m$; then, there exists a slot $t \in [1, \tau_m]$ such that $\overline{W}(t) > 0$, and let t_1 denote the latest such time slot in $[1, \tau_m]$ where $t_1 \leq t_0$.

During executing Fully-Utilize(i), at the moment of deciding the allocation of T_i at t_1, we have that

$$y_i(t_1) = \min\{k_i, D_i - \sum\nolimits_{\bar{t}=t_1+1}^{d_i} y_i(\bar{t})\} < \overline{W}(t_1) \tag{3.2}$$

since after the allocation at t_1 we still have $\overline{W}(t_1) > 0$; for all $t \in [1, t_1 - 1]$, at the moments of deciding the allocations at t and $t+1$ respectively, we have that

$$\min\{k_i, D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t})\} \leq \min\{k_i, D_i - \sum_{\bar{t}=(t+1)+1}^{d_i} y_i(\bar{t})\}. \tag{3.3}$$

On the other hand, before executing Fully-Utilize(i), we have for all $t \in [1, t_1 - 1]$ that

$$\overline{W}(t) \geq \overline{W}(t+1) \geq \overline{W}(t_1). \tag{3.4}$$

By (3.4), (3.2), and (3.3), we have for all $t \in [1, t_1 - 1]$ that

$$\overline{W}(t) > \min\{k_i, D_i - \sum\nolimits_{\bar{t}=t+1}^{d_i} y_i(\bar{t})\} \tag{3.5}$$

Finally, by (3.1), (3.2) and (3.5), we have that, when deciding the allocation of T_i at t,

$$y_i(t) = \min\{k_i, D_i - \sum\nolimits_{\bar{t}=t+1}^{d_i} y_i(\bar{t})\}, \tag{3.6}$$

for all $t \in [1, t_1]$; together with (3.3), we have upon completion of Fully-Utilize(i) that

$$y_i(1) \leq y_i(2) \leq \cdots \leq y_i(t_1). \tag{3.7}$$

As a result, due to (3.6), we derive the 1st point of Lemma 5 below; due to (3.4) and (3.7), we derive the 2nd point.

Lemma 5. *Upon completion of Fully-Utilize(i), let t_1 denote the latest such time slot in $[1, \tau_m]$ such that $\overline{W}(t_1) > 0$; in the case that the total allocation to $\mathcal{S}''$ is $< C \cdot \tau_m$,*

- *for all $t \in [1, t_1]$, if the total allocation of T_i in $[t, d_i]$ is $< D_i$, i.e., $D_i - \sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) > 0$, we have $y_i(t) = k_i$;*
- *as illustrated in the 1st subfigure of Fig. 3.3, the numbers of idle machines in $[1, t_1]$ have a stepped shape, i.e., $\overline{W}(1) \geq \cdots \geq \overline{W}(t_1) > 0$.*

Fully-Allocate(i). Upon completion of Fully-Utilize(i), the resource allocation states of T_i and $\mathcal{S}''$ are described in Lemma 5, respectively illustrated by the green area and the blue and green area in the first subfigure of Fig. 3.3. This enables us to propose Fully-Allocate(i), presented as Algorithm 2, to make T_i fully allocated; please notice that Algorithm 2 is explained by its comments.

Now, we explain the existence of $T_{i'}$ in line 12 of Routine($\cdot$) and the reason why T_i will be finally fully allocated by Fully-Allocate(i). The only operation that changes the allocation to T_i occurs at line 6 of Fully-Allocate(i). Hence, we have

Lemma 6. *Fully-Allocate(i) never decreases the allocation $y_i(t)$ to T_i at any time slot $t \in [1, d_i]$ during its execution, compared with the $y_i(t)$ just before executing Fully-Allocate(i).*

We could prove by contradiction that

Lemma 7. *When Routine(Δ, η_1, η_2, t) is called, the task $T_{i'}$ in line 12 exists if (i) the condition in line 4 or 7 is false, (ii) $y_i(t') = k_i$, and (iii) $y_i(t) < k_i$ and $\overline{W}(t) = 0$.*

Proof. Recall that $W(t)$ is the sum of the allocations $y_j(t)$ of all tasks $T_j \in \mathcal{S}$ at t and $\overline{W}(t) + W(t) = C$. Initially, we have the inequality that $W(t) - y_i(t) > W(t') - y_i(t')$ due to the conditions (i)-(iii) of Lemma 7, and, there exists a $T_{i'}$ such that $y_{i'}(t') < y_{i'}(t)$; otherwise, that inequality would not hold. In each iteration of Routine($\cdot$), $\overline{W}(t)$ becomes > 0 since partial allocation of $T_{i'}$ is transferred from t to t'; however, it still holds that $\overline{W}(t) < \Delta \leq k_i - y_i(t)$. So, we have

$$W(t) - y_i(t) = C - \overline{W}(t) - y_i(t) > W(t') - k_i = W(t') - y_i(t')$$

and such $T_{i'}$ can still be found like the initial case. $\square$

At each iteration of Fully-Allocate(i), if there exists a t' such that $\overline{W}(t') > 0$ in the loop of Routine($\cdot$), with Lemmas 5 and 6, we have $y_i(t') = k_i$. Since $\Omega > 0$ and $y_i(t) < k_i$, when Routine($\cdot$) is called, we have $\overline{W}(t) = 0$; otherwise, this contradicts Lemma 5. With Lemma 7, we will conclude that the task $T_{i'}$ in line 12 exists when it is called by Fully-Allocate(i). In addition, the operation

Algorithm 2: Fully-Allocate(i)

```
/* initially, let $\Omega = D_i - \sum_{\bar{t} \leq d_i} y_i(\bar{t})$; it denotes the remaining demand of
   $T_i$ after deducting its current allocation, illustrated by the green area
   with the red dotted frame in the 2nd subfigure of Fig. 3.3.                */
/* for all $t \in [1, t_1]$, the number of machines allocated to $T_i$ at $t$ is
   $y_i(t) = k_i$ if $\Omega > 0$, by Lemma 5; the slots in $[t_1 + 1, d_i]$ are fully
   utilized; such resource allocation states are also illustrated by the
   green area and the colored (i.e., blue and green) area in the 1st
   subfigure of Fig. 3.3.                                                      */
/* for every $t$ from $d_i$ to $t_1 + 1$, $T_i$ can utilize at most $k_i - y_i(t)$ more
   machines with the parallelism constraint; let $\Delta \leftarrow \min\{k_i - y_i(t), \Omega\}$ and
   if $\Delta > 0$ Fully-Allocate($\cdot$) repeatedly transfers the allocations of the
   previously allocated tasks $\mathcal{S}'$ at $t$ to earlier slots that are closest to
   $t$ and not fully utilized until (i) $\overline{W}(t) = \Delta$ or (ii) all slots in $[1, t_1]$
   are fully utilized (lines 2-5 below); then allocate $\overline{W}(t)$ more machines
   to $T_i$ at $t$ (line 6); Upon completion of the operations above, the
   resource allocation states of $T_i$ and $\mathcal{S}'$ are illustrated by the green and
   blue areas in the 2nd subfigure of Fig. 3.3 respectively.                   */
/* when Fully-Allocate($\cdot$) ends, we have either (i) all slots in $[1, t_1]$ are
   fully utilized (see lines 5, 7, 8 of Fully-Allocate($\cdot$) and lines 1-5 of
   Routine($\cdot$)) or (ii) $T_i$ is fully allocated; here $D_i \leq k_i \cdot d_i$.     */
1 for $t \leftarrow d_i$ to $t_1 + 1$ do
2     $\Omega = D_i - \sum_{\bar{t}=1}^{d_i} y_i(\bar{t})$;
3     if $\Omega > 0$ then
          // $T_i$ has not been fully allocated yet
4         $\Delta \leftarrow \min\{k_i - y_i(t), \Omega\}$// our aim is to make $T_i$ allocated $D_i$
              resource without violating the parallelism constraint.
5         $flag \leftarrow 0$, and call Routine($\Delta$, 1, 0, $t$), presented as Algorithm 3;
6         allocate $\overline{W}(t)$ more machines to $T_i$ at $t$: $y_i(t) \leftarrow y_i(t) + \overline{W}(t)$, and,
              $\Omega \leftarrow \Omega - \overline{W}(t)$// afterwards, $\overline{W}(t)$ becomes zero again.
7         if $flag = 1$ then
8             break;
```

at line 13 of Routine($\cdot$) does not change the total allocation to $T_{i'}$, and violate the parallelism bound $k_{i'}$ of $T_{i'}$ since the current $y_{i'}(t')$ is no more than the initial $y_{i'}(t)$.

Proposition 2. *Upon completion of Fully-Allocate(i), the task T_i is fully allocated.*

Proof. Fully-Allocate(i) ends up with one of the following two events. The first

Algorithm 3: Routine(Δ, η_1, η_2, t)

```
/* Given $t$, Routine($\cdot$) repeatedly finds a slot $t'$ earlier than but closest
   to $t$ such that $t'$ is not fully utilized (line 2 below); it transfers the
   allocation of a previously allocated task from $t$ to $t'$ (lines 12-13),
   under which the total allocation at $t'$ is increased and such $t'$ will
   possibly become fully utilized in future                                  */
/* the loop stops when (i) the number of idle machines $\overline{W}(t)$ becomes $\Delta$
   (line 1), or (ii) one of the conditions at lines 4, 7, 10 is true          */
1  while $\overline{W}(t) < \Delta$ do
2      $t' \leftarrow$ the current time slot earlier than and closest to $t$ so that
       $\overline{W}(t') > 0$
3      if $\eta_1 = 1$ then
4          if there exists no such $t'$ then
               // all slots in $[1, t-1]$ are fully utilized
5              $flag \leftarrow 1$, break
6      else
7          if $t' \leq t^{th}_{m-1}$, or there exists no such $t'$ then
               // all slots in $[t^{th}_{m-1}+1, t-1]$ are fully utilized
8              $flag \leftarrow 1$, break
9      if $\eta_2 = 1$ then
10         if $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) \leq \overline{W}(t)$ then
11             $flag \leftarrow 1$, break
       /* when Routine($\Delta$, 1, 0, $t$) is called in Fully-Allocate($i$), we have
          $\Omega > 0$ and $\overline{W}(t) = 0$, under which the $T_{i'}$ in line 12 exists if the
          condition in line 1 is true and the condition in line 4 is false;
          here, $y_i(t') = k_i$ since $\overline{W}(t') > 0$ and $\Omega > 0$, and $y_i(t) < k_i$ since
          $\Delta > 0$.                                                         */
       /* when Routine($\Delta$, $\eta_1$, 1, $t$) is called in AllocateRLM($i$, $\eta_1$, $x$), the
          $T_{i'}$ in line 12 exists if the condition in line 1 is true, the $t'$ in
          line 2 exists, and $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > \overline{W}(t)$; here, $y_i(t') = k_i$ since
          $\overline{W}(t') > 0$ and $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$, and $y_i(t) < k_i$ since $\Delta > 0$.   */
12     let $i'$ be a task such that $y_{i'}(t) > y_{i'}(t')$
13     $y_{i'}(t) \leftarrow y_{i'}(t) - 1$, $y_{i'}(t') \leftarrow y_{i'}(t') + 1$
```

is that the condition in line 4 of Routine($\cdot$) is true; then, we have for all $t \in [1, \tau_m]$ that $\overline{W}(t) = 0$ and as explained in Lemma 4 we conclude that all tasks in $\mathcal{S}$ has been fully allocated. The second occurs after finishing the iteration of Fully-Allocate(i) at time slot t_1+1; then, there is always a slot t' in $[1, t_1]$ that are not fully utilized. As a result, we have that T_i has been fully allocated and we prove this by contradiction. Otherwise, upon completion of Fully-Allocate(i), $\Omega > 0$, which also holds upon completion of each of its iterations at $t \in [t_1 + 1, d_i]$; we thus have $y_i(t) = k_i$. Moreover, we also have $y_i(t) = k_i$ for all $t \in [t_1]^+$ due to Lemma 5. This contradicts $\Omega > 0$ upon completion of Fully-Allocate$(\cdot)$ since $D_i \leq k_i \cdot d_i$. Finally, the proposition holds. □

Upon completion of Fully-Allocate(i), the resource allocation states of T_i and $\mathcal{S}' \cup \{T_i\}$ are described as follows, which are illustrated by the green area and the colored (green and blue) area in the 2nd subfigure of Fig. 3.3.

Lemma 8. *Upon completion of Fully-Allocate(i), if there exists a $t \in [1, \tau_m]$ such that $\overline{W}(t) > 0$, let t_2 be the latest such slot where $\overline{W}(t_2 + 1) = \cdots = \overline{W}(\tau_m) = 0$ if $t_2 < \tau_m$:*

— *for all $t \in [1, t_2]$, if the total allocation of T_i in $[t, d_i]$ is $< D_i$ (i.e., $D_i - \sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) > 0$), we have $y_i(t) = k_i$;*

— *the numbers of available machines in $[1, t_2]$ have a stepped shape, i.e, $\overline{W}(1) \geq \cdots \geq \overline{W}(t_2) > 0$.*

Here $t_2 \leq t_1$.

Proof. In the case that T_i has been allocated D_i resource just upon completion of Fully-Utilize$(\cdot)$, Fully-Allocate(i) does nothing upon its completion (see line 3) and the conclusion here is the same conclusion in Lemma 5 where $t_2 = t_1$; thus, the lemma holds.

In the following, we analyze the opposite case. Before executing Fully-Allocate(i), the resource allocation state satisfies Lemma 5: (a) the slots in $[t_1 + 1, d_i]$ are fully utilized and $\overline{W}(\bar{t} - 1) \geq \overline{W}(\bar{t}) > 0$ for all $\bar{t} \in [2, t_1]$, and (b) given any $t \in [1, t_1]$, if $\sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) < D_i$, we have $y_i(t) = k_i$. Fully-Allocate(i) considers every slot t from d_i to $t_1 + 1$ (line 1). It transfers the allocations of previous tasks at t to an earlier slot t' that is closest to t and not fully utilized (line 5); then, the total allocation to all tasks at t' is increased and $\overline{W}(t')$ may gradually become zero; so, some of the slots in $[1, t_1]$ may become fully utilized one by one from t_1 towards the earlier ones. Upon completion of Fully-Allocate(i), let t_2 denotes the latest slot in $[1, t_1]$ that is not fully utilized: (i) the allocations of all tasks at every $\bar{t} \in [1, t_2 - 1]$ is still the same as the ones before executing Fully-Allocate(i); (ii) the total allocation of all tasks at t_2 may be increased but the allocation to T_i at t_2 does not change; (iii) the slots in $[t_2 + 1, t_1]$ are fully utilized. Hence, due to (i)-(ii), the resource allocation state in $[1, t_2]$ still satisfies Lemma 5. On the other hand, after transferring the allocations of previous tasks at t to earlier slots, the $\overline{W}(t)$ idle machines are allocated to T_i (line 6) and t becomes fully utilized again; hence, (iv) all slots in $[t_1 + 1, d_i]$ are still fully utilized upon completion of Fully-Allocate(i). Due to (iii)-(iv), we have all slots in $[t_2 + 1, d_i]$ are fully utilized. Finally the proposition holds. □

3.2.3 Phase 2

AllocateRLM(i, η_1, x). Now, we introduce AllocateRLM(i, η_1, x), presented as Algorithm 4; please notice that Algorithm 4 is explained by its comments.

In this subsection, $\eta_1 = 1$ and $x = t_2 + 1$. The resource allocation states before executing AllocateRLM($\cdot$) are described in Lemma 8 and illustrated in the 2nd subfigure of Fig. 3.3; here, all slots in $[t_2 + 1, d_i]$ are fully utilized and every slot in $[1, t_2]$ is not fully utilized; the allocation state of T_i is illsutrated by the green area. AllocateRLM($\cdot$) transfers the allocations of T_i at the earliest slots in $[1, t_2]$ to the slots in $[t_2 + 1, d_i]$ closest to d_i; this is achieved by transferring the allocations of previous tasks in $[t_2 + 1, d_i]$ to the slots earlier than but closest to $t_2 + 1$. In particular, in AllocateRLM($\cdot$), every slot t is considered from d_i to $t_2 + 1$ (line 1). At each t, let t' be the latest slot in $[1, t_2]$ such that it is not fully utilized, i.e., $\overline{W}(t') > 0$, and it transfers the allocations of previous tasks from t to t' leaving some machines idle at t (line 9 of AllocateRLM($\cdot$) and lines 1, 2, 12, 13, 14 of Routine($\cdot$)); then, it allocates $\overline{W}(t)$ more machines to T_i at t and equivalently reduces its allocations at the earliest slots (line 10). Upon its completion, the resource allocation states of T_i is illustrated by the green area in the 3rd subfigure of Fig. 3.3.

Here, at every slot t, if the conditions in line 6 of AllocateRLM($\cdot$) and lines 4, 10 of Routine($\cdot$) are false, we have (i) $\sum_{\bar{t}=1}^{t-1} y_i(\bar{t}) > 0$ and $y_i(t) < k_i$, and (ii) $\overline{W}(t') > 0$ and $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$. We thus have (i) $\overline{W}(t) = 0$ and (ii) $y_i(t') = k_i$; otherwise, this contradicts Lemma 8. Hence, with Lemma 7, we conclude that the task $T_{i'}$ in line 12 of Routine($\cdot$) exists.

Based on our comments in Algorithm 4, we conclude that

Proposition 3. *Upon completion of AllocateRLM(i, 1, x) where $x = t_2 + 1$, the final allocation to $\mathcal{S}''$ can guarantee that Property C.4 holds where $\mathcal{S}'' = \mathcal{S}' \cup \{T_i\}$.*

Proof. Fully-Utilize(i), Fully-Allocate(i) and AllocateRLM(i, η_1, x) never change the allocation at any slot in $[\tau_m + 1, d]$. AllocateRLM(i, 1, x) ends up with one of the following four events. *The 1st event* occurs when the condition in line 4 of Routine($\cdot$) is true; then, the proposition holds trivially since all the slots $\bar{t} \in [1, d_i]$ have been fully utilized, i.e., $\overline{W}(\bar{t}) = 0$.

If the 1st event doesn't occur, there exists a slot in $[1, d_i]$ such that it is not fully utilized. Before executing AllocateRLM($\cdot$), the resource allocation state satisfies Lemma 8: **(a)** the slots in $[t_2 + 1, d_i]$ are fully utilized and $\overline{W}(\bar{t} - 1) \geq \overline{W}(\bar{t}) > 0$ for all $\bar{t} \in [2, t_1]$, and **(b)** given any $t \in [1, t_1]$, if $\sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) < D_i$, we have $y_i(t) = k_i$. The executing process of AllocateRLM($\cdot$) is as follows. **(1)** It considers every slot t from d_i to $t_2 + 1$ (line 1). **(2)** It transfers the allocations of previous tasks at t to an earlier slot t' that is closest to t and not fully utilized currently (line 9); then, the total allocation to all tasks at t' is increased and $\overline{W}(t')$ may become zero; here, some of the slots in $[1, t_1]$ may become fully utilized one by one from t_2 towards the earlier ones. On the other hand, after transferring the allocations of previous tasks at t to earlier slots, **(3)** the $\overline{W}(t)$ idle machines are allocated to T_i (line 10) and t becomes fully utilized again; the allocations of T_i at the earliest slots in $[1, t' - 1]$ are equivalently reduced, ensuring that T_i is exactly allocated D_i resource. As a result, when the resource allocation state upon completion of AllocateRLM($\cdot$) is compared with the state

Algorithm 4: AllocateRLM(i, η_1, x)

```
/* AllocateRLM(·) transfers the allocation of $T_i$ at the earliest slots in
   $[1, d_i]$ to the slots in $[x, d_i]$ that are closest to $d_i$            */
/* in particular, for every slot $t$ from $d_i$ to $x$, it transfers the
   allocations of previous tasks at $t$ to earlier slots that are not fully
   utilized (see lines 1, 5, 9 below); then, allocate more machines to $T_i$
   at $t$ and equivalently reduce its allocation at the earliest slots (line
   10)                                                                          */
/* in the case that $\eta_1 = 1$, it stops when (i) $t = x - 1$ (line 1), (ii) the
   total allocation of $T_i$ in $[1, t-1]$ becomes zero (lines 2-3), (iii) the
   total allocation of $T_i$ in $[1, t'-1]$ becomes zero (lines 10-11 of
   Routine(·) and lines 11-12), or (iv) all slots in $[1, t-1]$ are fully
   utilized (lines 4-5 of Routine(·) and lines 11-12)                          */
1  for $t \leftarrow d_i$ to $x$ do
       // execute the following operations at every slot $t \in [x, d_i]$
2      if $\sum_{\bar{t}=1}^{t-1} y_i(\bar{t}) = 0$ then
           // AllocateRLM(·) stops when $T_i$ is fully allocated in $[t, d_i]$.
3          exit
4      else
5          $\Delta \leftarrow \min\{k_i - y_i(t), \sum_{\bar{t}=1}^{t-1} y_i(\bar{t})\}$    // $\Delta$ denotes the maximum
            allocation of $T_i$ before $t$ that can be transferred to $t$ with the
            parallelism constraint.
6          if $\Delta = 0$ then
7              continue                                  // go to line 1 if $\Delta = 0$
8          else
9              $flag \leftarrow 0$, and call Routine($\Delta$, $\eta_1$, 1, $t$)    // it increases the
                number $\overline{W}(t)$ of available machines at $t$ to $\Delta$ if the conditions
                in lines 4 and 10 (or 7 and 10) of Routine(·) are false.
10             set $\theta \leftarrow \overline{W}(t)$, and $y_i(t) \leftarrow y_i(t) + \theta$; let $t''$ be such a slot that
                $\sum_{\bar{t}=1}^{t''-1} y_i(\bar{t}) < \theta$ and $\sum_{\bar{t}=1}^{t''} y_i(\bar{t}) \geq \theta$: (i) $y_i(\bar{t}) \leftarrow 0$ for all
                $\bar{t} \in [1, t''-1]$ and $\theta \leftarrow \theta - \sum_{\bar{t}=1}^{t''-1} y_i(\bar{t})$, and (ii)
                $y_i(t'') \leftarrow y_i(t'') - \theta$.
11             if $flag = 1$ then
12                 exit
```

before executing Allocate-B(i), we have

(i) the allocation to the previously allocated tasks $\mathcal{S}'$ at every $\bar{t} \in [1, t'-1]$ does not change;

(ii) the allocation to $\mathcal{S}'$ at t' is not decreased;

(iii) the slots in $[t'+1, d_i]$ are fully utilized, i.e., $\overline{W}(\bar{t}) = 0$ for all $\bar{t} \in [t'+1, d_i]$, where $d_i = \tau_m$.

The 2nd event that makes AllocateRLM($\cdot$) stop is the condition in line 2 of AllocateRLM($\cdot$) is true; then, T_i is fully allocated D_i resource in $[t, d_i] \subseteq [t_2+1, d_i]$. *The 3rd event* is the condition in line 10 of Routine($\cdot$) is true; then, T_i will be fully allocated D_i resource in $[t', d_i] \supseteq [t_2+1, d_i]$. When the 2nd or 3rd event happens, we have

(iv) T_i is fully allocated D_i resource in $[t', d_i]$;

The 4th event occurs upon completion of the iteration of AllocateRLM($\cdot$) at $t = t_2+1$. In this case, we have that the conditions in line 2 of AllocateRLM($\cdot$) and lines 4 and 10 of Routine($\cdot$) are always false; then, upon completion of AllocateRLM($\cdot$), we still have $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$; recall the executing process of AllocateRLM($\cdot$) (described in (1)-(3)) and the resource allocation state before executing it (described in (a)-(b)); then, during executing each iteration of AllocateRLM($\cdot$) at $t \in [t_2+1, d_i]$, we always have $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$ and upon completion of Routine($\cdot$) (line 9), we have $\overline{W}(t) = \Delta = k_i - y_i(t)$; T_i is finally allocated k_i machines at every t (line 10). Let $t''' \in [1, t'-1] \subseteq [1, t_2]$ denote the earliest slot such that $y_i(t''') \neq 0$ and we have:

(v) T_i is fully allocated D_i resource in $[t''', d_i]$;

(vi) for all $\bar{t} \in [t'''+1, t']$, the allocation to T_i at $\bar{t}$ is k_i, i.e., $y_i(\bar{t}) = k_i$.

Now, we first show that, when the 2nd, 3rd or 4th event happens, Property C.2 holds. When the 2nd or 3rd event happens, by (iv), we have $0 = y_i(1) = \cdots = y_i(t'-1) \leq y_i(t')$; when the 4th event happens, by (v) and (vi), we have $0 = y_i(1) = \cdots = y_i(t'''-1) < y_i(t''') \leq y_i(t'''+1) = \cdots = y_i(t') = k_i$. Now, we observe the resource allocation state of $\mathcal{S}'$ in $[1, t']$ described in (i)-(ii); then, no matter which of the 2nd, 3rd, and 4th events happens, we have $W(1) \leq \cdots \leq W(t')$ since Property C.2 holds before executing Allocate-B(i) where $t' \leq t_2$. As a result, we conclude that Property C.2 still holds upon completion of AllocateRLM($\cdot$).

Next, we show that Property C.1 holds, i.e., each interval $[\tau_l+1, d]$ is optimally utilized by $\mathcal{S}' \cup \{T_i\}$ for all $l \in [L-1]$. The allocation to $\mathcal{S}'$ in $[\tau_m+1, d]$ does not change where the deadline d_i of T_i is τ_m; we have for all $l \in [m, L-1]$ that the interval $[\tau_l+1, d]$ is optimally utilized by $\mathcal{S}' \cup \{T_i\}$ due to Property C.1. Let $m', m'' \in [m]^+$ be such that $t' \in [\tau_{m'-1}+1, \tau_{m'}]$ and $t''' \in [\tau_{m''-1}+1, \tau_{m'}]$. By (iii), all slots in $[\tau_{m'}+1, d_i]$ have been fully utilized; hence, we have for all $l \in [m', m-1]$ that every interval $[\tau_l+1, d]$ is optimally utilized by $\mathcal{S}' \cup \{T_i\}$, by Definitions 3 and 4. Now, we show for all $l \in [0, m'-1]$ that $[\tau_l+1, d]$ is optimally utilized by $\mathcal{S}' \cup \{T_i\}$. It suffices to show that (a) the maximum demand of T_i is processed on C machines in $[\tau_l+1, d]$, and (b) the maximum demand of $\mathcal{S}'$ is processed on C machines in $[\tau_l+1, d]$; then, by Lemma 3, we have $[\tau_l+1, d]$ is optimally utilized by $\mathcal{S}' \cup \{T_i\}$. When the 2nd or 3rd event happens, the maximum demand of T_i is processed in $[\tau_l+1, d]$ since $\sum_{\bar{t}=\tau_l+1}^{d} y_i(\bar{t}) = D_i$ by (iv); when the 4th event happens, it also holds since $0 = y_i(1) = \cdots = y_i(t'''-1) < y_i(t''') \leq y_i(t'''+1) = \cdots = y_i(d_i) = k_i$ and

$\sum_{\bar{t}=t'''}^{d_i} y_i(\bar{t}) = D_i$. On the other hand, by (i), the total allocation to $\mathcal{S}'$ in $[1, \tau_l]$ isn't changed by Allocate-B(i), and the maximum workload of $\mathcal{S}'$ will be processed in $[\tau_l + 1, d]$. Finally, Property C.1 holds. $\square$

Propositions 2 and 3 show that Allocate-B(i) satisfies Properties C.3 and C.4 and hence completes the proof of Proposition 1. We finally analyze the time complexity of Allocate-B(i).

Lemma 9. *The time complexity of Allocate-B($\cdot$) is $\mathcal{O}(n)$.*

Proof. The time complexity of Allocate-B(i) depends on Fully-Allocate(i) or AllocateRLM($\cdot$). In the worst case, Fully-Allocate(i) and AllocateRLM($\cdot$) have the same time complexity from the execution of Routine($\cdot$) at every time slot $t \in [1, d_i]$. In AllocateRLM($\cdot$) for every task $T_i \in \mathcal{T}$, each loop iteration at $t \in [1, d_i]$ needs to seek the time slot t' and the task $T_{i'}$ at most D_i times. The time complexities of respectively seeking t' and $T_{i'}$ are $\mathcal{O}(d)$ and $\mathcal{O}(n)$; the maximum of these two complexities is $\max\{d, n\}$. Since $d_i \leq d$ and $D_i \leq D$, we have that both the time complexity of Allocate-B(i) is $\mathcal{O}(dD \max\{d, n\})$. Since we assume that d and k are finitely bounded where $D \leq d \cdot k$, we conclude that $\mathcal{O}(dD \max\{d, n\}) = \mathcal{O}(n)$. $\square$

Since LDF($\mathcal{S}$) considers a total of n tasks, its complexity is $\mathcal{O}(n^2)$ with Lemma 9. Finally, we draw a main conclusion in this chapter from Lemma 1 and Proposition 1:

Theorem 1. *A set of tasks $\mathcal{S}$ can be feasibly scheduled and be completed by their deadlines on C machines* if and only if *the boundary condition holds, where the feasible schedule of $\mathcal{S}$ could be produced by LDF($\mathcal{S}$) with a time complexity $\mathcal{O}(n^2)$.*

In other words, if LDF($\mathcal{S}$) cannot produce a feasible schedule for $\mathcal{S}$ on C machines, then $\mathcal{S}$ cannot be successfully scheduled by any algorithm; as a result, LDF($\mathcal{S}$) is optimal. The relationships between the various algorithms in Chapters 2-3 are illustrated in Fig. 3.4 where GreedyRLM will be introduced in the next chapter.

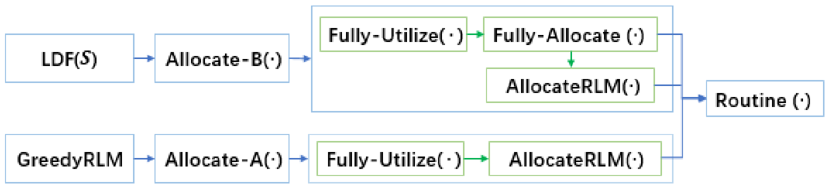

Figure 3.4 – Relationship among Algorithms: for $A \rightarrow B$, the blue and green arrows denote the relations that the algorithm A calls B, and, the algorithm B is executed upon completion of A.

Remarks. We are inspired by the GreedyRTL algorithm [3] in the construction of LDF($\cdot$). In terms of the two algorithms themselves, LDF($\cdot$) considers tasks in the decreasing order of deadlines while the order is determined by the marginal values in GreedyRTL$(\cdot)$. In both algorithms, the allocation to a task T_i is considered from d_i to 1 (once in GreedyRTL, and possibly three times in LDF$(\cdot)$); to make time slots t closest to the deadline of a task T_i being considered fully utilized, the key operations are finding a time slot t' earlier than t such that there exists a task $T_{i'}$ with $y_{i'}(t) > y_{i'}(t')$ when $\overline{W}(t)$, and transferring a part of the allocation of $T_{i'}$ at t to t'. In GreedyRTL$(\cdot)$, the existence of $T_{i'}$ requires that (i) the number $\overline{W}(t')$ of available machines at t' is $\geq k$ and (ii) [1] $\overline{W}(t) < k_i$; as a result, before doing any allocation to T_i at t, the existence could be proved by contradiction. In LDF$(\cdot)$, to achieve the optimality of resource utilization, one requirement for such existence is relaxed to be that the number of available machines at t' is ≥ 1. The existence is guaranteed by (i) first make every time slot from d_i to 1 fully utilized, as what Fully-Utilize(i) does, and (ii) a stepped-shape resource utilization state in $[1, d_i]$ upon completion of the allocation to the last task, as described in Property C.2.

1. The particular condition there is $\overline{W}(t) < \min\{k_i, D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t})\}$.

Chapter 4

Scheduling with Various Objectives

The majority see the obstacles; the few see the objectives; history records the successes of the latter, while oblivion is the reward of the former.

Alfred A. Montapert

In this chapter, we illustrate the applications of the results in Chapter 3 to (i) two algorithmic design techniques for the social welfare maximization objective in [2], [3], giving the optimal greedy algorithm and the first exact dynamic programming algorithm and (ii) two other objectives.

4.1 Greedy Algorithm

In this section, we illustrate the application of the results in Chapter 3 to the greedy algorithm for social welfare maximization. In terms of the maximization problem, *the general form of a greedy algorithm* is as follows [20], [22]: it tries to build a solution by iteratively executing the following steps until no item remains to be considered in a set of items: (1) selection standard: in a greedy way, choose and consider an item that is locally optimal according to a simple criterion at the current stage; (2) feasibility condition: for the item being considered, accept it if it satisfies a certain condition such that this item constitutes a feasible solution together with the items that have been accepted so far under the constraints of this problem, and reject it otherwise. Here, an item that has been considered and rejected will never be considered again. The selection criterion is related to the objective function and constraints, and is usually the ratio of 'advantage' to 'cost', measuring the efficiency of an item. In the problem of this thesis, the constraint comes from the capacity to hold the chosen tasks and the objective is to maximize the social welfare; therefore, the selection criterion here is the ratio of the value of a task to its demand, which we refer to as the *marginal value* of this task. Formally, the marginal value of a task T_i is defined as $v'_i = \frac{v_i}{D_i}$, i.e., the value obtained from per unit of demand when T_i is completed by its deadline.

Given the general form of greedy algorithm, we define a class GREEDY of algorithms that operate as follows:

1. Considers the tasks in the decreasing order of the marginal values; assume without loss of generality that $v'_1 \geq v'_2 \geq \cdots \geq v'_n$;
2. Denoting by $\mathcal{A}$ the set of the tasks that have been accepted so far, a task T_i being considered is accepted and fully allocated *iff* there exists a feasible schedule for $\mathcal{A} \cup \{T_i\}$.

In the following, we refer to the generic algorithm in GREEDY as *Greedy.* Recall that len_i is the (minimum) execution time of a task T_i when it always utilizes k_i machines throughout the execution where $len_i = \left\lceil \frac{D_i}{k_i} \right\rceil$. We define by $s_i = \frac{d_i}{len_i}$ the slackness of T_i that measures the urgency or flexibility of machine allocation where $s_i \geq 1$, e.g., $s_i = 1$ may mean that T_i should always utilize the maximum number of machines k_i from the slot 1 until its completion in order to meet the deadline; we let $s = \min_{T_i \in \mathcal{T}}\{s_i\}$ denote the slackness of the least flexible task. Besides the notation in Chapter 2.4, the additional key notation used for this section is also summarized in Table 4.1.

Proposition 4. *The best performance guarantee that a greedy algorithm in GREEDY can achieve is* $\frac{s-1}{s}$.

Proof. Let us consider a special instance: (i) let $\mathcal{D}_i = \{T_j \in \mathcal{T} | d_i = d'_i\}$, where $i \in [2]^+$, d'_2 and $d'_1 \in \mathcal{Z}^+$, and $d'_2 > d'_1$; (ii) for all $T_j \in \mathcal{D}_1$, $v'_j = 1 + \epsilon$, $D_j = 1$, $k_j = 1$, and, there is a total of $C \cdot d'_1$ such tasks, where $\epsilon \in (0, 1)$ is small enough; (iii) for all $T_j \in \mathcal{D}_2$, $v'_j = 1$, $k_j = 1$ and $D_j = d'_2 - d'_1 + 1$. Greedy will always fully allocate resource to the tasks in $\mathcal{D}_1$, with all the tasks in $\mathcal{D}_2$ rejected to be allocated any resource. The performance guarantee of Greedy will be no more than $\frac{C \cdot d'_1}{C \cdot [(1+\epsilon)(d'_1-1)+1\cdot(d'_2-d'_1+1)]}$. Further, with $\epsilon \rightarrow 0$, this performance guarantee approaches $\frac{d'_1}{d'_2}$. In this instance, $s = \frac{d'_2}{d'_2-d'_1+1}$ and $\frac{s-1}{s} = \frac{d'_1-1}{d'_2}$. When $d'_2 \rightarrow +\infty$, $\frac{d'_1}{d'_2} = \frac{s-1}{s}$. Hence, the proposition holds. $\square$

4.1.1 The Executing Process of Greedy

Greedy will consider tasks sequentially. The first considered task will be accepted definitely and then it will use the feasibility condition to determine whether or not to accept or reject the next task according to the current available resource and the characteristics of this task. To describe the process under which Greedy accepts or rejects tasks, we define the sets of consecutive accepted (i.e., fully allocated) and rejected tasks $\mathcal{A}_1, \mathcal{R}_1, \mathcal{A}_2, \cdots$. Specifically, let $\mathcal{A}_m = \{T_{i_m}, T_{i_m+1}, \cdots, T_{j_m}\}$ be the m-th set of the adjacent tasks that are accepted by Greedy where $i_1 = 1$ while $\mathcal{R}_m = \{T_{j_m+1}, \cdots, T_{i_{m+1}-1}\}$ is the m-th set of the adjacent tasks that are rejected following the set $\mathcal{A}_m$, where $m \in [K]^+$ for some integer K. Integer K represents the last step: in the K-th step, $\mathcal{A}_K \neq \emptyset$ and $\mathcal{R}_K$ can be empty or non-empty. We also denote by c_m the maximum deadline of all rejected tasks in the first m phases, i.e.,

$$c_m = \max_{T_i \in \bigcup_{l=1}^{m} \mathcal{R}_l} \{d_i\},$$

and by c'_m the maximum deadline of all accepted tasks in the first m phases, i.e.,

$$c'_m = \max_{T_i \in \bigcup_{l=1}^{m} \mathcal{A}_l} \{d_i\}.$$

While the tasks in $\mathcal{A}_m \cup \mathcal{R}_m$ are being considered, we refer to Greedy as being in the m-th phase. Before the execution of Greedy, we refer to it as being in the 0-th phase. Upon completion of the m-th phase of Greedy, we define a *threshold* parameter t_m^{th} such that

(i) if $c_m \geq c'_m$, set $t_m^{\text{th}} = c_m$, and

(ii) if $c_m < c'_m$, set t_m^{th} to some time slot in $[c_m, c'_m]$;

here, $d_i \leq t_m^{th}$ for all $T_i \in \cup_{j=1}^{m} \mathcal{R}_j$. For ease of the subsequent exposition, we let $t_0^{th} = 0$ and $t_{K+1}^{th} = d$. We also add a dummy time slot 0 but the task $T_i \in \mathcal{T}$ can not get any resource there, that is, $y_i(0) = 0$ forever. We also let $\mathcal{A}_0 = \mathcal{R}_0 = \mathcal{A}_{K+1} = \mathcal{R}_{K+1} = \emptyset$.

4.1.2 A New Algorithmic Analysis

We will show that as soon as the resource allocation done by Greedy satisfies some features, its performance guarantee can be deduced immediately, i.e., the main result of this subsection is Theorem 2.

For all $m \in [K]^+$, upon completion of Greedy, we define the following two features that we want the allocation to $\cup_{j=1}^{m} \mathcal{A}_j$ to satisfy. The first feature says that the utilization of the C machines achieved by the allocation to $\bigcup_{j=1}^{m} \mathcal{A}_j$ in $[1, t_m^{th}]$ is no smaller than r, where $r \in [0, 1]$:

***Feature* 4.1.1.** *The total allocation to $\bigcup_{j=1}^{m} \mathcal{A}_j$ in $[1, t_m^{th}]$ is at least $r \cdot C \cdot t_m^{th}$.*

There are C machines on which tasks are executed and each task has the paralellelism and deadline constraints. As elaborated in Chapter 3.1, given any slot $t \in [1, d]$, these factors constrain the maximum workload of $\cup_{j=1}^{m} \mathcal{A}_j$ that could be processed in $[t, d]$ on C machines, denoted by $\lambda_t^C \left(\cup_{j=1}^{m} \mathcal{A}_j\right)$. The second feature is as follows.

Table 4.1 – Main Notation for Chapter 4

Notation	Explanation
$\mathcal{A}_1, \mathcal{R}_1, \mathcal{A}_2, \cdots, \mathcal{R}_K$	the sets of consecutive accepted (i.e., fully allocated) and rejected tasks by Greedy where $\bigcup_{m=1}^{K} \mathcal{A}_m \cup \mathcal{R}_m = \mathcal{T}$
c_m	the maximum deadline of all rejected tasks of $\cup_{l=1}^{m}\mathcal{R}_l$
c'_m	the maximum deadline of $\cup_{l=1}^{m}\mathcal{A}_l$
t_m^{th}	a threshold parameter such that (i) if $c_m \geq c'_m$, set $t_m^{\text{th}} = c_m$, and (ii) if $c_m < c'_m$, set t_m^{th} to any time slot in $[c_m, c'_m]$; when introducing GreedyRLM, it will be set to a specific value
v'_i	the marginal value of T_i, i.e., $v'_i = \frac{v_i}{D_i}$
len_i	the (minimum) execution time of T_i when T_i always utilizes k_i machines throughout the execution, i.e., $len_i = \lceil \frac{D_i}{k_i} \rceil$
s_i	the slackness of a task, i.e., $\frac{d_i}{len_i}$, measuring the urgency of machine allocation to complete T_i by the deadline
s	the minimum slackness of all tasks of $\mathcal{T}$, i.e., $\min_{T_i \in \mathcal{T}}\{s_i\}$

***Feature* 4.1.2.** *Upon completion of Greedy, the interval $\left[t_m^{th}+1, d\right]$ is optimally utilized by $\bigcup_{j=1}^{m} \mathcal{A}_j$, i.e., $\lambda_t^C \left(\cup_{j=1}^{m}\mathcal{A}_j\right)$ workload of $\cup_{j=1}^{m}\mathcal{A}_j$ is processed in $\left[t_m^{th}+1, d\right]$.*

Theorem 2. *If Greedy achieves a resource allocation state that satisfies Feature 4.1.1 and Feature 4.1.2 for all $m \in [K]^+$, it gives a r-approximation to the optimal social welfare.*

In the rest of this subsection, we prove Theorem 2. We refer to the original problem of scheduling $\mathcal{A}_1$, $\mathcal{R}_1$, $\cdots$, $\mathcal{A}_K$, $\mathcal{R}_K$ on C machines to Maximize the Social Welfare as **the MSW-I problem**.

In the following, we define a relaxed version of the MSW-I problem whose optimal social welfare is an upper bound of the optimal of the MSW-I problem. There are C machines and a set of tasks including $\mathcal{A}_1$, $\mathcal{R}'_1$, $\mathcal{A}_2$, $\cdots$, $\mathcal{R}'_K$. Here, $\mathcal{R}'_m$ only consists of a single task T'_m whose deadline is t_m^{th}, whose size is infinite, and whose marginal value is the largest one of the tasks in $\mathcal{R}_m$, denoted by $\hat{v}'_m$; here, different from the tasks of $\mathcal{R}_m$, we assume that there is no parallelism constraint on T'_m, i.e., the parallelism bound is C, and t_m^{th} is $\geq$ the task deadlines of $\mathcal{R}_m$. The task characteristics of $\mathcal{A}_1, \cdots, \mathcal{A}_K$ are the same as the ones in the MSW-I problem. However, partial execution of any task can yield linearly proportional value, e.g., if a task $T_i \in \mathcal{A}_l$ is allocated $\sum_{t=1}^{d_i} y_i(t) < D_i$ resource by its deadline, a value $(\sum_{t=1}^{d_i} y_i(t)/D_i) \cdot v_i$ will still be added to the social welfare. We refer to the problem of scheduling $\mathcal{A}_1, \mathcal{R}'_1, \cdots, \mathcal{A}_K, \mathcal{R}'_K$ on C machines as **the MSW-II problem**.

Lemma 10. *The optimal social welfare of the MSW-II problem is an upper bound of the optimal social welfare of the MSW-I problem.*

Proof. Let us consider an optimal allocation to $\mathcal{A}_1, \mathcal{R}_1, \cdots, \mathcal{A}_K, \mathcal{R}_K$ for the MSW-I problem. If we replace an allocation to a task in $\mathcal{R}_m$ with the same allocation to a task in $\mathcal{R}'_m$ and do not change the allocation to $\mathcal{A}_m$, this generates a feasible schedule for the MSW-II problem, which yields at least the same social welfare since the marginal value of the task in $\mathcal{R}'_m$ is $\geq$ the ones of the tasks in $\mathcal{R}_m$; hence, Lemma 10 holds. $\square$

Due to Feature 4.1.1, Feature 4.1.2, and the fact that the marginal value of T'_m is smaller than the ones of the tasks of $\cup_{l=1}^{m}\mathcal{A}_l$, we derive the following two lemmas:

Lemma 11. *The following schedule that may violate the capacity constraint achieves an upper bound of the optimal social welfare of the MSW-II problem:*

1. *for all tasks of $\mathcal{A}_1, \cdots, \mathcal{A}_K$, their allocation is the same as the one achieved by Greedy with Features 4.1.1 and 4.1.2 satisfied;*
2. *for all $m \in [K]^+$, execute a part of the task T'_m such that the amount of workload processed in $\left[t_{m-1}^{th}+1, t_m^{th}\right]$ is $\alpha_m \cdot C$ where $\alpha_m = (1-r)\cdot\left(t_m^{th} - t_{m-1}^{th}\right)$.*

Proof. We will show in an optimal schedule of the MSW-II problem that (i) only the tasks of $\mathcal{R}'_m$, $\mathcal{A}_1, \cdots, \mathcal{A}_K$ will be executed in $[t_{m-1}^{th}+1, t_m^{th}]$ for all $m \in [K]^+$, and (ii) each interval $[\tau_m + 1, d]$ is optimally utilized by $\cup_{j=1}^{K}\mathcal{A}_j$ for all $m \in [K]^+$; the latter uniquely determines the amount of resources occupied by $\cup_{j=1}^{K}\mathcal{A}_j$ in each interval. When the machines are optimally utilized (i.e., Feature 4.1.2 holds), we have that (iii) Feature 4.1.1 also holds here. With the three points above, an optimal schedule of the MSW-II problem is as follows: (a) the allocation of $\bigcup_{j=1}^{K}\mathcal{A}_j$ satisfies Property 4.1.2 and 4.1.1 and (b) afterwards, execute the workload of T'_m as much as possible in the interval $[\tau_{m-1}+1, \tau_m]$ for all $m \in [K]^+$ such that all the remaining idle resource in this interval is utilized; let β_m denote the amount of the workload of T'_m processed in $[\tau_{m-1}+1, \tau_m]$. By Property 4.1.1, the allocation to $\bigcup_{j=1}^{m}\mathcal{A}_j$ achieves a resource utilization $\geq r$ in each interval $[1, \tau_m]$ for all $m \in [K]^+$; we thus have $\sum_{l=1}^{m}\alpha_m = t_m^{\mathrm{th}}\cdot(1-r)\cdot C \geq \sum_{l=1}^{m}\beta_m$.

In the optimal schedule of the MSW-II problem, the total gain from executing $T'_1, \cdots, T'_K$ is $\sum_{m=1}^{K}\beta_m\cdot\hat{v}'_m$; its upper bound can be the optimal value of the following maximization problem where we view $\beta_1, \cdots, \beta_K$ as variables:

$$\text{maximize} \sum_{m=1}^{K}\beta_m\cdot\hat{v}'_m \tag{4.1}$$

subject to

$$\begin{aligned} \beta_1 &\leq t_1^{\mathrm{th}}\cdot(1-r)\cdot C \\ \beta_1+\beta_2 &\leq t_2^{\mathrm{th}}\cdot(1-r)\cdot C \\ &\cdots \\ \beta_1+\beta_2+\cdots+\beta_K &\leq t_K^{\mathrm{th}}\cdot(1-r)\cdot C \end{aligned} \tag{4.2}$$

and

$$\overline{v}'_1 > \overline{v}'_2 > \cdots > \overline{v}'_K \tag{4.3}$$

Due to (4.3), an optimal solution to (4.1) is to make β_1, then β_2, $\cdots$, finally β_K as large as possible with the constraint (4.2), leading to $\beta_1 = \alpha_1$, $\cdots$, $\beta_K = \alpha_K$. Hence, the total value generated by executing all tasks of $\mathcal{A}_1, \cdots, \mathcal{A}_K$ and $(1-r)\cdot(t_m^{th} - t_{m-1}^{th})\cdot C$ workload of each $\mathcal{R}'_m$ ($m \in [K]^+$) is an upper bound of the optimal social welfare for the MSW-II problem and the lemma holds.

Now, we prove the first point mentioned at the beginning of this proof by contradiction. Given a $m \in [K]^+$, if $m \geq 2$, all tasks of $\mathcal{R}'_1, \cdots, \mathcal{R}'_{m-1}$ could not be processed in $[t_{m-1}^{th}+1, t_m^{th}]$ due to the deadline constraint. If $m \leq K-1$, the marginal value of the task in $\mathcal{R}'_m$ is larger than the ones of $\mathcal{R}'_{m+1}, \cdots, \mathcal{R}'_K$; instead of processing $\mathcal{R}'_{m+1}, \cdots, \mathcal{R}'_K$ in $[t_{m-1}^{th}+1, t_m^{th}]$, processing $\mathcal{R}'_m$ could generate a higher value. Hence, the first point holds. The second point is also proved by contradiction. In an optimal schedule if there exists a $m \in [K]^+$ such that the interval $\left[t_m^{\text{th}}+1, d\right]$ is not optimally utilized by $\cup_{j=1}^m \mathcal{A}_j$, the following operations will improve the gain of this schedule, which contradicts the fact that this schedule is optimal: (1) transfer a unit of workload of $\cup_{j=1}^m \mathcal{A}_j$ in $[1, t_m^{\text{th}}]$ to $\left[t_m^{\text{th}}+1, d\right]$, (2) equivalently reduce the allocation of $T'_{m+1}, \cdots, T'_K$ in $\left[t_m^{\text{th}}+1, d\right]$ by one if $m \leq K-1$, and (3) execute a unit of extra workload of $\cup_{j=1}^m \mathcal{R}'_j$ in $[1, t_m^{\text{th}}]$. □

Lemma 12. *In the schedule of Lemma 11, we have for all $m \in [K]^+$ that the total value generated by executing $T'_1, \cdots, T'_m$ is no larger than $\frac{1-r}{r}$ times the total value generated by the allocation to $\cup_{l=1}^m \mathcal{A}_l$ in $\left[1, t_m^{th}\right]$.*

Proof. There are C machines and the allocation of a task T_i at t is $y_i(t) \in \{0, 1, \cdots, k_i\}$. Let $x_i(t, c) \in \{0, 1\}$ denote the allocation of T_i at slot t on the c-th machine; here, $y_i(t) = \sum_{c=1}^C x_i(t, c)$. Let $\mathcal{U}$ denote the allocations of all tasks of $\cup_{l=1}^m \mathcal{A}_l$ in $\left[1, t_m^{th}\right]$, i.e.,

$$\mathcal{U} = \left\{x_i(t,c) \,|\, T_i \in \cup_{j=1}^m \mathcal{A}_j, t \in [t_m^{\text{th}}]^+, c \in [C]^+, x_i(t,c) = 1\right\}.$$

An allocation $x_i(t, c)$ will generate a value $v'_i \cdot x_i(t, c)$, and the total value generated from $\cup_{j=1}^m \mathcal{A}_j$ is $\sum_{x_i(t,c)\in\mathcal{U}} v'_i \cdot x_i(t, c)$; since the allocaiton of $\cup_{l=1}^m \mathcal{A}_l$ achieves a utilization $\geq r$ in $\left[1, t_m^{\text{th}}\right]$, we have that $|\mathcal{U}| \geq r \cdot t_m^{\text{th}} \cdot C$.

By Property 4.1.1, the allocation of $\cup_{j=1}^l \mathcal{A}_j$ achieves a utilization $\geq r$ in $[1, t_l^{\text{th}}]$ for all $l \in [m]^+$. This enables defining a series of sets of allocations as follows: (i) $\mathcal{U}_1$ is an arbitrary subset of

$$\left\{x_i(t,c) \,|\, T_i \in \mathcal{A}_1, t \in \left[t_1^{\text{th}}\right]^+, c \in [C]^+, x_i(t,c) = 1\right\}$$

such that $|\mathcal{U}_1| = r \cdot t_1^{\text{th}} \cdot C$, and (ii) for all $l \in [2, m]$, $\mathcal{U}_l$ is an arbitrary subset of

$$\left\{x_i(t,c) \,|\, T_i \in \cup_{j=1}^l \mathcal{A}_j, t \in \left[t_l^{\text{th}}\right]^+, c \in [C]^+, x_i(t,c) = 1\right\} - \bigcup_{j=1}^{l-1} \mathcal{U}_j$$

such that $|\mathcal{U}_l| = r \cdot \left(t_l^{\text{th}} - t_{l-1}^{\text{th}}\right) \cdot C$. Here, $\mathcal{U}_l$ is composed of the allocations of the tasks of $\cup_{j=1}^{l}\mathcal{A}_j$; $\mathcal{U}_1, \cdots, \mathcal{U}_m$ are disjoint sets and their union is a subset of $\mathcal{U}$, with which we have that

$$\textstyle\sum_{x_i(t,c)\in\mathcal{U}} v_i' \cdot x_i(t,c) \geq \sum_{x_i(t,c)\in\mathcal{U}_1} v_i' \cdot x_i(t,c) + \cdots + \sum_{x_i(t,c)\in\mathcal{U}_m} v_i' \cdot x_i(t,c).$$

In the schedule of Lemma 11, the total value generated by executing $T_1', \cdots, T_m'$ is $\alpha_1 \cdot \hat{v}_1' + \alpha_2 \cdot \hat{v}_2' + \cdots + \alpha_m \cdot \hat{v}_m'$ where $\frac{\alpha_l}{1-r} = \frac{|\mathcal{U}_l|}{r}$. For all $l \in [m-1]^+$, the marginal values of tasks of $\mathcal{A}_l$ are larger than the marginal value of T_l' that are further larger than the ones of $\mathcal{A}_{l+1}$; thus, for all $l \in [m]^+$, considering the allocation composition of $\mathcal{U}_l$, we have that

$$\sum_{x_i(t,c)\in\mathcal{U}_l} v_i' \cdot \tfrac{x_i(t,c)}{r} > \sum_{x_i(t,c)\in\mathcal{U}_l} \hat{v}_l' \cdot \tfrac{x_i(t,c)}{r} = \hat{v}_l' \cdot \tfrac{|\mathcal{U}_l|}{r} > \hat{v}_l' \cdot \tfrac{\alpha_l}{1-r},$$

that is, $\hat{v}_l' \cdot \alpha_l < \frac{1-r}{r} \cdot \sum_{x_i(t,c)\in\mathcal{U}_l} v_i' \cdot x_i(t,c)$. Hence, we have that

$$\tfrac{1-r}{r} \cdot \sum_{x_i(t,c)\in\mathcal{U}} v_i' \cdot x_i(t,c) \geq \sum_{l=1}^{m} \alpha_l \cdot \hat{v}_l'$$

and the lemma holds. □

In the case that $m = K$, the total value from $T_1', \cdots, T_K'$ is no larger than $\frac{1-r}{r}$ times the total value from the allocation to $\cup_{l=1}^{K}\mathcal{A}_l$ in $[1, t_K^{th}]$. Hence, the total value generated by the schedule in Lemma 11 is no larger than $1 + \frac{1-r}{r} = \frac{1}{r}$ times the total value generated by the allocation to all tasks of $\mathcal{A}_1, \cdots, \mathcal{A}_K$. The totoal value of Greedy is from $\mathcal{A}_1, \cdots, \mathcal{A}_K$; hence, by Lemmas 11 and 10, Theorem 2 holds.

4.1.3 Optimal Algorithm Design

In this subsection, under the guidance of Theorem 2, we propose a best possible greedy algorithm GreedyRLM that achieves an approximation ratio $\frac{s-1}{s}$. An overview of GreedyRLM is as follows and its executing process is presented in Algorithm 5:

(1) consider tasks in the decreasing order of their marginal values (line 2 of Algorithm 5).

(2) in the m-th phase, for a task T_i being considered,

— if the allocation condition is satisfied, i.e.,

$$\textstyle\sum_{t\leq d_i} \min\{\overline{W}(t), k_i\} \geq D_i,$$

then, T_i is accepted and Allocate-A(i) is called to make it fully allocated, presented as Algorithm 6 where the details on Fully-Utilize(i) and AllocateRLM(i, 0, $t_{m-1}^{th} + 2$) can be found in Chapter 3.2.2 and Chapter 3.2.3.

— if the allocation condition is not satisfied, T_i is rejected; when T_i is the last task rejected in the m-th phase, set the correspoding threshold parameter t_m^{th} that is defined by lines 11-14 of Algorithm 5.

Algorithm 5: GreedyRLM

```
Input  : $n$ tasks with $type_i = \{v_i, d_i, D_i, k_j\}$
Output: A feasible allocation of resources to tasks
1  initialize: $y_i(t) \leftarrow 0$ for all $T_i \in \mathcal{T}$ and $t \in [1, d]$, $m = 1$, $t_0^{th} = 0$;
2  sort tasks in the decreasing order of their marginal values:
   $v'_1 > v'_2 > \cdots > v'_n$;
3  $i \leftarrow 1$; // in the 1st phase of Greedy, the first accepted task is $T_1$, i.e.,
   $i_1 = i = 1$
4  while $i \leq n$ do
5      if $\sum_{t \leq d_i} \min\{\overline{W}(t), k_i\} \geq D_i$ then
6          Allocate-A($i$); // allocate resource to the accepted task $T_i$ in the
            $m$-th phase
7          $i \leftarrow i + 1$;
8      else
           /* the first rejected task in the $m$-th phase is the current $T_i$,
              i.e., $j_m + 1 = i$                                          */
9          while $\sum_{t \leq d_{i+1}} \min\{\overline{W}(t), k_{i+1}\} < D_{i+1}$ do
10             $i \leftarrow i + 1$; // when this while loop stops, the current $T_i$ is the
                first accepted task in the $(m+1)$-th phase where $i_{m+1} = i$
11         if $c_m \geq c'_m$ then
12             $t_m^{\mathrm{th}} \leftarrow c_m$;
13         else
14             set $t_m^{\mathrm{th}}$ to the earliest time slot $t$ in $[c_m, c'_m]$ such that there
                exist idle machines at $t + 1$, i.e., $\overline{W}(t + 1) > 0$;
15         $m \leftarrow m + 1$;
```

When the condition in line 5 of GreedyRLM is true, the considered task T_i is accepted and Allocate-A(i) is called where Fully-Utilize(i) and AllocateRLM(i, 0, $t_{m-1}^{th} + 2$) are sequentially executed. In that condition, $\overline{W}(t)$ is the number of available machines idle at t and k_i is the parallelism bound; further, $\min\{\overline{W}(t), k_i\}$ is the maximum number of machines that could be utilized by T_i at slot t and the condition means that the total amount of machines available over $[1, d_i]$ is no smaller than the workload of T_i. So, T_i can be fully allocated D_i resource by Fully-Utilize(i). Here, Fully-Utilize(i) considers every slot t from the deadline d_i to 1 and at slot t,

— T_i is allocated $\min\{\overline{W}(t), k_i\}$ machines if after such an allocation there is still remaining workload to be processed, i.e., $D_i - \sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) > 0$;

— otherwise, T_i is allocated $D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t})$ machines; here, $D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t})$ is the remaining workload to be processed after the allocations of T_i over $[t + 1, d_i]$.

Algorithm 6: Allocate-A(i)

```
/* below, for every slot $t$ from the deadline $d_i$ to 1, Fully-Utilize($i$) sets
   $y_i(t)$ to $\min\{k_i, D_i - \sum_{\bar{t}=t+1}^{d_i} y_i(\bar{t}), \overline{W}(t)\}$; as a result, $T_i$ is fully
   allocated $D_i$ resource since the condition in line 5 of GreedyRLM is
   true                                                                    */
1 Fully-Utilize($i$);
2 if $d_i \geq t_{m-1}^{th} + 2$ then
      /* below, AllocateRLM($\cdot$) transfers the allocations of $T_i$ at the
         earliest slots to the slots in $[t_{m-1}^{th} + 2, d_i]$ closest to $d_i$; here,
         $x = t_{m-1}^{th} + 2$ and $\eta_1 = 0$                                   */
      /* in particular, for every slot $t$ from $d_i$ to $t_{m-1}^{\mathrm{th}} + 2$, it transfers the
         allocations of previous tasks at $t$ to a slot $t' \in [t_{m-1}^{\mathrm{th}} + 1, t-1]$ with
         $\overline{W}(t') > 0$ (lines 1, 5, 9 of AllocateRLM($\cdot$)); then, allocate $\overline{W}(t)$
         more machines to $T_i$ at $t$ and equivalently reduce its allocation at
         the earliest slots (line 10 of AllocateRLM($\cdot$))                       */
3     AllocateRLM($i$, 0, $t_{m-1}^{th} + 2$);
```

In terms of GreedyRLM, we have the following conclusion.

Proposition 5. *GreedyRLM gives a $\frac{s-1}{s}$-approximation to the optimal social welfare with a time complexity of $\mathcal{O}(n^2)$.*

In the following, we prove Proposition 5. There are n tasks and GreedyRLM considers tasks one by one; for an accepted task, Allocate-A(i) is called and its time complexity depends on AllocateRLM($\cdot$). Using the analysis of AllocateRLM($\cdot$) in Lemma 9, we get that its time complexity is $\mathcal{O}(n)$; so the time complexity of GreedyRLM is $\mathcal{O}(n^2)$. Next, due to Theorem 2, we only need to prove that Features 4.1.1 and 4.1.2 hold in GreedyRLM where $r = \frac{s-1}{s}$, which is given in Propositions 6 and 7 below.

The utilization of GreedyRLM is derived mainly by analyzing the resource allocation state when a task T_i cannot be fully allocated (then, the condition in line 5 of GreedyRLM is not satisfied), and we have that

Proposition 6. *Upon completion of GreedyRLM, Feature 4.1.1 holds in which $r = \frac{s-1}{s}$.*

Proof. Our analysis is retrospective and we first show that the resource utilization of $\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_m$ in $[1, \tau_m]$ is r upon completion of the m-th phase of GreedyRLM. Let us consider a task $T_i \in \cup_{l=1}^{m} \mathcal{R}_l$ such that $d_i = c_m$ and assume that $T_i \in \mathcal{R}_{m'}$ for some $m' \in [m]^+$. Since T_i is not accepted when being considered, it means that $\sum_{t \leq d_i} \min\{k_i, \overline{W}(t)\} < D_i$ at that time and there are at most $len_i - 1 = \left\lceil \frac{d_i}{s_i} \right\rceil - 1$ time slots t with $\overline{W}(t) \geq k_i$ in $[1, c_m]$. Then, we assume that the number of the time slots t with $\overline{W}(t) \geq k_i$ is μ. Since T_i isn't fully allocated, we have the current resource utilization of $\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_{m'}$ in $[1, c_m]$ is at least

$$\frac{C \cdot d_i - \mu \cdot C - (D_i - \mu \cdot k_i)}{C \cdot d_i} \geq \frac{C \cdot d_i - D_i - (len_i - 1)(C - k_i)}{C \cdot d_i}$$
$$\geq \frac{C \cdot (d_i - len_i) + (C - k_i) + (len_i \cdot k_i - D_i)}{C \cdot d_i} \geq \frac{s-1}{s} \geq r.$$

Hence, upon completion of the m'-th phase of GreedyRLM, the utilization of C machines in $[1, c_m]$ is at least r.

Now, we first show that, after $T_i \in \mathcal{R}_{m'}$ is rejected, the subsequent execution of Allocate-A(j) for each accepted task $T_j \in \mathcal{A}_l$ in the $(m'+1)$-th, $\cdots$, K-th phases of GreedyRLM doesn't change the utilization in $[1, c_m]$ where $l \in [m'+1, K]$. Fully-Utilize(j) does not change the allocation of the previously accepted tasks; in AllocateRLM(j, 0, $t_{l-1}^{th}+2$), the operations of changing the allocation of other tasks happen in its call to Routine(Δ, 0, 1, t) where we have $c_m = c_{m'} \leq t_{m'}^{th} \leq t_{l-1}^{th}$. Due to the function of lines 6-8 of Routine(Δ, 0, 1, t), we have that the subsequent execution of Allocate-A(j) will never change the allocation of $\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_{m'}$ in $[1, c_m]$. Secondly, we discuss two cases with t_m^{th} and c_m. In the first case that $t_m^{th} = c_m$, we have from the above analysis that upon completion of GreedyRLM the resource utilization of $\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_m$ in $[1, t_m^{th}]$ is $\geq r$ where $m' \leq m$. In the second case that $t_m^{th} > c_m$, upon completion of the m-th phase, the slots in $[c_m + 1, t_m^{\text{th}}]$ are fully utilized by the definition of t_m^{th} and the resource utilization in $[c_m+1, t_m^{th}]$ is 1; so, we have from the above analysis that the resource utilization of $\mathcal{A}_1, \cdots, \mathcal{A}_m$ in $[1, t_m^{th}]$ is also at least r. Here, if $m = K$, the lemma has held; otherwise, as discussed above, the execution of Allocate-A(j) in the $(m+1)$-th, $\cdots$, K-th phases will also not change the allocation state of $\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_m$ in $[1, t_m^{\text{th}}]$ upon completion of the m-th phase and the lemma also holds. □

For any $m \in [K]^+$, in the m-th phase of GreedyRLM, when a task T_i is accepted (lines 5 and 6), Allocate-A(i) is called to make it fully allocated. Now, we compare the resource utilization states before and after executing Allocate-A(i). In Allocate-A(i), Fully-Utilize(i) and AllocateRLM($\cdot$) are sequentially executed; both of them consider every time slots t from the deadline towards earlier ones.

Initial Allocation to T_i. Fully-Utilize($\cdot$) makes T_i utilize the remaining (idle) machines at t, and it does not change the allocations of the previous tasks; afterwards, the total allocation of tasks, i.e., $\overline{W}(t)$, may be increased at t due to the allocation of machines to T_i.

Transferring the Allocation of T_i. AllocateRLM(i, 0, $t_{m-1}^{th}+2$) transfers the allocation of T_i at the earliest slots to the latest slots in $[t_{m-1}^{th}+2, d_i]$, by equivalently transferrig the allocations of the previously allocated tasks at these latest slots to earlier slots, under the parallelism constraint. Here, we have that

— upon completion of Allocate-A(i), the number of allocated machines at each slot does not decrease, in contrast to the amount before its execution, i.e., before executing Fully-Utilize(i).

In particular, in AllocateRLM($\cdot$), there is a loop; for each iteration at $t \in [t_{m-1}+2, d_i]$, in the case that some conditions are satisfied (lines 2, 4, 5, 6, 8 of AllocateRLM($\cdot$) and line 10 of Routine($\cdot$)), the operations of changing the allocation of tasks happen: (a) they transfer the allocations of the previously allocated tasks from t to an earlier slot t' that is closest to t but not fully utilized (i.e., with idle machines) (lines 5, 9), and (b) afterwards let $\theta = \overline{W}(t)$, and they increase the allocation of T_i at t by θ and, correspondingly reduce the allocations of T_i (allocated by Fully-Utilize(i)) at the earliest slots in $[1, t'-1]$ by θ (line 10), ensuring that the total allocation to T_i is still D_i. For every iteration at t, if the related conditions are satisfied so that the operations of changing the allocation of tasks will be executed, we have $y_i(t) < k_i$, and $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$.

Lemma 13. *During executing AllocateRLM(i, 0, $t^{th}_{m-1}+2$), at the beginning of an iteration at t, if $y_i(t) < k_i$, and $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$, we have $\overline{W}(t) = 0$.*

Proof. We prove this by contradiction. If $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$ at the beginning of the iteration at t, we have it also holds upon completion of Fully-Utilize(i); the reason for this is that the past execution of AllocateRLM($\cdot$) transferred the allocation of T_i at the earliest slots in $[1, t'-1]$ to the latest slots in $[t+1, d_i]$ and the allocation of Fully-Utilize(i) to T_i at the earliest slots was reduced if $t < d_i$. Similarly, if $\overline{W}(t) > 0$ at the beginning of the iteration at t, it also holds upon completion of Fully-Utilize(i); the reason is that, the past execution of AllocateRLM($\cdot$) transferred the allocation of T_i at the earliest slots in $[1, t'-1]$ to the latest slots in $[t+1, d_i]$ and the allocation of all tasks at t has never been reduced by AllocateRLM($\cdot$).

So, we have that, upon completion of Fully-Utilize(i), $\overline{W}(t) > 0$, $y_i(t) < k_i$, and $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$; in the following, we show these three condtions cannot hold simultaneously. $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$ implies that after the allocation of Fully-Utilize(i) at t we have $D_i - \sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) > 0$; then, Fully-Utilize($i$) allocates $\min\left\{\overline{W}(t), k_i\right\}$ machines to T_i at t. Further, $y_i(t) < k_i$ implies that $\overline{W}(t) < k_i$ and T_i is allocated $\overline{W}(t)$ machines at the moment of deciding the allocation of T_i at t; after the allcoation at t, t will be fully utilized, which contradicts that $\overline{W}(t) > 0$ upon completion of Fully-Utilize(i). □

The threshold t^{th}_m is defined in lines 11-14 of Algorithm 5 where we have $\overline{W}(t^{\mathrm{th}}_m+1) > 0$ upon completion of the m-th phase of GreedyRLM. Since each execution of Allocate-A($\cdot$) does not reduce the amount of idle machines at every slot, we have that, for every accepted task in the first m phases of GreedyRLM, i.e., for every $T_i \in \bigcup_{l=1}^{m} \mathcal{A}_l$,

— upon each completion of Allocate-A(i), time slot t^{th}_m+1 is not fully utilized, i.e., $\overline{W}(t^{\mathrm{th}}_m+1) > 0$;

conversely, we can draw the following conclusion:

Lemma 14. *For all $m \in [K]^+$, given any accepted task T_i in the m-th phase, upon completion of each Allocate-A(i),*

— *we have for all $j \in [m, K]$ that time slot $t_j^{th} + 1$ is not fully utilized, i.e., $\overline{W}(t_j^{th} + 1) > 0$.*

During executing Allocate-A(i), AllocateRLM(i, 0, $t_{m-1}^{\text{th}} + 2$) is called in which there is a loop, and for its iteration at $t \in [t_{m-1}^{\text{th}} + 2, d_i]$, when Routine($\cdot$) is called (line 9), it will search for a slot t' earlier than but closest to t such that $\overline{W}(t') > 0$.

Lemma 15. *For all $m \in [K]^+$, let T_i denote any accepted task in the m-th phase, i.e., $T_i \in \mathcal{A}_m$. During executing AllocateRLM(i, 0, $t_{m-1}^{th} + 2$), when $t \in [t_j^{th} + 2, d_i]$ where $j \in [m, K]$, if such t' exists, we have $t' \in [t_j^{th} + 1, t - 1]$.*

Proof. We prove this by contradiction. By Lemma 14, the slot $t_j^{\text{th}}+1$ is not fully utilized upon completion of Allocate-A(i). If Lemma 15 does not hold, there exists an iteration at some t such that $t' \leq t_j^{\text{th}}$ and all slots in $[t_j^{\text{th}} + 1, t - 1]$ are fully utilized; then, in the iterations at $t, \cdots, t_j^{\text{th}} + 1$, AllocateRLM($\cdot$) will never changes the total allocation at $t_j^{\text{th}} + 1$ since t' will always be $\leq t_j^{\text{th}}$ and the current t' will becomes smaller than smaller as the sequential execution of the iterations at $t, \cdots, t_j^{\text{th}} + 1$ goes, which contradicts that $t_j^{\text{th}} + 1$ is not fully utilized upon completion of Allocate-A(i). □

Lemma 16. *At the beginning of an iteration at $t \in [t_{m-1}^{th} + 2, d_i]$, when the conditions in lines 2, 6 of AllocateRLM($\cdot$) and lines 7, 10 of Routine($\cdot$) are all false, we have that $\overline{W}(t) = 0$ and $y_i(t') = k_i$.*

Proof. AllocateRLM($\cdot$) is executed upon completion of Fully-Utilize(i) and it considers every slot t from d_i to $t_{m-1}^{\text{th}} + 2$. At t, it transfers the allocation of previous tasks to an earlier slot t' that is closest to t and not fully utilized; then, it allocates more machines to T_i at t and equivalently reduces its allocations at the earliest slots.

At t, if the conditions in lines 2, 6 of AllocateRLM($\cdot$) are false, it means that after Fully-Utilize(i) allocates machines to T_i at t, we have (a) $D_i - \sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) > 0$, and (b) $y_i(t) < k_i$; so, we have $\overline{W}(t) = 0$ by Lemma 13. At t, if the conditions in lines 7, 10 of Routine($\cdot$) are false, we have (i) $\overline{W}(t') > 0$ and (ii) $\sum_{\bar{t}=1}^{t'-1} y_i(\bar{t}) > 0$; due to (ii), the allocation of T_i at t' is still the same as the allocation achieved by Fully-Utilize(i). During executing the iterations at $d_i, \cdots, t + 1$ of AllocateRLM($\cdot$), the total allocation of previous tasks at t' is not decreased and the allocations to T_i at $1, \cdots, t'$ are not increased; hence, we have that after Fully-Utilize(i) decides the allocation at t, $\overline{W}(t') > 0$ and $D_i - \sum_{\bar{t}=t'}^{d_i} y_i(\bar{t}) > 0$; as a result, we have $y_i(t') = k_i$. □

By Lemmas 16 and 7, when the conditions in lines 2, 6 of AllocateRLM($\cdot$) and lines 7, 10 of Routine($\cdot$) are false, we have that there exists a $T_{i'}$ in line 12 of Routine($\cdot$).

Now, like Definition 1, given any $t \in [1, d]$, the maximum workload of a task T_i that could be processed in the time interval $[t, d]$ under the parallelism and deadline constraints, denoted by $\beta_{i,t}$, is defined as follows: (i) $\beta_{i,t} = 0$ if

$d_i < t$, (ii) $\beta_{i,t} = k_i \cdot (d_i - t + 1)$ if $len_i > d_i - t + 1 > 0$, and (iii) $\beta_{i,t} = D_i$ if $len_i \leq d_i - t + 1$. We say that the interval $[t, d]$ is optimally utilized by T_i if the total allocation of T_i over $[t, d]$ is $\beta_{i,t}$. In the following, to prove that Feature 4.1.2 holds in GreedyRLM, it suffices to show that upon completion of GreedyRLM the total allocation of each task $T_i \in \bigcup_{j=1}^{m} \mathcal{A}_j$ over $\left[t_m^{\text{th}} + 1, d\right]$ is $\beta_{i,m}$.

Lemma 17. *Upon completion of Allocate-A(i), we have for all $j \in [m, K]$ that $[t_j^{th} + 1, d]$ is optimally utilized by each task $T_i \in \mathcal{A}_m$.*

Proof. By Lemma 15, for each iteration of AllocateRLM(·) at $t \in [t_j^{\text{th}} + 2, d]$, there always exists a t' in line 2 of Routine(·) such that $t' \in [t_j^{\text{th}} + 1, t - 1]$. There are five cases to discuss. In the 1st case that $d_i \leq t_j^{\text{th}}$, $[t_j^{\text{th}} + 1, d]$ is optimally utilized by T_i trivially. In the 2nd case, the condition in line 2 of AllocateRLM(·) is true; then, T_i is fully allocated in $[t, d_i]$ and $[t_j^{\text{th}} + 1, d]$ is optimally utilized by T_i; afterwards, AllocateRLM(·) will stop. In the 3rd case, the condition in line 10 of Routine(·) is true; then, T_i is fully allocated in $[t', d_i] \subseteq [t_j^{\text{th}} + 1, d_i]$ and $[t_j^{\text{th}} + 1, d]$ is optimally utilized by T_i; afterwards, AllocateRLM(·) will stop.

In the 4th case, the condition in line 6 is true and nothing is done in the iteration at t; then, T_i has been allocated k_i machines at t. In the 5th case, all conditions in the 2nd, 3rd, and 4th cases are always false and the loop of Routine(·) stops when $\overline{W}(t) = \Delta$ (see line 9 of AllocateRLM(·)); afterwards, either $y_i(t) = k_i$ or $\sum_{\bar{t}=t}^{d_i} y_i(\bar{t}) = D_i$ (line 10). Hence, when every iteration of AllocateRLM(·) at $t \in [t_j^{\text{th}} + 2, d_i]$ is executed where the 4th or 5th case occurs, we have that either $y_i(t) = k_i$ for all $t \in [t_j^{\text{th}} + 2, d_i]$ or $\sum_{\bar{t}=t_j^{\text{th}}+2}^{d_i} y_i(\bar{t}) = D_i$. Here, if $y_i(t) = k_i$ for all $t \in [t_j^{\text{th}} + 2, d_i]$, we need to further consider two subcases. The first is $\sum_{\bar{t}=1}^{t_j^{\text{th}}} y_i(\bar{t}) = 0$, showing that T_i is fully allocated in $[t_j^{\text{th}} + 1, d_i]$; the second is $\sum_{\bar{t}=1}^{t_j^{\text{th}}} y_i(\bar{t}) > 0$. For every iteration of AllocateRLM(·) at $t \in [t_j^{\text{th}} + 2, d_i]$, $t' \in [t_j^{\text{th}} + 1, t - 1]$ by Lemma 15; the total allocation of T_i in $[1, t_j^{\text{th}}]$ is not increased by AllocateRLM(·), compared with the allocation achieved by Fully-Utilize(i). Hence, upon completion of Fully-Utilize(i), T_i is allocated k_i machines at $t_j^{th} + 1$; further, upon completion of AllocateRLM(·), we have $y_i(t) = k_i$ for all $t \in [t_j^{\text{th}} + 1, d_i]$. In the 2nd subcase, $[t_j^{\text{th}} + 1, d]$ is also optimally utilized by T_i. □

Lemma 18. *Upon completion of GreedyRLM, we have for all $j \in [m, K]$ that $[t_j^{th} + 1, d]$ is optimally utilized by each task $T_i \in \mathcal{A}_m$.*

Proof. By Lemma 17, upon completion of Allocate-A(i), we have for all $j \in [m, K]$ that $[t_j^{\text{th}} + 1, d]$ is optimally utilized by each task $T_i \in \mathcal{A}_m$. Now, we show that, such a conclusion still holds upon each completion of Allocate-A(i_1) where T_{i_1} denotes an arbitrary task accepted after T_i.

For all $m_1 \in [m, K]$, assume that T_{i_1} is accepted in the m_1-th phase where $T_{i_1} \in \mathcal{A}_{m_1}$. During executing Allocate-A(i_1), Fully-Utilize(i_1) does not make any change to the allocations of previously allocated tasks. AllocateRLM(i_1, 0, $t^{\text{th}}_{m_1-1}+2$) considers every slot t from d_{i_1} to $t^{\text{th}}_{m_1-1}+2$. For each iteration at t, it transfers the allocation of some previously allocated tasks (possibly including T_i) to an earlier slot t' that is closest to t and not fully utilized; then, it allocates more machines to T_{i_1} at t and equivalently reduces the allocations of T_{i_1} at the earliest slots.

Hence, for each iteration at t, the adjustment of the allocation of the previously allocated tasks (possibly including T_i) only happens at t and t' where $t' < t$ and $t \in \left[t^{\text{th}}_{m_1-1}+2, d_{i_1}\right]$; after the adjustment, the total allocation of the previously allocated tasks in $[t', t]$ does not change. Now, given any $m_1 \in [m, K]$ and $T_{i_1} \in \mathcal{A}_{m_1}$, we show after executing AllocateRLM(i_1, 0, $t^{\text{th}}_{m_1-1}+2$) that the total allocation of T_i in $[t^{\text{th}}_j+1, d]$ does not change and $[t^{\text{th}}_j+1, d]$ is still optimized by T_i, where $j \in [m, K]$.

Firstly, given any $j \in [m, L]$, in the case that $j \geq m_1$, $t^{\text{th}}_j > t^{\text{th}}_{m_1-1}$ and we have for each iteration of AllocateRLM(i_1, 0, $t^{\text{th}}_{m_1-1}+2$) at $t \in [t^{\text{th}}_j+2, d_{i_1}]$ that $t' \in [t^{\text{th}}_j+1, t-1]$ by Lemma 15; hence, the total allocation of T_i in $[t^{\text{th}}_j+1, d]$ does not change. For the iteration at $t = t^{\text{th}}_j+1$, we have at the beginning that $\overline{W}(t) > 0$; then, by Lemma 13, we either have $y_{i_1}(t) = k_i$ or $\sum_{\bar{t}=1}^{t'-1} y_{i_1}(\bar{t}) = 0$; when the condition in line 6 of AllocateRLM($\cdot$) or the condition in 10 of Routine($\cdot$) is true, the operations of changing the allocation of tasks (lines 9, 10 of AllocateRLM($\cdot$)) will not be executed and nothing is done for the iteration at t. Finally, we have upon completion of AllocateRLM(i_1, 0, $t^{\text{th}}_{m_1-1}+2$) that the total allocation of T_i in $[t^{\text{th}}_j+1, d_i] \subseteq [t^{\text{th}}_j+1, d]$ does not change. Secondly, in the case that $j = m_1 - 1$, we have $t^{\text{th}}_j = t^{\text{th}}_{m_1-1}$ and always have $t' > t^{\text{th}}_j$ if such t' exists by line 7 of Routine($\cdot$); hence, upon completion of AllocateRLM(i_1, 0, $t^{\text{th}}_{m_1-1}+2$), the total allocation of T_i in $[t^{\text{th}}_j+1, d]$ does not change. Thirdly, in the case that $j \leq m_1 - 2$, we have $t^{\text{th}}_j < t^{\text{th}}_{m_1-1}$. As analyzed in the second case, the total allocation of T_i in $[t^{\text{th}}_{m_1-1}+1, d]$ does not change upon completion of AllocateRLM(i_1, 0, $t^{\text{th}}_{m_1-1}+2$) and the allocation of T_i over $[t^{\text{th}}_j+1, t^{\text{th}}_{m_1-1}]$ will not be changed. Hence, for any $m_1 \in [m, K]$ and $T_{i_1} \in \mathcal{A}_{m_1}$, the execution of AllocateRLM(i_1, 0, $t^{\text{th}}_{m_1-1}+2$) does not affect the total allocation of T_i in $[t^{\text{th}}_j+1, d]$ and the lemma holds. □

By Lemma 18, we conversely have that given any $m \in [K]^+$, the interval $[t^{\text{th}}_m+1, d]$ is optimally utilized by each task $T_i \in \mathcal{A}_j$ for all $j \leq m$. Hence, we conclude that

Proposition 7. *Upon completion of GreedyRLM, for all $m \in [K]^+$, the interval $\left[t^{th}_m+1, d\right]$ is optimally utilized by $\bigcup_{j=1}^{m} \mathcal{A}_j$.*

4.2 Dynamic Programming

In this section, we show the application of the dynamic programming technique to the social welfare maximization problem.

By Theorem 1, a set of malleable tasks with deadlines can be feasibly scheduled on C machines if and only if the boundary condition in Lemma 1 is satisifed. For any solution to the social welfare maximizatioin problem, a subset of tasks is chosen and there must exist a feasible schedule of these chosen tasks; the set of tasks in an optimal solution also satisfies the boundary condition. Then, to find the optimal solution, we only need address the following problem: if we are given C machines, how can we choose a subset $\mathcal{S}$ of tasks in $\mathcal{T}$ such that (i) this subset satisfies the boundary condition, and (ii) no other subset of selected tasks achieves a better social welfare? This problem can be solved via dynamic programming (DP).

To propose a DP algorithm, we need to identify a dominant condition for the model of malleable tasks [21]. Let $\mathcal{F}$ be an arbitrary subset of $\mathcal{T}$, i.e., $\mathcal{F} \subseteq \mathcal{T}$, and recall that the notation $\lambda_m^C(\mathcal{F})$ in Chapter 3.1 denotes the maximum workload of $\mathcal{F}$ that could be processed in $[\tau_{L-m}+1, d]$ on C machines. Now, we define a L-dimensional vector

$$H(\mathcal{F}) = (\lambda_1^C(\mathcal{F}) - \lambda_0^C(\mathcal{F}), \cdots, \lambda_L^C(\mathcal{F}) - \lambda_{L-1}^C(\mathcal{F})),$$

where $\lambda_m^C(\mathcal{F}) - \lambda_{m-1}^C(\mathcal{F})$, $m \in [L]^+$, denotes the maximum resource that $\mathcal{F}$ can utilize on C machines in the segmented timescale $[\tau_{L-m}+1, \tau_{L-m+1}]$ after $\mathcal{F}$ has utilized $\lambda_{m-1}^C(\mathcal{F})$ resource in $[\tau_{L-m+1}+1, \tau_L]$. Let $v(\mathcal{F})$ denote the total value of the tasks in $\mathcal{F}$, i.e., $v(\mathcal{F}) = \sum_{T_i \in \mathcal{F}} v_i$.

Now, we introduce the notion of one pair *dominating* another pair: given two subsets $\mathcal{F}, \mathcal{F}' \subseteq \mathcal{T}$, we say that the pair $(\mathcal{F}, v(\mathcal{F}))$ dominates the pair $(\mathcal{F}', v(\mathcal{F}'))$ if $H(\mathcal{F}) = H(\mathcal{F}')$ and $v(\mathcal{F}) \geq v(\mathcal{F}')$; when $\mathcal{F}$ and $\mathcal{F}'$ are two feasible solution to the social welfare maximization problem, if $(\mathcal{F}, v(\mathcal{F}))$ dominates $(\mathcal{F}', v(\mathcal{F}'))$, the solution $\mathcal{F}$ uses the same amount of resource as the solution $\mathcal{F}'$, but obtains at least the same total value.

Armed with the notion of one pair dominating another pair, we give a general DP procedure DP($\mathcal{T}$), also presented as Algorithm 7 [21]. Here, we iteratively construct the lists $A(i)$ for all $i \in [n]^+$. Each $A(i)$ is a list of pairs $(\mathcal{F}, v(\mathcal{F}))$: $\mathcal{F}$ is a subset of $\{T_1, T_2, \cdots, T_i\}$ satisfying the boundary condition, and $v(\mathcal{F})$ is the total value of the tasks in $\mathcal{F}$. Each list only maintains all the dominant pairs. Specifically, we start with $A(1) = \{(\emptyset, 0), (\{T_1\}, v_1)\}$. For each $i = 2, \cdots, n$, we first set $A(i) \leftarrow A(i-1)$, and for each $(\mathcal{F}, v(\mathcal{F})) \in A(i-1)$, we add $(\mathcal{F} \cup \{T_i\}, v(\mathcal{F} \cup \{T_i\}))$ to the list $A(i)$ if $\mathcal{F} \cup \{T_i\}$ satisfies the boundary condition. We finally remove from $A(i)$ all the dominated pairs. DP($\mathcal{T}$) will select a subset $\mathcal{S}$ of $\mathcal{T}$ from all pairs $(\mathcal{F}, v(\mathcal{F})) \in A(n)$ so that $v(\mathcal{F})$ is the maximum possible.

Proposition 8. *DP($\mathcal{T}$) outputs a subset $\mathcal{S}$ of $\mathcal{T}$ such that $v(\mathcal{S})$ is the maximum value subject to the condition that $\mathcal{S}$ satisfies the boundary condition, and its time complexity is $\mathcal{O}(n \cdot d^L \cdot C^L)$.*

Proof. The proof is similar to the one in the knapsack problem [21]. By induction, we need to prove that $A(i)$ contains all the non-dominated pairs corresponding to feasible sets $\mathcal{F} \in \{T_1, \cdots, T_i\}$. When $i = 1$, the proposition

Algorithm 7: DP($\mathcal{T}$)

```
1  $\mathcal{F} \leftarrow \{T_1\}$;
2  $A(1) \leftarrow \{(\emptyset, 0), (\mathcal{F}, v(\mathcal{F}))\}$;
3  for $i \leftarrow 2$ to $n$ do
4      $A(j) \leftarrow A(i-1)$;
5      for each $(\mathcal{F}, v(\mathcal{F})) \in A(i-1)$ do
6          if $\{T_i\} \cup \mathcal{F}$ satisfies the boundary condition then
7              if there exist a pair $(\mathcal{F}', v(\mathcal{F}')) \in A(i)$ so that (1)
               $H(\mathcal{F}') = H(\mathcal{F} \cup \{T_i\})$, and (2) $v(\mathcal{F}') \geq v(\mathcal{F} \cup \{T_i\})$ then
8                  Add $(\{T_i\} \cup \mathcal{F}, v(\{T_i\} \cup \mathcal{F}))$ to $A(i)$;
9                  Remove the dominated pair $(\mathcal{F}', v(\mathcal{F}'))$ from $A(i)$;
10             else
11                 Add $(\{T_i\} \cup \mathcal{F}, v(\{T_i\} \cup \mathcal{F}))$ to $A(i)$;
12 $\mathcal{S} \leftarrow \underset{(\mathcal{F}, v(\mathcal{F})) \in A(n)}{\operatorname{argmax}} \{v(\mathcal{F})\}$;
```

holds obviously. Now suppose it hold for $A(i-1)$. Let $\mathcal{F}' \subseteq \{T_1, \cdots, T_i\}$ and $H(\mathcal{F}')$ satisfies the boundary condition. We claim that there is some pair $(\mathcal{F}, v(\mathcal{F})) \in A(i)$ such that $H(\mathcal{F}) = H(\mathcal{F}')$ and $v(\mathcal{F}) \geq v(\mathcal{F}')$. First, suppose that $T_i \notin \mathcal{F}'$. Then, the claim follows by the induction hypothesis and by the fact that we initially set $A(i)$ to $A(i-1)$ and removed dominated pairs. Now suppose that $T_i \in \mathcal{F}'$ and let $\mathcal{F}_1' = \mathcal{F}' - \{T_i\}$. By the induction hypothesis there is some $(\mathcal{F}_1, v(\mathcal{F}_1)) \in A(i-1)$ that dominates $(\mathcal{F}_1', v(\mathcal{F}_1'))$. Then, the algorithm will add the pair $(\mathcal{F}_1 \cup \{T_i\}, v(\mathcal{F}_1 \cup \{T_i\}))$ to $A(i)$. Thus, there will be some pair $(\mathcal{F}, v(\mathcal{F})) \in A(i)$ that dominates $(\mathcal{F}', v(\mathcal{F}'))$. Since the size of the space of $H(\mathcal{F})$ is no more than $(C \cdot T)^L$, the time complexity of DP($\mathcal{T}$) is $n \cdot d^L \cdot C^L$. □

Proposition 9. *Given the subset $\mathcal{S}$ output by DP($\mathcal{T}$), LDF($\mathcal{S}$) gives an optimal solution to the welfare maximization problem with a time complexity* $\mathcal{O}(\max\{n^2, n \cdot d^L \cdot C^L\})$.

Proof. It follows from Propositions 8 and 1. □

Remark. As in the knapsack problem [21], to construct the algorithm DP($\mathcal{T}$), the pairs of the possible state of resource utilization and the corresponding best social welfare have to be maintained and a L-dimensional vector has to be defined to indicate the resource utilization state. This seems to imply that we cannot make the time complexity of a DP algorithm polynomial in L.

4.3 Machine Minimization

Given a set of tasks $\mathcal{T}$, the minimal number of machines needed to produce a feasible schedule of $\mathcal{T}$ is exactly the minimum C^* such that the boundary condition is satisfied, by Theorem 1, where the feasible schedule could be produced with a time complexity $\mathcal{O}(n^2)$. An upper bound of the minimum C^* is $k \cdot n$ and this minimum C^* can be obtained through a binary search procedure with a time complexity of $\log(k \cdot n) = \mathcal{O}(\log n)$; the corresponding algorithm is presented as Algorithm 8.

Lemma 19. *In each iteration of the binary search procedure, the time complexity of determining the satisfiability of boundary condition (line 4 of Algorithm 8) is $\mathcal{O}(L \cdot n)$ where $L \leq n$.*

Proof. Recall the process of defining $\mu_m^C(\mathcal{S})$ where $\mathcal{S} = \mathcal{T}$. In Definition 2 that defines $\lambda_m(\mathcal{T})$, n tasks are considered sequentially for each $m \in [L]^+$, leading to a complexity $L \cdot n$. In Definition 3 that derives $\lambda_m^C(\mathcal{T})$ from $\lambda_m(\mathcal{T})$, $\lambda_1^C(\mathcal{T})$, $\lambda_2^C(\mathcal{T})$, $\cdots$, $\lambda_L^C(\mathcal{T})$ are considered sequentially, leading to a complexity $\mathcal{O}(L)$. Finally, $\mu_m^C(\mathcal{T}) = \sum_{T_i \in \mathcal{T}} D_i - \lambda_m^C(\mathcal{T})$. Hence, the time complexity of determining the satisfiability of boundary condition depends on Definition 2 and is $\mathcal{O}(L \cdot n)$. □

With Lemma 19, the loop of Algorithm 8 has a complexity $\mathcal{O}(L \cdot n \cdot \log n)$. Based on the above discussion, we conclude that

Proposition 10. *Algorithm 8 produces an exact algorithm for the machine minimization problem with a time complexity of $\mathcal{O}(n^2, L \cdot n \cdot \log n)$.*

Algorithm 8: Machine Minimization

```
1  $L \leftarrow 1, U \leftarrow k \cdot n$;  // $L$ and $U$ are respectively the lower and upper bounds
   of the minimum number of needed machines
2  $mid \leftarrow \frac{L+U}{2}$;
3  while $U - L \leq 1$ do
4      if the boundary condition is satisfied with $C = mid$ then
5          $U \leftarrow mid$;                                   // successful
6      else
7          $L \leftarrow mid$;                                   // failed
8      $mid \leftarrow \frac{L+U}{2}$;
9  $C^* \leftarrow U$;                           // the optimal number of machines
10 call the algorithm LDF($\mathcal{T}$) to produce a schedule of $\mathcal{T}$ on $C^*$ machines;
```

4.4 Minimizing Maximum Weighted Completion Time

When independent malleable tasks are considered with the objective of minimizing the maximum weighted completion time of tasks, a direction application of LDF($\mathcal{S}$) improves the algorithm in [8] by a factor 2. In particular, in [8], Nagarajan *et al.* first associated a unique deadline d_i with each task T_i and proposed a scheduling algorithm such that whenever there exists a feasible schedule to complete every task by its deadline, their proposed algorithm can produce a schedule of all tasks by at most two times their deadlines; then, with a polynomial time complexity, they found for each task T_i a completion time d_i that is $1+\epsilon$ times the optimal completion time. As a result, when using their algorithm to complete every task by $2 \cdot (1+\epsilon) \cdot d_i$, a $(2+2\epsilon)$-approximation algorithm is obtained. Instead, by using the optimal scheduling algorithm LDF($\mathcal{S}$) proposed in this thesis, we have that

Proposition 11. *There is a* $(1+\epsilon)$*-approximation algorithm for scheduling independent malleable tasks under the objective of minimizing the maximum weighted completion time of tasks.*

Chapter 5

Other Scheduling Problems

There is nothing so practical as a good theory.

Kurt Lewin

5.1 Introduction

5.1.1 Background and Motivations

In Chapters 2-4, the tasks are assumed to be malleable and work-preserving, i.e., the number of machines assigned to a task can be varying during the execution (which brings the operation of preempting the execution of a task), and the workload of a task does not increase with the number of machines assigned to it. In a more general case when the parallelism bound is large, parallelizing tasks and preempting their execution will inevitably introduce inefficiencies due to the extra communication overhead among machines etc. So, we consider another type of monotonic moldable tasks. For a monotonic task, its workload increases with the number of machines assigned to execute it while its execution time decreases. For a moldable task, it may be allocated any number of machines; however once specified before the task's execution, this number cannot change throughout the execution (modeling a potentially high cost of migration).

A key aspect of moldable tasks that crucially conditions their scheduling is the dependence of the execution time of a task on the number of assigned processors. A popular speedup model is to assume that tasks are *monotonic*: as more processors are assigned to a task, its execution time decreases but its total workload increases [23], [24]. This model takes well into account the overhead of communications among different parts of a parallel task that are run on different processors. It allows designing algorithms with bounded worst-case performance for objectives such as minimizing the makespan (i.e., the maximum completion time of all tasks).

To date, the most efficient result for makespan minimization is a $(\frac{3}{2} + \epsilon)$-approximation algorithm with a time complexity of $\mathcal{O}(mn \log \frac{n}{\epsilon})$ [24], where n is the number of tasks and m is the number of processors. Here, for a minimization problem, a ρ-approximation algorithm is such that its performance (e.g., makespan) is always $\leq \rho$ times the performance of an optimal algorithm where $\rho \geq 1$. In practice, it is always desired to have performance bounds closer to one, while keeping algorithms simple to run efficiently. As shown in our subsequent analysis (see our ideas in Chapter 5.2.1), the monotonicity of tasks does not hinder us to achieve better algorithms when every task is executed on a large number of processors; however, it indeed do when there exist tasks executed on a small number of processors. In the latter case, a precise speedup description could help.

Fortunately, many typical benchmarks have been studied so far; here, each benchmark represents a type of computations whose instances form the tasks to be executed in this chapter, and the tasks of the same type of computations have the same speedup behaviour (with the same δ_j and k_j below). Let $D_{j,p}$ denote the workload to be processed when a task T_j is assigned p processors, and the execution time $t_{j,p}$ of this task is $D_{j,p}/p$. The following **speedup mode** was observed in [25]. When a small number of processors (up to a threshold δ_j) is assigned to T_j, the speedup is linear, i.e., the workload remains constant and the execution time decreases linearly as p ranges from 1 to δ_j. Then when the task is assigned $> \delta_j$ processors, the speedup declines, i.e., the execution time still decreases as p increases but the workload begins to increase (as for monotonic tasks). Finally, there is a larger threshold k_j such that when $p > k_j$, the workload of T_j begins to increase to a greater extent and its execution time does not decrease any more as p increases.

The study [25] considered typical computations that arise in a wide range of applications: Conjugate Gradient (CG), Fourier Transform (FT), Integer Sort (IS), Block Tridiagonal (BT), Embarrassingly Parallel [26], Multi-Grid [27], High Performance Linpack (HPL). For example, the CG and BT benchmarks solve a system of linear equations whose matrix are symmetric and positive-definite, and block-tridiagonal respectively. The FT benchmark is a 3-D partial differential equation solver. The HPL benchmark involves floating point computations and has been used to classify the top 500 fastest computers in the world. In these benchmarks, the parameter δ_j ranges from 25 to 150 and k_j ranges from 75 to 300. Besides, parallel implementations of many other computations are also consistent with the observations in [25]. For example, in cryptography, central to the lattice-based cryptosystems is a shortest vector problem (SVP) and an implementation of the SVP-solver [28] shows that δ_j could be set to 64. In computer vision, the scale-invariant feature transform (SIFT) algorithm is used to detect and describe local features in images where $\delta_j = 8$ [29]. Its applications include object recognition, robotic mapping and navigation, 3D modeling, etc. More examples can be found in [30]–[32].

Table 5.1 – The value of $\mu(\delta)$ when the value of δ is in different ranges.

δ	$\mu(\delta)$	δ	$\mu(\delta)$	δ	$\mu(\delta)$
[5, 9]	0.7500	[22, 26]	0.8571	58	0.9000
[10, 16]	0.8000	[27, 37]	0.8750	[59, 74]	0.9091
[17, 21]	0.8333	[38, 57]	0.8889	[75, 101]	0.9167

As a result, given a set of tasks that might execute one or multiple types of computations, we let δ and k respectively denote the maximum and minimum integers such that $\delta \leq \delta_j$ and $k \geq k_j$ for every task T_j; as shown in the above study, $\delta \geq 8$ and k could be set to a value ≤ 300. Finally, by summarizing the speedup mode observed in [25], we introduce in this thesis **the notion of (δ, k)-monotonic tasks** [33] under which tasks are moldable and for every task T_j that is assigned p processors, we have that

(i) when p ranges in $[1, \delta]$, its workload $D_{j,p}$ remains constant and the speedup is linear;

(ii) when p ranges in $[\delta + 1, k]$, the workload is non-decreasing in p while its execution time first decreases and then even begins to increase once p exceeds some threshold (i.e., k_j);

(iii) the parallelism bound (i.e., the maximum number of processors that is allowed to assigned to a task) is k.

We show that this speedup model allows designing simpler yet more efficient scheduling algorithms than the monotonic model.

Here, we note that, in [25], the speedup mode above is also approximated and expressed by a unified function $t_{j,p} = D_{j,1}/p + (p-1) \cdot c$, where $D_{j,p} = p \cdot t_{j,p}$, and c is a very small positive real number [1]. Under this function, related online scheduling problems have been studied [34]–[37]. Previously, the case that the parallelism bound is δ_j where the speedup is linear has been studied when offline batch scheduling is considered by [3]; the case that all tasks have a common parallelism bound $\delta = \min_{T_j \in \mathcal{T}}\{\delta_j\}$ has been studied when online scheduling is considered by [3]; differently, their tasks are malleable, i.e., the number of assigned processors is allowed to vary during the execution of a task, rather than moldable.

It should be pointed out that m is large since our problem arises in large-scale parallel systems such as modern clusters and massively parallel processing systems; for example, earlier supercomputers such as IBM BlueGene/L have $m = 2^{16}$ processors inside [38], [39], and modern clusters contain even more processors [3], [40].

1. When the number p of assigned processors is also small, the effect of the term $(p-1) \cdot c$ on the task's execution time t is negligible.

5.1.2 Algorithmic Results

Specifically, we consider the problem of scheduling a set of n moldable tasks on m processors under the proposed notion of (δ, k)-monotonic tasks, and a main algorithmic result for scheduling this type of tasks is

— a $\frac{1}{\theta(\delta)} \cdot (1+\epsilon)$-approximation algorithm with a time complexity $\mathcal{O}(n \log \frac{n}{\epsilon})$ to minimize the makespan. In particular, $\theta(\delta)$ is of the following form:

$$\theta(\delta) = \mu(\delta) - \mathcal{O}(\tfrac{1}{m}).$$

Here, the value of $\mu(\delta)$ under a particular δ is a constant, illustrated in Table 5.1; m is large in large-scale parallel systems such that $\mathcal{O}(\frac{1}{m}) \to 0$, and $\theta(\delta) \to \mu(\delta)$.

Every task is an instance of some type of computations (e.g., a procedure of solving linear equations); as discussed above, we have in many typical computations [25], [28]–[32] that $\delta \geq 8$ and then the approximation ratio of our algorithm approaches $\frac{4}{3}$. When the benchmarks/computations of [25] are considered alone, $\delta \geq 25$ and the ratio approaches $\frac{7}{6}$. When the computations in a lattice-based cryptosystem are executed [28], $\delta = 64$ and the ratio approaches $\frac{11}{10}$. As a result, we have proposed better approximation algorithms than the best $(\frac{3}{2} + \epsilon)$-approximation algorithm under the monotonic assumption, by taking advantage of a new description of the relation between the speedup and the number of processors assigned to a task.

Besides, as a by-product in the process of deriving the algorithm above, *another result* that we obtain is a $\theta(\delta)$-approximation algorithm with a time complexity $\mathcal{O}(n^2)$ to maximize the social welfare (i.e., the sum of values of tasks completed by a deadline)—see the definition of approximation algorithms in Chapter 5.3. To the best of our knowledge, we are the first to address the social welfare maximization objective for moldable tasks, while this objective has been addressed for other types of tasks [3], [41]–[43].

5.2 Related Work

We introduce the related works following which our high-level ideas for scheduling (δ, k)-monotonic tasks are also described.

5.2.1 Makespan Minimization

We discuss some typical works on scheduling moldable tasks for makespan minimization [23], [24], [44]–[55]. The problem is strongly NP-hard when $m \geq 5$ [56]. The first two lines of works are based on a two-phases approach; the first phase selects the number of processors assigned to every task and the second phase goes to solve the resulting non-moldable scheduling problem.

First Line of Works. Of the great relevance to our work is the first line of works where monotonic tasks are considered [23], [24]; here, the following definition is given:

Definition 5. *Given a positive real number d, we define for every task T_j its canonical number of processors $\gamma(j,d)$ as the minimum number of processors needed to execute task T_j by time d; here, for every task $T_j \in \mathcal{T}$, $1 \leq \gamma(j,d) < +\infty$.*

A monotonic task is such that its execution time decreases but its workload increases with the number of used processors. We denote by $D_{j,p}$ and $t_{j,p}$ the workload of T_j and its execution time when assigned p processors where $D_{j,p} = p \cdot t_{j,p}$. By the definition of $\gamma(j,d)$ and the relation that $D_{j,\gamma(j,d)} > D_{j,\gamma(j,d)-1} = (\gamma(j,d)-1) \cdot t_{j,\gamma(j,d)-1} > (\gamma(j,d)-1) \cdot d$, we conclude that

$$d \geq t_{j,\gamma(j,d)} > \frac{\gamma(j,d)-1}{\gamma(j,d)} \cdot d. \tag{5.1}$$

In this type of works, the key is classifying tasks by solving a knapsack problem and so far the best result is presented in [24]. In particular, the algorithm of [24] classifies the tasks into two subsets $\mathcal{T}_1$ and $\mathcal{T}_2$, by solving a knapsack problem via dynamic programming with a complexity of $\mathcal{O}(m)$, where the tasks respectively in $\mathcal{T}_1$ and $\mathcal{T}_2$ are allocated $\gamma(j,d)$ and $\gamma(j,\frac{d}{2})$ processors, and then the total workload of the tasks is $\leq m \cdot d$. Then, the total number of processors allocated to $\mathcal{T}_2$ may be $> m$ and a series of reductions to the numbers of processors assigned to tasks is taken to get a feasible schedule with a makespan $\leq \frac{3}{2} \cdot d$. Finally, Mounié *et al.* obtained a $\frac{3}{2} \cdot (1+\epsilon)$-approximation algorithm with a complexity of $\mathcal{O}(mn \log \frac{n}{\epsilon})$ in [24].

Our Ideas for Makespan Minimization. Similar to the work of [24], we also determine the number of processors assigned to every task in advance and it originates from the following preliminary observation for monotonic tasks. Tasks are available at time 0. Suppose there exists a schedule of all tasks, denoted by *Sched*, whose makespan is d and that achieves a resource utilization r in $[0,d]$; under the condition that the minimum workload of every task T_j is processed (i.e., assigned $\gamma(j,d)$ processors), the schedule *Sched* will be a $\frac{1}{r}$-approximation algorithm for makespan minimization. The reason is as follows. Denote by d^* the makespan of an optimal schedule denoted by *Sched**, where $d^* \leq d$; thus, the workload of T_j when assigned $\gamma(j,d^*)$ processors $D_{j,\gamma(j,d^*)} \geq D_{j,\gamma(j,d)}$. In the schedule *Sched**, the workload of every task is $\geq D_{j,\gamma(j,d^*)}$ since the number of processors assigned to a task T_j is at least $\gamma(j,d^*)$; thus the total workload of all tasks is $\leq m \cdot d^*$ but $\geq$ its counterpart in *Sched* that is $\geq r \cdot m \cdot d$. So, we derive that the optimal makespan d^* is $\geq \frac{r \cdot m \cdot d}{m} = r \cdot d$, that is, $\frac{d}{d^*} \leq \frac{1}{r}$.

Now, only if we could design such a schedule *Sched* with a utilization $r > 2/3$, an algorithm better than the one in [24] could be obtained. Our first problem is to give the schedule *Sched*; to achieve this, a challenge arises from the existence of tasks with small $\gamma(j,d)$. In particular, given an integer $H \geq 4$, we call the tasks with $\gamma(j,d) \geq H$ as the tasks with large $\gamma(j,d)$. Every task with large $\gamma(j,d)$ has an execution time $> \frac{H-1}{H} \cdot d$ by Inequality (5.1) when assigned $\gamma(j,d)$ processors; these processors could achieve a utilization $\geq \frac{H-1}{H} \geq \frac{3}{4}$ in $[0,d]$. To cope with the tasks with small $\gamma(j,d) \leq H-1$, we introduce the notion of

(δ, k)-monotonic tasks where $H-1 \leq \delta$, enabling a generic classification of these tasks (see Chapter 5.4.1); here, every task will be assigned the same number of processors ($\leq \delta$) and its minimum workload is processed. We can thus propose such a schedule *Sched* whose utilization r approximates $\frac{H-1}{H}$ (see Chapter 5.4.2 and 5.4.3).

As seen later, our second problem is that the utilization of *Sched* can be derived only when m processors are not enough to process all tasks by time d where some tasks are rejected; however, what we need is the utilization when all tasks are scheduled. To address this, let U and L be such that *Sched* can produce a feasible schedule of all tasks by time U but fails to do so by time L, and we apply a binary search procedure to *Sched*. After the procedure ends, we have for such U and L that $U \leq L \cdot (1+\epsilon)$ and r denotes the utilization of *Sched* when $d = U$. After an extended analysis of our preliminary observation, we could derive that the final schedule of all tasks by time U is a $\frac{1}{r} \cdot (1+\epsilon)$-approximation algorithm (see Chapter 5.5.1).

Second and Third Lines of Works. As for the second line of works, Turek *et al.* and Ludwig *et al.* showed that any λ-approximation algorithm of a time complexity $\mathcal{O}(f(m,n))$ for the problem of scheduling rigid tasks can be adapted into a λ-approximation algorithm of a complexity $\mathcal{O}\left(n \log^2 m + f(m,n)\right)$ for the moldable scheduling problem [57]–[59]; here, a rigid task requires to be executed on a fixed number of processors; then, the strip packing algorithm in [60] could be applied to obtain a polynomial time 2-approximation algorithm. The third line of work [61] formulates the original problem as a linear programming problem and propose an $(1+\epsilon)$-approximation algorithm with a time complexity of $\mathcal{O}(n)$ given a fixed number of processors; here, independent of ϵ, the actual complexity is also exponential in the number of processors.

5.2.2 A Unified Speedup Function

As stated in Chapter 5.1.1, the speedup model for (δ, k)-monotonic tasks is also approximated by $t_{j,p} = D_{j,1}/p + (p-1) \cdot c$ [25], under which related scheduling problems have been studied for makespan minimization where tasks arrive over time. In particular, Havill and Mao studied an algorithm that assigns p_j processors to a task T_j such that its execution time t_{j,p_j} is minimized [34] and showed that it achieves an approximation ratio $4 - \frac{4}{m}$ and $4 - \frac{4}{m+1}$ respectively when m is even and odd. Subsequently, some special cases are also studied where the number of processors m is ≤ 4. For example, assume that the arrival time of a task T_j is a_j and Dutton and Mao studied an algorithm that assigns p_j processors such that its completion time $a_j + t_{j,p}$ is minimized subject to the number of processors idle at a_j; they showed that it has an approximation ratio 2, 9/4, and 20/9 respectively for $m = 2$, 3, and 4 [35], [36]. Furthermore, a more general speedup model was studied in [37] where c is a task-dependent value (i.e., c_j); here, Guo and Kang showed any online algorithm has an approximation ratio $\geq \phi = \frac{1+\sqrt{5}}{2}$ and in the special case where $m = 2$ gave an algorithm with an approximation ratio ϕ.

5.2.3 Social Welfare Maximization

Several works have studied the problems of scheduling other types of parallel tasks to maximize the sum of values of tasks completed by a common deadline [41], [43] or individual deadlines [3], [4].

In [41], [43], Jansen *et al.* considered rigid tasks each of which requires to be executed on a fixed number of processors; for example, in [41], they applied the theory of knapsack problem and linear programming to propose an $(\frac{1}{2}+\epsilon)$-approximation algorithm. Here, different types of tasks have different features and, given a set of tasks, the type (e.g., rigid or moldable) will determine the way that we can design a schedule (that will specify for each chosen task a time interval within which it is executed and the number of assigned processors). For (δ, k)-monotonic tasks, the moldablity of tasks requires us to determine how many processors to be assigned to every task, and the involved monotonicity also implies that the value obtained from processing a unit of its workload might vary with the number of assigned processors. These are different from the cases with rigid tasks.

Recently, [3] and [4] studied malleable tasks whose speedup is linear within a parallelism bound, as introduced in Chapter 5.1.1. Jain *et al.* proposed a $(\frac{C-\delta_{max}}{C}\cdot\frac{s-1}{s})$-approximation algorithm for the offline scheduling problem [3] and afterwards proposed a $\frac{1}{g(s)}$-approximation algorithm for the online scheduling problem [2] [3], where $g(s) = \left(2+\mathcal{O}(\frac{1}{(\sqrt[3]{s}-1)^2})\right)$. Here, δ_{max} is the maximum parallelism bound of all tasks, i.e., $\delta_{max} = \max_{T_j\in\mathcal{T}}\{\delta_j\}$; every task T_j has the minimum execution time when the number of assigned processors is its parallelism bound δ_j, and the parameter s has a value $\leq$ the ratio of every task's deadline minus arrival time to its minimum execution time. The motivation of designing such algorithms is that many applications are delay-tolerant such that s could be much greater than 1.

Our Ideas for Social Welfare Maximization. In this thesis, our ideas are as follows. We first give a generic (greedy) algorithm that will define the order in which tasks are accepted for scheduling; here, the final algorithm will only accept a part of tasks due to the capacity constraint. Then we retrospectively analyze this algorithm and define what parameters will determine its performance. As a result, since the minimum workload of every accepted task is processed in our scheduling procedure in Chapter 5.4, a direct application of this procedure to that greedy algorithm leads to an algorithm whose approximation ratio is its utilization.

5.3 Problem Description

There is a set of n independent moldable tasks $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$, which is considered to be scheduled on m identical processors. When a task T_j is assigned p processors, we denote by $D_{j,p}$ its workload to be processed and

[2] See the conclusion part of [3] for this updated approximation ratio.

by $t_{j,p}$ its execution time where $D_{j,p} = t_{j,p} \cdot p$. All tasks of $\mathcal{T}$ are available at the starting time 0. Before executing a moldable task, the scheduler has the opportunity to determine the number of processors assigned to each task; however, once executed, the task could not be terminated until its completion and this number of assigned processors cannot be changed during execution. We consider a class of moldable tasks defined as follows.

Definition 6 ((δ, k)-monotonic tasks)**.** *A (δ, k)-monotonic task T_j is a moldable task that satisfies:*

1. *its workload remains constant when p ranges in $[1, \delta]$, i.e., $D_{j,1} = D_{j,p}$;*
2. *its workload is non-decreasing in p when p ranges in $[\delta, k]$, i.e., $D_{j,p} \leq D_{j,p+1}$ if $\delta \leq p \leq k-1$;*
3. *the maximum number of processors assigned to T_j is $\leq k$;*

here, its execution time $t_{j,p}$ is $D_{j,p}/p$ for any p.

By the definition of (δ, k)-monotonic tasks, we have that

Lemma 20. *The workload of T_j is non-decreasing in p when p ranges in $[1, k]$, that is, $D_{j,p} \leq D_{j,p+1}$ for $p \in [1, k-1]$.*

As is given in Definition 5 for monotonic tasks, we still use $\gamma(j, d)$ to denote the minimum integer such that $t_{j,\gamma(j,d)} \leq d$.

Lemma 21. *Inequality (5.1) holds under Definition 6.*

Proof. By the definition of $\gamma(j, d)$, we have $t_{j,\gamma(j,d)} \leq d$ and $t_{j,\gamma(j,d)-1} > d$. By Lemma 20, $D_{j,\gamma(j,d)} \geq D_{j,\gamma(j,d)-1}$ and we have $t_{j,\gamma(j,d)} \cdot \gamma(j, d) \geq t_{j,\gamma(j,d)-1} \cdot (\gamma(j, d) - 1) > (\gamma(j, d) - 1) \cdot d$. Hence, Inequality (5.1) holds. □

We consider two scheduling objectives separately. The first objective is minimizing the makespan, i.e., the maximum completion time of all tasks $\mathcal{T}$. In addition, every task in $\mathcal{T}$ may be associated with a value and the second objective is maximizing the social welfare, i.e., the sum of values of tasks completed by a deadline τ, where partial execution of a task by the deadline yields no value. The problem with the second objective involves selecting a subset of $\mathcal{T}$ where not all tasks could be completed by time d due to the constraint of available machines. Our goal for each objective is to propose appropriate schedules/algorithms to specify for each processed task a time interval within which it is executed and the number of processors assigned to it.

As introduced in Chapter 2.4, the performance of an algorithm is indicated by the makespan or social welfare that it achieves; the algorithm's quality could be measured by a performance's ratio, i.e., the ratio of the proposed algorithm's performance to the performance of an ideally optimal algorithm (that is unknown to us), which is referred to as approximation ratio [21]. Formally, we denote by $A(\mathcal{T})$ and $OPT(\mathcal{T})$ respectively the performance of an algorithm and the optimal one, and an algorithm is called a ρ-approximation algorithm if there exists a value ρ such that, for an arbitrary set $\mathcal{T}$, (i) when a minimization problem is considered,

$$\frac{A(\mathcal{T})}{OPT(\mathcal{T})} \leq \rho, \ \ \rho \geq 1,$$

and (ii) when a maximization problem is considered,

$$\frac{A(\mathcal{T})}{OPT(\mathcal{T})} \geq \rho, \ \ 0 \leq \rho \leq 1.$$

In particular, given an arbitrary $\mathcal{T}$, a ρ_1-approximation algorithm ($\rho_1 \geq 1$) for the makespan minimization problem always returns a solution whose makespan is at most ρ_1 times the optimal makespan, while a ρ_2-approximation algorithm ($\rho_2 \leq 1$) for the social welfare maximization problem returns a solution whose social welfare is at least ρ_2 times the optimal one. In this thesis, our final goal is to propose low-complexity algorithms that achieve approximation ratios as close to 1 as possible for both problems.

5.4 Scheduling for Utilization

In this section, we consider the situation where the number of tasks is so large that only a subset of $\mathcal{T}$ could be completed by a deadline d on m processors due to the capacity constraint; we will manage to select an appropriate subset of tasks such that a simple schedule of them could achieve a high utilization of processors.

5.4.1 Parameter Identification and Task Classification

In this subsection, we will classify all tasks of $\mathcal{T}$ according to their execution time when assigned $\gamma(j,d)$ processors or another number of processors (i.e., δ' processors). The classification enables understanding the way of assigning tasks onto processors efficiently, and our final aim in this section is to propose a schedule whose utilization approaches $r = \frac{H-1}{H}$ where H is an integer in $[4, \delta+1]$. As seen later, all tasks of $\mathcal{T}$ will be classified as follows: (i) the tasks with $t_{j,\gamma(j,d)} \geq r \cdot d$, (ii) the tasks with $t_{j,\delta'} < (1-r) \cdot d$, and (iii) the tasks with $t_{j,\gamma(j,d)} < r \cdot d$ and $t_{j,\delta'} \geq (1-r) \cdot d$. Now, we start to elaborate the classification, as well as the meaning of related parameters.

First Class of Jobs. Every task $T_j \in \mathcal{T}$ is associated with an integer $\gamma(j,d)$, i.e., the minimum number of processors assigned to T_j such that it can be completed by time d. By Inequality (5.1), every task $T_j \in \mathcal{T}$ with $\gamma(j,d) \geq H$ has an execution time $\geq r \cdot d$ when assigned $\gamma(j,d)$ processors. For every $h \in [1, H-1]$, we denote by

$$\mathcal{A}'_h = \{T_j \in \mathcal{T} \mid \gamma(j,d) = h, t_{j,h} \geq r \cdot d\}$$

the set of tasks satisfying (i) $\gamma(j,d) = h$ and (ii) their execution time is $\geq r \cdot d$ when assigned h processors.

The first class of tasks is denoted by $\boldsymbol{\mathcal{A}'}$ and defined as follows:

$$\boldsymbol{\mathcal{A}'} = \{T_j \in \mathcal{T} \mid \gamma(j,d) \geq H\} \cup \mathcal{A}'_1 \cup \cdots \cup \mathcal{A}'_{H-1},$$

containing all the tasks whose execution time is $\geq r \cdot d$ when assigned $\gamma(j, d)$ processors. We thus conclude that

Proposition 12. *Each task in $\mathcal{A}'$ has an execution time $\geq r \cdot d$ when assigned $\gamma(j, d)$ processors.*

Second Class of Jobs. We proceed to classify the remaining tasks in $\mathcal{T} - \mathcal{A}'$. Recall the definition of $\mathcal{A}'$ and we have

$$\mathcal{T} - \mathcal{A}' = \{T_j \in \mathcal{T} \mid \gamma(j, d) \in [1, H-1], t_{j,\gamma(j,d)} < r \cdot d\}.$$

By Definition 6, we have the knowledge of speedup when a task is assigned $\leq \delta$ processors, i.e., it is linear; to have such knowledge, we assume in the following that $H - 1 \leq \delta$.

Firstly, we consider the tasks of $\mathcal{T} - \mathcal{A}'$ with small $\gamma(j, d)$, i.e., $\gamma(j, d)$ smaller than some value ν where $\nu \leq H - 1$; in this case, assigning a properly large number of processors (i.e., δ' processors) to such a task can greatly reduce its execution time to an extent we expect (i.e., $< (1-r) \cdot d$) where $\nu - 1 < \delta' \leq \delta$. As will be seen in Observation 5.4.1, this can bring some desired feature when designing a schedule, e.g., sequentially executing as many such tasks as possible on δ' processors in $[0, d]$ can lead to a utilization $\geq r$.

In particular, let δ' be an integer in $[\nu, \delta]$ and for every $T_j \in \mathcal{T} - \mathcal{A}'$ with $\gamma(j, d) \leq \nu - 1$ we have by Definition 6 that $t_{j,\gamma(j,d)} \cdot \gamma(j, d) = t_{j,\delta'} \cdot \delta'$. Hence, when T_j is assigned δ' processors, its execution time $t_{j,\delta'}$ is $< \frac{\gamma(j,d)}{\delta'} \cdot r \cdot d \leq \frac{\nu-1}{\delta'} \cdot r \cdot d$. We thus go to seek two integers ν and δ' such that

$$\nu \leq H - 1 \leq \delta, \text{ and } \nu \leq \delta' \leq \delta, \tag{5.2a}$$

$$\frac{\nu}{\delta'} \cdot r \geq 1 - r, \tag{5.2b}$$

$$\frac{\nu - 1}{\delta'} \cdot r < 1 - r. \tag{5.2c}$$

The existence of such parameters ν and δ' will be shown subsequently. Here, when the parameter ν satisfies Inequality (5.2c), we can guarantee the relation $t_{j,\delta'} < (1-r) \cdot d$ for every $T_j \in \mathcal{T} - \mathcal{A}'$ with $\gamma(j, d) \leq \nu - 1$, which will be formalized in Lemma 22; (5.2b) indicates that ν is the maximum such integer that can guarantee this relation. Formally, for every $h \in [1, \nu - 1]$, we let

$$\mathcal{A}''_h = \{T_j \in \mathcal{T} - \mathcal{A}' \mid \gamma(j, d) = h\} = \left\{T_j \in \mathcal{T} \mid \gamma(j, d) = h, t_{j,\gamma(j,d)} < r \cdot d\right\},$$

and have that

Lemma 22. *Every task $T_j \in \mathcal{A}''_1 \cup \cdots \cup \mathcal{A}''_{\nu-1}$ has an execution time $< (1-r) \cdot d$ when assigned δ' processors.*

Next, the rest are the tasks of $\mathcal{T} - \mathcal{A}' - \mathcal{A}''_1 \cup \cdots \cup \mathcal{A}''_{\nu-1}$, i.e., the tasks of $\mathcal{T} - \mathcal{A}'$ with $\gamma(j, d) \in [\nu, H-1]$. For every $h \in [\nu, H - 1]$, all tasks $T_j \in \mathcal{T} - \mathcal{A}'$ with $\gamma(j, d) = h$ are classified as follows:

— $\boldsymbol{\mathcal{A}_h} = \{T_j \in \mathcal{T} - \mathcal{A}' | \gamma(j,d) = h, t_{j,\delta'} \geq (1-r) \cdot d\}$;

— $\mathcal{A}_h''' = \{T_j \in \mathcal{T} - \mathcal{A}' | \gamma(j,d) = h, t_{j,\delta'} < (1-r) \cdot d\}$.

The second class of tasks is denoted by $\boldsymbol{\mathcal{A}''}$ and defined as follows:

$$\boldsymbol{\mathcal{A}''} = \text{the union of } \bigcup_{h=1}^{\nu-1} \mathcal{A}_h'' \text{ and } \bigcup_{h=\nu}^{H-1} \mathcal{A}_h''';$$

it contains all tasks of $\mathcal{T} - \mathcal{A}'$ with $\gamma(j,d) \leq \nu - 1$ and all tasks of $\mathcal{T} - \mathcal{A}'$ with $\nu \leq \gamma(j,d) \leq H-1$ and $t_{j,\delta'} < (1-r) \cdot d$. Due to Lemma 22 and the definition of $\mathcal{A}_h'''$, we conclude that

Proposition 13. *Each task in $\mathcal{A}''$ has an execution time $< (1-r) \cdot d$ when assigned δ' processors.*

As described below, all tasks of $\mathcal{A}''$ have some desired properties when designing a schedule.

***Observation* 5.4.1.** *Assume that there are δ' processors on which a set of tasks $\mathcal{T}'$ has already been sequentially executed in the time interval $[0,d]$. Given a task T_j whose execution time is $< (1-r) \cdot d$ when assigned δ' processors, if $\mathcal{T}' \cup \{T_j\}$ cannot be sequentially completed by time d on these δ' processors, the sequential execution of the tasks in $\mathcal{T}'$ leads to that the utilization of these δ' processors in $[0,d]$ is $\geq r$.*

Proof. We prove this by contradiction. Suppose that the utilization of the δ' processors in $[0,d]$ is $< r$; the sequential execution time of the tasks in $\mathcal{T}'$ on the δ' processors is $< r \cdot d$. As a result, T_j can still be completed on these processors by time d after sequentially executing the tasks of $\mathcal{T}'$, since T_j has an execution time $\leq (1-r) \cdot d$ on these processors. □

Third Class of Jobs. So far, all tasks with $t_{j,\gamma(j,d)} \geq r \cdot d$ are denoted by $\mathcal{A}'$ while all tasks with $t_{j,\delta'} < (1-r) \cdot d$ are denoted by $\mathcal{A}''$. Now, only the tasks of $\mathcal{T} - \mathcal{A}' - \mathcal{A}''$ are left, i.e., the tasks of $\mathcal{A}_\nu \cup \cdots \cup \mathcal{A}_{H-1}$ defined above; the property of these tasks in terms of their execution time enables the following observation: for every $h \in [\nu, H-1]$, after sequentially executing several tasks of $\mathcal{A}_h$ on δ' processors, the total execution time of these tasks is in $[r \cdot d, d]$. We thus seek an integer x_h for every $h \in [\nu, H-1]$ such that

$$H - 1 \leq \delta', \tag{5.3a}$$

$$\frac{h}{\delta'} \cdot r \cdot x_h \leq 1, \tag{5.3b}$$

$$\max\{1-r, \frac{h-1}{\delta'}\} \cdot x_h \geq r. \tag{5.3c}$$

Here, since $\gamma(j,d)$ is the minimum number of processors needed to complete T_j by time d and we are considering the tasks with $\gamma(j,d) \leq H-1$, (5.3a) is required to ensure that every task can be completed by d. With Inequalities (5.3b)-(5.3c), we come to the following conclusion:

Proposition 14. *When assigned δ' processors,*

(i) a task of $\mathcal{A}_h$ has an execution time $< \frac{h}{\delta'} \cdot r \cdot d$, and

(ii) any x_h tasks of $\mathcal{A}_h$ have a total execution time in $[r \cdot d, d]$,

where $h \in [\nu, H-1]$.

Proof. When assigned $\gamma(j,d) = h$ processors, a task $T_j \in \mathcal{A}_h$ has an execution time $< r \cdot d$ but $> \frac{h-1}{h} \cdot d$ by Inequality (5.1) where $h \leq H-1 \leq \delta$; by Definition 6, when assigned δ' processors where $\delta' \leq \delta$, the execution time of T_j is linearly reduced by a factor $\frac{h}{\delta'}$ and is in $\left(\frac{h-1}{\delta'} \cdot d, \frac{h}{\delta'} \cdot r \cdot d\right)$. So, the proposition's first point holds. Besides, by the definition of $\mathcal{A}_h$, when assigned δ' processors, the execution time of $T_j \in \mathcal{A}_h$ is also $\geq (1-r) \cdot d$ and is in $\left[d \cdot \max\{1-r, \frac{h-1}{\delta'}\}, \frac{h}{\delta'} \cdot r \cdot d\right]$. Further, due to Inequalities (5.3b)-(5.3c), any x_h tasks have a total of execution time in $[r \cdot d, d]$ and the proposition's second point holds. □

***Observation* 5.4.2.** *When assigning one or multiple tasks of $\mathcal{A}_h$ onto δ' processors where $h \in [\nu, H-1]$, we have the following observation:*

1. *After sequentially executing as many tasks of $\mathcal{A}_h$ on δ' processors until the next task of $\mathcal{A}_h$ cannot be completed by time d, the δ' processors achieve a utilization $\geq r$ in $[0, d]$.*
2. *If some other tasks (each of which can have an arbitrary execution time in $[0, d]$) have been sequentially executed on δ' processors by time d but the next task that is from $\mathcal{A}_h$ cannot be completed on these processors by d, these processors achieve a utilization $\geq 1 - \frac{h}{\delta'} \cdot r$ in $[0, d]$.*

Proof. By the second point of Proposition 14, at least x_h tasks of $\mathcal{A}_j$ will be executed in the first point of Observation 5.4.2 and this point thus holds. In the second point of Observation 5.4.2, in the process of assigning tasks to processors, there exists a task such that (a) its execution time is $< \frac{h}{\delta'} \cdot r \cdot d$ when assigned δ' processors by Proposition 14, and (b) when considering assigning it to the δ' processors, it could not be completed by d. The existence of such a task means that, all the previous executed tasks on these processors have a total execution time $\geq d - \frac{h}{\delta'} \cdot r \cdot d = (1 - \frac{h}{\delta'} \cdot r) \cdot d$; otherwise, the next task from $\mathcal{A}_h$ could be completed by d. The second point thus holds. □

Final Task Classification. To sum up, in this subsection, all tasks of $\mathcal{T}$ are finally partitioned as follows:

$$\mathcal{A}'', \mathcal{A}_{H-1}, \cdots, \mathcal{A}_v, \mathcal{A}', \tag{5.4}$$

which are illustrated in Figure 5.1. The desired properties of these tasks have been presented in Propositions 12, 13, and 14; their functions in the design of a schedule are also clarified in Observations 5.4.1 and 5.4.2.

Existence of Parameters. Now, we need to show the existence of parameters H, ν, δ' and x_h in Inequalities (5.2a), (5.2b), (5.2c), (5.3a), (5.3b), and (5.3c). Here, (5.2a) and (5.3a) are unified as follows:

$$1 \leq \nu \leq H-1 \leq \delta' \leq \delta. \tag{5.5}$$

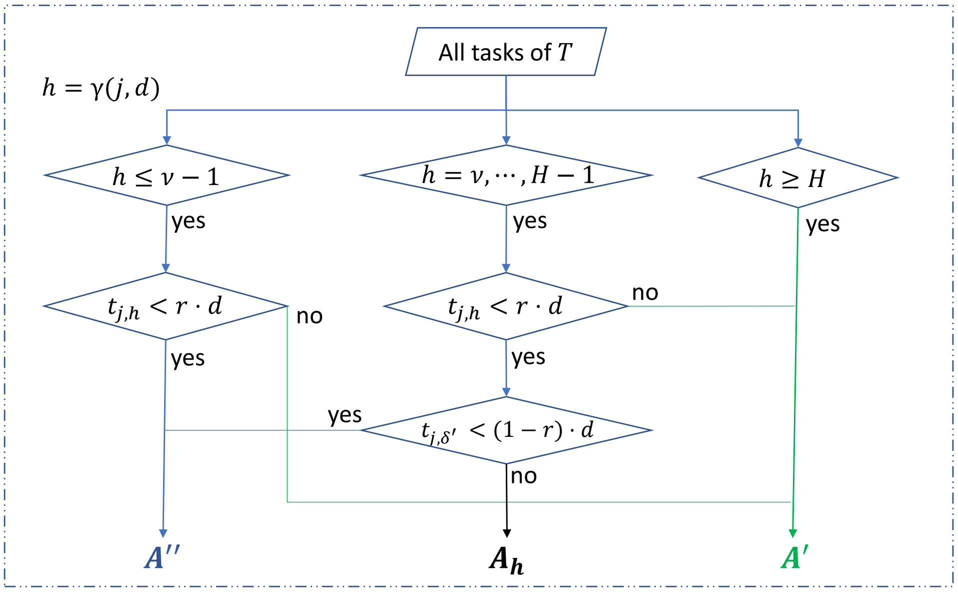

Figure 5.1 – Task Classification

Proposition 15. *Given an arbitrary $\delta \geq 1$, there exists a set of feasible parameters H, δ', ν, and $x_\nu, \cdots, x_{H-1}$ that satisfy Inequalities (5.2b), (5.2c), (5.3b), (5.3c) and (5.5).*

Proof. Given an arbitrary $\delta \geq 1$, we set $H = 1 = \delta' = 1$ where $r = \frac{H-1}{H} = \frac{1}{2}$; then, we set $\nu = 1$ and $x_1 = 1$ where $h \in [\nu, H - 1]$. The values of these parameters satisfy Inequalities (5.2b), (5.2c), (5.3b), (5.3c) and (5.5). Hence, the proposition holds. □

In this thesis, our aim is to find the maximum H and feasible δ', ν, $x_\nu, \cdots, x_{H-1}$ such that Inequalities (5.2b), (5.2c), (5.3b), (5.3c) and (5.5) are satisfied where $r = (H - 1)/H$. Given the value of δ, we can achieve this by exhaustive search; a corresponding search procedure is presented in Algorithm 9; here the 5 loops that respectively begin at the lines 1, 2, 3, 8, and 9 of this procedure lead to that it has a time complexity $\mathcal{O}(\delta^5)$ where δ is a fixed system parameter.

5.4.2 Example

Now, we illustrate how to apply the above classification of tasks to propose a scheduling algorithm by considering a special case with $\delta = 5$. In the next subsection, we will give a generic schedule.

Using Algorithm 9 to obtain a set of feasible parameters satisfying Inequalities (5.2b), (5.2c), (5.3b), (5.3c) and (5.5), we have that $\nu = 2$, $H = 4$, $\delta' = 5$,

Algorithm 9: A Search Procedure to Obtain Feasible Parameters

```
/* seek the maximum possible $H$ under which there exist feasible $\delta'$, $\nu$, $x_\nu$,
   $\cdots$, $x_{H-1}$ such that Inequalities (2b), (2c), (3b), (3c) and (5) are
   satisfied                                                              */
/* By (5), we have $1 \leq H-1 \leq \delta$, $H-1 \leq \delta' \leq \delta$, and $1 \leq \nu \leq H-1$        */
1  for $H \leftarrow \delta + 1$ to $2$ do
2      for $\delta' \leftarrow \delta$ to $H-1$ do
3          for $\nu \leftarrow 1$ to $H-1$ do
4              $r \leftarrow (H-1)/H$;
5              $temp1 \leftarrow \frac{\nu}{\delta'} \cdot r + r$,   $temp2 \leftarrow \frac{\nu-1}{\delta'} \cdot r + r$;
6              if $temp1 \geq 1 \wedge temp2 < 1$ then
                   // (2b) and (2c) are satisfied; next, we go to seek feasible
                      $x_\nu, \cdots, x_{H-1}$ under the current $H$, $\delta'$, $\nu$
7                  $flag \leftarrow$ an array of $H-1$ elements whose initial values are
                    all set to 0;
                   /* for every $h \in [\nu, H-1]$, if a feasible $x_h$ exists, we will
                      set $flag(h)$ to 1                                  */
8                  for $h \leftarrow \nu$ to $H-1$ do
9                      for $x_h \leftarrow 1$ to $\left\lceil \frac{\delta'}{r \cdot h} \right\rceil$ do
                           // the upper bound of $x_h$ is determined by (3b)
10                         $temp3 \leftarrow \frac{h}{\delta'} \cdot r \cdot x_h$, $temp4 \leftarrow max\left\{1-r, \frac{h-1}{\delta'}\right\} \cdot x_h$;
11                         if $(temp3 \leq 1) \wedge (temp4 \geq r)$ then
                               // (3b) and (3c) are satisfied, and a feasible
                                  $x_h$ is found
12                             $flag(h) \leftarrow 1$;
13                             break;
14                 temp5 $\leftarrow$ the sum of all the element's values of the array
                    $flag$;
15                 if temp5 $= H - \nu$ then
                       // we have found all feasible $x_\nu, \cdots, x_{H-1}$ under the
                          maximum possible $H$
16                     exit Algorithm 9;
```

$x_3 = 2$, and $x_2 = 3$. Here, $r = \frac{H-1}{H} = \frac{3}{4}$. As a result, all tasks of $\mathcal{T}$ are finally divided into 4 subsets $\mathcal{A}'$, $\mathcal{A}_3$, $\mathcal{A}_2$, and $\mathcal{A}''$ as shown in (5.4):

(1) Every task of $\mathcal{A}'$ has an execution time $\geq r \cdot d$ when assigned $\gamma(j, d)$ processors by Proposition 12.

(2) Every $x_3 = 2$ tasks of $\mathcal{A}_3$ or every $x_2 = 3$ tasks of $\mathcal{A}_2$ have a total execution time in $[r \cdot d,\ d]$ when sequentially executed on $\delta' = 5$ processors by Proposition 14.

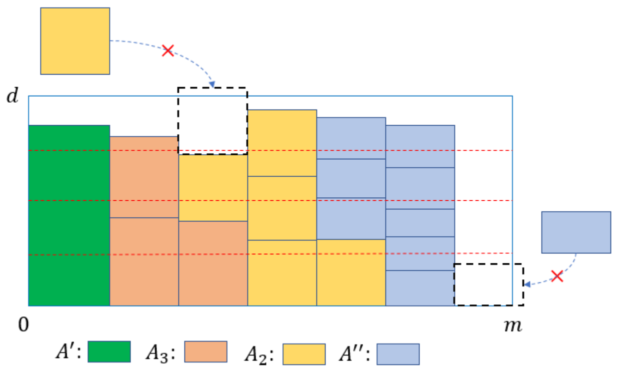

Figure 5.2 – Basic Scheduling Structure

(3) Every task of $\mathcal{A}''$ has an execution time $< (1-r)\cdot d$ when assigned δ' processors by Proposition 13.

We illustrate the proposed schedule in Figure 5.2 where each green, orange, gold, and blue rectangle denotes a task in $\mathcal{A}'$, $\mathcal{A}_3$, $\mathcal{A}_2$, and $\mathcal{A}''$, respectively; the particular way to schedule $\mathcal{T}$ is as follows:

Step 1. Assign every (green) task T_j of $\mathcal{A}'$ on the leftmost $\gamma(j,d)$ unoccupied processors.

Step 2. The remaining tasks are considered in the order of $\mathcal{A}_3, \mathcal{A}_2, \mathcal{A}''$.

(a) Assign as many (orange) tasks of $\mathcal{A}_3$ on the leftmost $\delta' = 5$ unoccupied processors by time d.

(b) When there is one (orange) task of $\mathcal{A}_3$ left, assign it to the leftmost δ' unoccupied processors; then, assign as many (gold) tasks of $\mathcal{A}_2$ as possible on these δ' processors by time d.

(c) For the remaining tasks of $\mathcal{A}_2$, assign as many (gold) tasks of $\mathcal{A}_2$ to the leftmost δ' unoccupied processors by time d.

(d) When there is one (gold) task left in $\mathcal{A}_2$, assign it to the leftmost δ' unoccupied processors; then, assign as many (blue) tasks in $\mathcal{A}''$ as possible by time d.

(e) Subsequently, assign as many (blue) tasks in $\mathcal{A}''$ on the leftmost δ' unoccupied processors by time d.

(f) Stop assigning tasks to processors when there are $< \delta'$ processors, even if there are still remaining tasks that are unassigned.

In Figure 5.2, the $\gamma(j,d)$ or δ' processors that are used to exclusively process the task(s) with the same color have a utilization $\geq r \cdot d$, which is due to Proposition 12, Observation 5.4.2 and Observation 5.4.1; here, the tasks of $\mathcal{A}'$, $\mathcal{A}_3$, $\mathcal{A}_2$, and $\mathcal{A}''$ are processed respectively in Steps (1), (2.a), (2.c), and (2.e). In Step (2.d), both gold and blue tasks are assigned and when it stops assigning tasks, there exists a (blue) task of $\mathcal{A}'$ that cannot be completed by time d; by Observation 5.4.1, the δ' processors here have a utilization $\geq r$ in $[0, d]$. As a result, there are only two places where the processors might have a utilization $< r$ in $[0, d]$. The first place is Step (2.f) where the schedule stops: there are at most $\delta' - 1$ processor idle. The second place is Step (2.b) where both orange and gold tasks are assigned; here, when this step finishes, there exists a (gold) task of $\mathcal{A}_2$ that cannot be completed by time d. By the second point of Observation 5.4.2, the δ' processors here have a utilization $\geq 1 - \frac{h}{\delta'} \cdot r = \frac{7}{10}$ where $h = 2$. In Steps (2.b) and (2.f), there are a total of $2 \cdot \delta' - 1$ processors whose utilization is $< r$. Finally, the above schedule achieves an average utilization of at least

$$\begin{aligned}&\frac{(m - 2\delta' + 1) \cdot r \cdot d + \delta' \cdot (d - \frac{3}{10} \cdot d) + (\delta' - 1) \cdot 0}{m \cdot d} = \frac{3}{4} - \frac{3.25}{m} = \\ &r - \frac{\delta' - 1}{m} \cdot r - \frac{\delta'}{m} \cdot \left(r - (1 - \frac{h}{\delta'} \cdot r) \right),\end{aligned} \tag{5.6}$$

where $h = 2$. We emphasize that in the above example there are two places where the occupied processors have a utilization $< r$ in $[0, d]$: (i) Step (2.f) where the schedule stops, and (ii) Step (2.b) where tasks from multiple subsets are sequentially executed on δ' processors and there exists a task from $\mathcal{A}_{H-2}, \cdots, \mathcal{A}_\nu$ (excluding $\mathcal{A}_{H-1}$ and $\mathcal{A}''$) that cannot be completed by time d on these processors.

5.4.3 Generic Schedule

In this subsection, we generalize the example in Chapter 5.4.2 to the case with an arbitrary δ; the corresponding algorithm is presented in Algorithm 10, also referred to as UnitAlgo; as seen later, its utilization is a generalization of Expression (5.6). In Algorithm 10, the set $\mathcal{T}$ of tasks is partitioned in several sets $\mathcal{A}', \mathcal{A}_{H-1}, \cdots, \mathcal{A}_v, \mathcal{A}''$, and these sets are also sorted and considered in this order where the tasks in each set will be chosen in a randomized order:

(i) Assign every task T_j of $\mathcal{A}'$ to $\gamma(j, d)$ idle processors alone.

(ii) For every δ' idle processors, assign onto them as many tasks as possible from the remaining unassigned tasks of $\mathcal{A}_{H-1} \cup \cdots \cup \mathcal{A}_v \cup \mathcal{A}''$ such that the sequential execution time of these tasks is $\leq d$ (see the formal description in the lines 12-15 of Algorithm 10).

Algorithm 10 ends when there is no enough idle processors for assigning the remaining tasks of $\mathcal{T}$ (i.e., it ends at line 9 or 17) or when all tasks of $\mathcal{T}$ have been assigned onto the m processors by time d (i.e., the conditions at lines 3 and 10 have become false). Recall that we consider in this section the situation that m processors are not enough for completing $\mathcal{T}$ by time d and our subsequent analysis of Theorem 3 is based on this.

Now, we give some explanation of the lines 12-15 of Algorithm 10. We also denote $\mathcal{A}''$ by $\mathcal{A}_{\nu-1}$ to be uniform with $\mathcal{A}_{H-1}, \cdots, \mathcal{A}_\nu$; in Algorithm 10, the initial $\mathcal{X}_{H-1}, \cdots, \mathcal{X}_{\nu-1}$ are set to $\mathcal{A}_{H-1}, \cdots, \mathcal{A}_{\nu-1}$ for recording the currently

Algorithm 10: UnitAlgo

```
1  $\mathcal{X}' \leftarrow A'$, $\mathcal{X}_{H-1} \leftarrow A_{H-1}, \cdots, \mathcal{X}_{\nu} \leftarrow A_{\nu}$, $\mathcal{X}_{\nu-1} \leftarrow A''$        // $X'$,
   $\mathcal{X}_{H-1}, \cdots, \mathcal{X}_{\nu}$, $\mathcal{X}_{\nu-1}$ are used to record the currently unassigned tasks
   during the algorithm execution
2  $m' \leftarrow m$        // record the number of currently unoccupied processors
3  while $\mathcal{X}' \neq \emptyset$ do
4      Randomly choose a task $T_j \in \mathcal{X}'$.
5      if $\gamma(j,d) \leq m'$ then
               /* the following corresponds to Step (1) in the example of
           Chapter 5.4.2 */
6          Assign the unassigned task $T_j$ onto the leftmost $\gamma(j,d)$
           unoccupied processors.
7          Remove $T_j$ from $\mathcal{X}'$: $\mathcal{X}' \leftarrow \mathcal{X}' - \{T_j\}$, and reset the number of
           currently unoccupied processors: $m' \leftarrow m' - \gamma(j,d)$.
8      else
9          exit Algorithm 10    // the currently unoccupied processors are not
           enough
10 while $\mathcal{X}_{H-1} \cup \cdots \cup \mathcal{X}_{v} \cup \mathcal{X}_{\nu-1} \neq \emptyset$ do
11     if $\delta' \leq m'$ then
               /* each loop corresponds to one of the Steps (2.a)-(2.e) in
           Chapter 5.4.2 */
12         Consider the tasks in the order of $\mathcal{X}_{H-1}, \cdots, \mathcal{X}_{\nu}, \mathcal{X}_{\nu-1}$; here, a
           task in $\mathcal{X}_i$ will never be chosen if the set $\mathcal{X}_{i+1}$ is non-empty, and
           the tasks in each set will chosen in a randomized order.
13         Choose as many tasks as possible from $\mathcal{X}_{H-1}, \cdots, \mathcal{X}_{\nu}, \mathcal{X}_{\nu-1}$ such
           that they could be sequentially completed by time $d$ on $\delta'$
           processors, denoting the set of the chosen tasks by $\mathcal{C}$.
14         Assign the chosen tasks onto the leftmost $\delta'$ unoccupied
           processors.
15         Remove the assigned tasks:
           $\mathcal{X}_{H-1} \cup \cdots \cup \mathcal{X}_{v} \cup \mathcal{X}_{\nu-1} \leftarrow \mathcal{X}_{H-1} \cup \cdots \cup \mathcal{X}_{v} \cup \mathcal{X}_{\nu-1} - \mathcal{C}$, and reset
           the number of currently unoccupied processors:
           $m' \leftarrow m' - \gamma(j,d)$.
16     else
17         exit Algorithm 10        // it corresponds to Step (2.f) in
           Chapter 5.4.2
```

unassigned tasks of $\mathcal{T}$. For every δ' processors, there are two cases upon completion of the task assignment onto them:

1. The δ' processors are used to sequentially process tasks from the same subset alone, i.e. some $\mathcal{A}_h$ where $h \in [\nu - 1, H - 1]$;
2. Tasks from multiple subsets will be processed in these processors. In this case, assume the first task that is assigned onto these processors is from $\mathcal{A}_h$ where $h \in [\nu,\, H - 1]$ and subsequently some tasks from the sets $\mathcal{A}_{h-1}, \cdots, \mathcal{A}_{h'}$ are also assigned where $h' \leq h - 1$; here, the value of h' is such that, after finishing the task assignment on these processors in $[0,\, d]$, (i) there is no unassigned tasks in $\mathcal{A}_h, \cdots, \mathcal{A}_{h'+1}$, and (ii) there still exist unassigned tasks in $\mathcal{A}_{h'}$ that cannot be completed by time d.

Examples of the first case include the Steps (2.a), (2.c), and (2.e) in Chapter 5.4.2 where every δ' processors process the same colored tasks; examples of the second case include Steps (2.b) and (2.d), e.g., in Step (2.b), both orange and gold tasks are processed where $h = 3$ and $h' = 2$.

In the first case, we conclude by Observation 5.4.1 and the first point of Observation 5.4.2 that

— no matter whether the processed tasks are from $\mathcal{A}_{\nu-1}$ or from $\mathcal{A}_{H-1}, \cdots, \mathcal{A}_{\nu}$, these δ' processors have a utilization $\geq r$ in $[0, d]$.

In the second case, since $h \in [\nu,\, H-1]$ and $h' \leq h-1$, we have $h' \in [\nu-1, H-2]$: we conclude by the second point of Observation 5.4.2 that

— if $h' \in [\nu, H - 2]$, these processors have a utilization $\geq 1 - \frac{h'}{\delta'} \cdot r$;

we conclude by Observation 5.4.1 that

— if $h' = \nu - 1$, these processors have a utilization $\geq r$.

By observing the possible values of h', we conclude that in the while loop of lines 10-17 there are at most $(H - \nu - 1) \cdot \delta'$ processors whose utilization in $[0, d]$ is $< r$; here, the second case occurs $H - \nu - 1$ times with $h' \in [\nu, H - 2]$.

Theorem 3. *With a time complexity of $\mathcal{O}(n)$, UnitAlgo achieves a resource utilization of at least*

$$\theta(\delta) = \begin{cases} r - \frac{r\cdot(k-1)}{m}, & \text{if } \nu = H - 1, \\ r - \max\left\{\frac{r\cdot(k-1)}{m}, \frac{r\cdot(\delta'-1)+\sum_{h=\nu}^{H-2}(r+\frac{h\cdot r}{\delta'}-1)\cdot\delta'}{m}\right\}, & \text{otherwise,} \end{cases}$$

where ν, δ', r, and H are computed by Algorithm 9.

Proof. At the line 6 of Algorithm 10, when a task is allocated $\gamma(j, d)$ processors, these processors achieve a utilization $\geq r$ in $[0,\, d]$ by Proposition 12.

Firstly, we analyze the case that Algorithm 10 stops at line 9; then, there are at most $\gamma(j, d) - 1$ processors idle and the average utilization of the m processors is $\geq$

$$\frac{(m - \gamma(j, d) + 1) \cdot r \cdot d}{m \cdot d} \geq r - \frac{r \cdot (k - 1)}{m}, \tag{5.7}$$

where $\gamma(j, d) \leq k$.

Secondly, we analyze the case that Algorithm 10 stops at line 17; then, there are at most $\delta' - 1$ processors idle. In this case, lines 12-15 will be executed where two cases will occur, which have been explained above. As explained, in the first case, for every $h \in [\nu - 1, H - 1]$, the δ' assigned processors have a utilization $\geq r$ in $[0, d]$.

In the second case, we discuss two possibilities for $H - 1$ and ν. The first possibility is $H - 1 = \nu$; then the value of h can only be ν and we have $h' = \nu - 1$. As a result, the δ' processors have a utilization $\geq r$ in $[0, d]$. Hence, if $H - 1 = \nu$, there are at most $\delta' - 1$ idle processors that occur at line 17, and the utilization of the m processors is $\geq$

$$\frac{(m - \delta' + 1) \cdot r \cdot d}{m \cdot d} \geq r - \frac{r \cdot (\delta' - 1)}{m}. \tag{5.8}$$

To sum up, if $H - 1 = \nu$, the worst-case utilization is the minimum of (5.7) and (5.8), i.e., (5.7); it happens when Algorithm 10 exits at line 9.

The other possibility is $H - 1 > \nu$ and the worst-case utilization occurs when $h' = h - 1$ for every $h \in [\nu + 1, H - 1]$: there is a total of $(H - 1 - \nu) \cdot \delta'$ processors whose utilization is $< r \cdot d$ in $[0, d]$. Hence, when Algorithm 10 stops at line 17, the utilization of the m processors is $\geq$

$$\begin{aligned} &\frac{(m - (\delta' - 1) - (H - 1 - \nu) \cdot \delta') \cdot r \cdot d + \sum\limits_{h'=\nu}^{H-2} \delta' \cdot (1 - \frac{r \cdot h'}{\delta'}) \cdot d + (\delta' - 1) \cdot 0}{m \cdot d} \\ &= r - \frac{r \cdot (\delta' - 1) + \sum_{h=\nu}^{H-2} (\frac{r \cdot h}{\delta'} + r - 1) \cdot \delta'}{m}. \end{aligned} \tag{5.9}$$

As a result, if $H - 1 > \nu$, the worst-case utilization is the minimum of (5.7) and (5.9).

By summarizing the utilizations derived above, we conclude that the worst-case utilization is (5.7) if $H - 1 = \nu$, and the minimum of (5.7) and (5.9) if $H - 1 > \nu$. Finally, in Algorithm 10, tasks are sequentially considered and assigned to processors one by one and it stops until there are not enough processors to assign the remaining tasks (see line 9 or 17 of Algorithm 10); hence, Algorithm 10 has a time complexity of $\mathcal{O}(n)$. $\square$

In Theorem 3, we let $\theta(\delta) = \mu(\delta) - \vartheta(\delta)$ where $\mu(\delta) = r$ and $\vartheta(\delta) = \max\left\{\beta_1 \cdot \frac{k-1}{m}, \beta_2 \cdot \frac{1}{m}\right\}$; here, if $\nu = H - 1$, $\beta_2 = 0$. Under a particular δ, the value of $\mu(\delta)$ is illustrated in Table 5.1, and the values of β_1 and β_2 are illustrated in Table 5.2; since β_1 and β_2 under a particular δ are constants and k is a system parameter, we have

$$\theta(\delta) = \mu(\delta) - \mathcal{O}(\tfrac{1}{m}).$$

Given a set of tasks $\mathcal{T}$, let $\mathcal{S}$ denote the subset of tasks that are selected by Algorithm 10 and scheduled on m processors. We give the following lemma that will be used in Chapter 5.5.

Table 5.2 – The values of β_1 and β_2 when δ is in different ranges.

δ	$\{\beta_1, \beta_2\}$	δ	$\{\beta_1, \beta_2\}$	δ	$\{\beta_1, \beta_2\}$
[5, 9]	$\{\frac{3}{4}, 3.25\}$	[22, 26]	$\{\frac{6}{7}, 19.43\}$	58	$\{\frac{9}{10}, 53.2\}$
[10, 16]	$\{\frac{4}{5}, 7.6\}$	[27, 37]	$\{\frac{7}{8}, 25.75\}$	[59, 74]	$\{\frac{10}{11}, 58.55\}$
[17, 21]	$\{\frac{5}{6}, 13.83\}$	[38, 57]	$\{\frac{8}{9}, 36.22\}$	[75, 101]	$\{\frac{11}{12}, 74\}$

Lemma 23. *In the schedule of $\mathcal{S}$ output by Algorithm 10, each task $T_j \in \mathcal{S}$ has a workload $D_{j,\gamma(j,d)}$ to be processed; here, $D_{j,\gamma(j,d)}$ is the minimum workload needed to be processed in order to be complete T_j by time d.*

Proof. All tasks of $\mathcal{T}$ are divided into several subsets: $\mathcal{A}'$, $\mathcal{A}_{H-1}$, $\cdots$, $\mathcal{A}_\nu$, $\mathcal{A}''$. To complete T_j by d, the minimum number of processors needed is $\gamma(j,d)$ and the minimum workload to be processed is $D_{j,\gamma(j,d)}$. In Algorithm 10, every scheduled task $T_j \in \mathcal{A}'$ is allocated $\gamma(j,d)$ processors. For every scheduled task $T_j \in \mathcal{A}_{H-1}, \cdots, \mathcal{A}_\nu, \mathcal{A}''$, we have $\gamma(j,d) \leq H-1 \leq \delta$ and it is executed on δ' processors where $\delta' \leq \delta$. Due to Definition 6, the workload of a task is a constant when assigned $\leq \delta$ processors, and thus the minimum workload of T_j is processed. Hence, the lemma holds. □

5.5 Optimizing Objectives

In this section, we propose algorithms to separately optimize two specific scheduling objectives: (i) minimizing the makespan, and (ii) maximizing the total value of tasks completed by a deadline.

5.5.1 Makespan

The main idea for makespan minimization has been introduced in Chapter 5.2. Built on Algorithm 10, we propose in this subsection a binary search procedure to find a feasible makespan for all tasks of $\mathcal{T}$, referred to as the OMS algorithm (Optimized MakeSpan). It proceeds as follows. At its beginning, let U and L be such that UnitAlgo can produce a feasible schedule of all tasks by time U but fails to do so by time L, e.g., $U = 2 \cdot n \cdot \max_{T_j \in \mathcal{T}}\{t_{j,1}\}$ and $L = 0$; we explain the reason why UnitAlgo can produce a schedule of all tasks by time U at the end of this subsection. The OMS algorithm will repeatedly execute the following operations, and stop when $U \leq L \cdot (1+\epsilon)$ where $\epsilon \in (0,1)$ is small enough:

1. $M \leftarrow \frac{U+L}{2}$;
2. if Algorithm 10 fails to produce a feasible schedule for all tasks of $\mathcal{T}$ by the deadline M (i.e., Algorithm 10 exits at line 9 or 17), set $L \leftarrow M$;
3. otherwise, set $U \leftarrow M$, and Algorithm 10 produces a feasible schedule of $\mathcal{T}$ by time M.

When the OMS Algorithm stops, we have that (i) Algorithm 10 could produce a feasible schedule of $\mathcal{T}$ with a makespan $\leq U$ but $> L$, and, (ii) $U \leq (1+\epsilon) \cdot L$. Recall that the makespan is the maximum completion time of tasks, and a schedule of all tasks with a makespan $\leq d$ represents a schedule that completes all tasks by the deadline d.

Theorem 4. *The OMS algorithm gives a* $\frac{1}{\theta(\delta)} \cdot (1+\epsilon)$*-approximation to the makespan minimization problem with a time complexity of* $\mathcal{O}(n \log(\frac{n}{\epsilon}))$ *.*

Proof. By abuse of notation, let $\theta = \theta(\delta)$ and by Theorem 3 it is the worst-case utilization achieved by Algorithm 10. In the following, we consider the state when the OMS Algorithm ends. Algorithm 10 fails to generate a feasible schedule for all the tasks of $\mathcal{T}$ with a makespan L but could generate it with a makespan $\leq U$. Let d^* denote the optimal makespan when scheduling $\mathcal{T}$ on m processors where $d^* \leq U$. The proof proceeds by considering two cases where $L \leq d^*$ and $L > d^*$ respectively.

In the case where $L \leq d^*$, since $U \leq L \cdot (1+\epsilon)$, we have $\frac{U}{d^*} \leq 1+\epsilon$; the produced schedule with a makespan $\leq U$ has a makespan $\leq (1+\epsilon)$ times d^*. In the other case where $L > d^*$, we have for all $T_j \in \mathcal{T}$ that $\gamma(j, L) \leq \gamma(j, d^*)$ and further $D_{j,\gamma(j,L)} \leq D_{j,\gamma(j,d^*)}$ by Lemma 20. By time L, Algorithm 10 can only schedule a subset of $\mathcal{T}$ due to the capacity constraint; the processed workload of each scheduled task is $D_{j,\gamma(j,L)}$ by Lemma 23, and the total workload of the scheduled tasks is $\geq \theta \cdot m \cdot L$. In contrast, in an optimal schedule of all tasks of $\mathcal{T}$ to minimize the makespan, the workload of every task is $\geq D_{j,\gamma(j,d^*)}$ since the task needs to be assigned at least $\gamma(j, d^*)$ processors in order to complete it by time d^*. As a result, in an optimal schedule of $\mathcal{T}$, the total workload of $\mathcal{T}$ is $\leq m \cdot d^*$ but $> \theta \cdot m \cdot L$ since $D_{j,\gamma(j,L)} \leq D_{j,\gamma(j,d^*)}$. Further, we have that $d^* \geq \frac{\theta \cdot m \cdot L}{m} = \theta \cdot L$, and $\frac{U}{d^*} \leq \frac{U}{\theta \cdot L} \leq \frac{1}{\theta} \cdot (1+\epsilon)$. Since the OMS algorithm finally produce a schedule with a makespan $\leq U$, it achieves an approximation ratio $\frac{1}{\theta} \cdot (1+\epsilon)$.

Finally, the initial values of U and L are $2 \cdot n \cdot \max_{T_j \in \mathcal{T}}\{t_{j,1}\}$ and 0. The binary search stops when $U \leq L \cdot (1+\epsilon)$ and the number of iterations is $\mathcal{O}(\log(\frac{n}{\epsilon}))$. Further, at each iteration, Algorithm 10 is run and it has a time complexity $\mathcal{O}(n)$ and, as a result, the OMS algorithm has a complexity $\mathcal{O}(n \log(\frac{n}{\epsilon}))$. $\square$

Finally, we explain the reason why UnitAlgo can produce a schedule of all tasks by time $U = 2 \cdot n \cdot \max_{T_j \in \mathcal{T}}\{t_{j,1}\}$. The values of H, δ' and ν are determined by the parameter δ and could be computed by Algorithm 9. As introduced at the beginning of this chapter, we have that $\delta \geq 3$; then, we have $\nu \geq 2$ and $H \geq 3$ where $r = (H-1)/H \geq 2/3$. When the initial value of U is $2 \cdot n \cdot \max_{T_j \in \mathcal{T}}\{t_{j,1}\}$, which is $\geq$ two times the total execution time of all tasks when every task is assigned one processor, where $|\mathcal{T}| = n$. Then, we have for all $T_j \in \mathcal{T}$ that $\gamma(j, U) = 1$, and $t_{j,\gamma(j,U)} \leq U/2 < r \cdot U$. As illustrated in Fig. 1 in the main manuscript, all tasks of $\mathcal{T}$ belong to $\mathcal{A}''$, and the other $\mathcal{A}', \mathcal{A}_{H-1}, \cdots, \mathcal{A}_\nu$ are empty; all tasks of $\mathcal{A}''$ can be sequentially completed by time U on δ' processors since $t_{j,1} = t_{j,\delta'} \cdot \delta'$. Hence, UnitAlgo can produce a feasible schedule of all tasks of $\mathcal{T}$ by time U (mainly see lines 1 and 13 of Algorithm 10) when the value of U is $2 \cdot n \cdot \max_{T_j \in \mathcal{T}}\{t_{j,1}\}$.

5.5.2 Scheduling to Maximize the Total Value

In this subsection, we consider the objective of maximizing the social welfare, i.e., the sum of values of tasks completed by a deadline τ. The main idea of this subsection has been introduced in Chapter 5.2.

Analysis of a Generic Greedy Algorithm. For a (δ, k)-monotonic task T_j, we define v'_{j,p_j} as its marginal value $\frac{v_j}{D_{j,p_j}} = \frac{v_j}{p_j \cdot t_{j,p_j}}$ and it is the value obtained from processing a unit of workload of T_j when T_j is assigned p_j processors and completed by the deadline τ. When $p_j \leq \delta$, the workload of T_j is a constant by Definition 6 and so is its marginal value; however, it is possible that its workload becomes increasing with p_j when p_j ranges from $\delta + 1$ to k, with its marginal value decreasing with p_j.

In order to complete T_j by time τ, the minimum number of processors needed is $\gamma(j, \tau)$ by Definition 5; thus, $D_{j,\gamma(j,\tau)}$ is the minimum workload that remains to be processed. Let $v'_j = v_j / D_{j,\gamma(j,\tau)}$, and v'_j is the maximum possible marginal value of T_j that has to be completed by time τ. In this subsection, we assume without loss of generality that $v'_1 \geq v'_2 \geq \cdots \geq v'_n$. We propose a generic greedy algorithm called GenGreedyAlgo; here some scheduling algorithm is used but not specified for now, referred to as GS (short for Generic Schedule). GenGreedyAlgo is presented in Algorithm 11: it considers tasks in the non-increasing order of their maximum possible marginal values v'_j and finally finds the maximum i' such that GS can output a feasible schedule by time τ for $\mathcal{S}_{i'}$ that includes the first i' tasks of $\mathcal{T}$.

Algorithm 11: GenGreedyAlgo

```
1 initialize $\mathcal{S}_i = \{T_1, T_2, \cdots, T_i\}$ $(1 \leq i \leq n)$;
2 for $i \leftarrow 1$ to $n$ do
3     if a feasible schedule of $\mathcal{S}_i$ by time $\tau$ is output by GS then
4         $i' = i$;
5     else
6         exit;
```

Before analyzing GenGreedyAlgo, we first give an upper bound of the optimal social welfare of the problem of this subsection, denoted by $\mathcal{OPT}$. Let $\mathcal{S}'$ contain the first σ tasks of $\mathcal{T}$ with the highest marginal values such that the total workload of these tasks satisfies

$$D_{1,\gamma(1,\tau)} + D_{2,\gamma(2,\tau)} + \cdots + D_{\sigma-1,\gamma(\sigma-1,\tau)} + \beta \cdot D_{\sigma,\gamma(\sigma,\tau)} = m \cdot \tau,$$

where $\beta \in (0, 1]$; then, we have that

Lemma 24. *The sum $v_1 + v_2 + \cdots + v_{\sigma-1} + \beta \cdot v_\sigma$ is an upper bound of $\mathcal{OPT}$.*

Proof. The minimum workload of each task T_j is $D_{j,\gamma(j,\tau)}$ when assigned the minimum number of processors in order to be completed by τ. We consider a relaxed scheduling problem: each task T_j has a value v_j, a workload $D_{j,\gamma(j,\tau)}$, and a deadline τ; however, T_j can be executed on any number of processors and partial execution of T_j by time τ will yield partial value linearly proportional to the amount of processed workload D'_j, i.e., $v_j \cdot (D'_j/D_{j,\gamma(j,\tau)})$. In the problem of this subsection, we assume in an optimal solution that $\mathcal{S}^*$ denotes the set of tasks selected to be processed, and each task $T_j \in \mathcal{S}^*$ is assigned p^*_j processors; the optimal social welfare is $\sum_{T_j\in\mathcal{S}^*} v_j$. Since $\sum_{T_j\in\mathcal{S}^*} D_{j,p^*_j} \leq \tau \cdot m$ and $D_{j,p^*_j} \geq D_{j,\gamma(j,\tau)}$ where $\gamma(j,\tau) \leq p^*_j$, a feasible solution of the relaxed problem is scheduling $\mathcal{S}^*$ where the workload of each task is $D_{j,\gamma(j,\tau)}$; thus, we conclude that the optimal social welfare of this relaxed problem is an upper bound of $\sum_{T_j\in\mathcal{S}^*} v_j$.

In the relaxed problem, the maximum workload that can be processed on m processors in [0, τ] is $m\cdot\tau$, and it is equivalent to a fractional knapsack problem [62]: a knapsack has a capacity $m \cdot \tau$ and there are n divisible items, each with a size $D_{j,\gamma(j,\tau)}$ and a value v_j; its optimal solution is just like what is described before this lemma: choose the first σ items with the highest marginal values to fill the knapsack where possibly only a fraction of the last item could be packed into the knapsack. Hence, its optimal social welfare is $v_1+v_2+\cdots+v_{\sigma-1}+\beta\cdot v_\sigma$, which is an upper bound of the optimal social welfare of the problem of this subsection $\sum_{T_j\in\mathcal{S}^*} v_j$; the lemma thus holds. $\square$

Without loss of generality, we assume that every task T_j accepted for processing in GenGreedyAlgo is finally assigned p_j processors where $p_j \geq \gamma(j,\tau)$. To bound the optimal social welfare, we define for every scheduled task T_j a parameter

$$\alpha_j = \tfrac{D_{j,\gamma(j,\tau)}}{D_{j,p_j}}.$$

To complete a task T_j by time τ, the minimum workload to be processed is $D_{j,\gamma(j,\tau)}$, and the ratio α_j represents the efficiency of processing a single task T_j: as more processors are assigned, completing the same task possibly needs to occupy more resource for processing an increased workload but yields the same value v_j, possibly leading to a smaller α_j and a lower marginal value. In terms of the set of chosen tasks $\mathcal{S}_{i'}$, we define

$$\alpha = \min_{1\leq j\leq i'}\{\alpha_j\}.$$

We denote by ω the utilization of the m processors in $[0,\tau]$ achieved by GenGreedyAlgo, i.e.,

$$\textstyle\sum_{j=1}^{i'} D_{j,p_j} = \omega\cdot\tau\cdot m.$$

Finally, we have the following bound:

Theorem 5. *An upper bound of the optimal social welfare of our problem is $1/(\omega{\cdot}\alpha)$ times the social welfare achieved by GenGreedyAlgo, i.e., $\sum_{T_j\in\mathcal{S}_{i'}} v_j/(\omega\cdot\alpha)$.*

Proof. GenGreedyAlgo accepts the first i' tasks with the highest marginal values v_j', and the achieved social welfare is $\sum_{j=1}^{i'} v_j$. Let

$$X_1 = \frac{1}{\omega}\cdot\sum_{j=1}^{i'} D_{j,p_j} - \sum_{j=1}^{i'} D_{j,\gamma(j,\tau)}, \text{ and } X_2 = \sum_{j=i'+1}^{\sigma-1} D_{j,\gamma(j,\tau)} + \beta\cdot D_{\sigma,\gamma(j,\tau)};$$

recall $\tau\cdot m = (1/\omega)\cdot\sum_{j=1}^{i'} D_{j,p_j} = D_{1,\gamma(1,\tau)} + \cdots + D_{\sigma-1,\gamma(\sigma-1,\tau)} + \beta\cdot D_{\sigma,\gamma(\sigma,\tau)}$ and we have $X_1 = X_2$ since $\tau\cdot m - X_1 = \sum_{j=1}^{i'} D_{j,\gamma(j,\tau)} = \tau\cdot m - X_2$.

$\sum_{j=1}^{i'} v_j$ is the total value obtained by GenGreedyAlgo and we have

$$\begin{aligned}
&\frac{\sum_{j=1}^{i'} v_j}{\omega\cdot\tau\cdot m} = \frac{\sum_{j=1}^{i'} \frac{v_j}{D_{j,\gamma(j,\tau)}}\cdot\alpha_j\cdot D_{j,\gamma(j,p_j)}}{\omega\cdot\tau\cdot m} \geq \frac{\alpha\cdot\sum_{j=1}^{i'} v_j'\cdot D_{j,p_j}}{\omega\cdot\tau\cdot m}\\
&\overset{(a)}{=} \frac{\sum_{j=1}^{i'} v_j + \sum_{j=1}^{i'} v_j'\cdot(\frac{1}{\omega}\cdot D_{j,p_j} - D_{j,\gamma(j,\tau)})}{(\tau\cdot m)/\alpha} \overset{(b)}{\geq} \frac{\sum_{j=1}^{i'} v_j + v_{i'}'\cdot X_1}{(\tau\cdot m)/\alpha}\\
&\overset{(c)}{=} \frac{\sum_{j=1}^{i'} v_j + v_{i'}'\cdot X_2}{(\tau\cdot m)/\alpha} \overset{(d)}{\geq} \frac{\sum_{j=1}^{i'} v_j + \left(\sum_{j=i'+1}^{\sigma-1} v_j + \beta\cdot v_\sigma\right)}{(\tau\cdot m)/\alpha} \overset{(e)}{\geq} \frac{\mathcal{OPT}}{(\tau\cdot m)/\alpha}.
\end{aligned} \tag{5.10}$$

Here, in Equation (a), $v_j = v_j'\cdot D_{j,\gamma(j,\tau)}$; Inequalities (b) and (d) are due to that $v_1'\geq\cdots\geq v_{i'}\geq\cdots\geq v_n'$; (c) is due to that $X_1 = X_2$; (e) is due to Lemma 24. Due to Inequality (5.10), we have $\mathcal{OPT}\leq\sum_{j=1}^{i'} v_j/(\omega\cdot\alpha)$, and the theorem thus holds. □

Recall the definition of approximation algorithms in Chapter 5.5.2, and Theorem 5 shows that GenGreedyAlgo achieves an approximation ratio $\omega\cdot\alpha$. Now, it has been clear that a good greedy algorithm can be obtained only if the scheduling algorithm GS in GenGreedyAlgo could achieve both a high utilization in $[0, \tau]$ on m processors and a large α; the latter could be achieved by making the number of processors assigned to every task T_j close to $\gamma(j,\tau)$ where its workload to be processed is close to $D_{j,\gamma(j,\tau)}$.

In the following, we consider the case when GS is replaced by the scheduling algorithm UnitAlgo, i.e., Algorithm 10 in Chapter 5.4.

Proposition 16. *GenGreedyAlgo, with GS replaced by UnitAlgo, gives an $\theta(\delta)$-approximation algorithm with a complexity of $\mathcal{O}(n^2)$ for the social welfare maximization problem.*

Proof. Due to Lemma 23, the workload of each task T_j scheduled by UnitAlgo is $D_{j,\gamma(j,\tau)}$; we thus have $\alpha_j = 1$ and further $\alpha = 1$. As a result of Theorem 5, the approximation ratio of GenGreedyAlgo is the utilization achieved by UnitAlgo, which is bounded by $\theta(\delta)$ according to Theorem 3. Hence, GenGreedyAlgo is a $\theta(\delta)$-approximation algorithm. GenGreedyAlgo considers $\mathcal{S}_1,\cdots,\mathcal{S}_n$ one by one until UnitAlgo cannot produce a feasible schedule for some $\mathcal{S}_i$ ($i\in[1, n]$); when UnitAlgo attempts to schedule tasks on m processor by time τ, it has a time complexity at most $\mathcal{O}(n)$. Hence, the time complexity of GenGreedyAlgo is $\mathcal{O}(n^2)$. Finally, the proposition holds. $\square$

Part II

Cost-Optimal Use of Cloud Computing

Chapter 6

Introduction

A good designer must rely on experience, on precise, logic thinking; and on pedantic exactness. No magic will do.

Niklaus Wirth

6.1 Background

Many businesses possess a small infrastructure that they can use for their computing tasks, but also often need to buy extra computing resources from clouds, as illustrated in Fig. 6.1. Infrastructure as a Service (IaaS) holds exciting potential of elastically scaling users' computation capacity up and down to match their time-varying demand. This eliminates the users' need of purchasing servers to satisfy their peak demand, without causing an unacceptable latency. IaaS is seeing a fast growth and nowadays has become the second-largest public cloud subsegment [63], [64], accounting for almost half of all data center infrastructure shipments. Cost management in IaaS clouds is therefore a premier concern for users and has received significant attention as illustrated in Fig. 6.2.

Two common purchase options in the cloud are on-demand and spot instances. The former are always available with a fixed price and tenants[1] pay only for the period in which instances are consumed at an hourly rate. Users can also bid a price for spot instances and can successfully get them only if their bid price is above the spot price. Spot instances will then run as long as the bid is above the spot price but they will be terminated if the spot price becomes

1. In this thesis, we use "users" and "tenants" interchangeably.

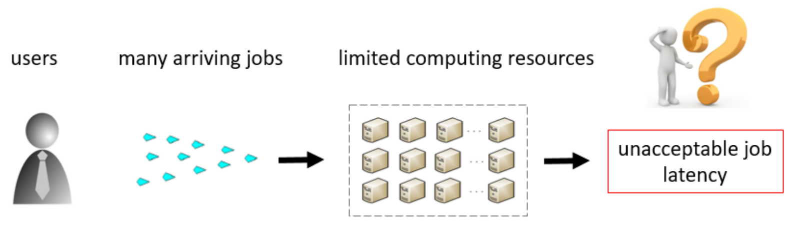

Figure 6.1 – Motivation of using IaaS clouds.

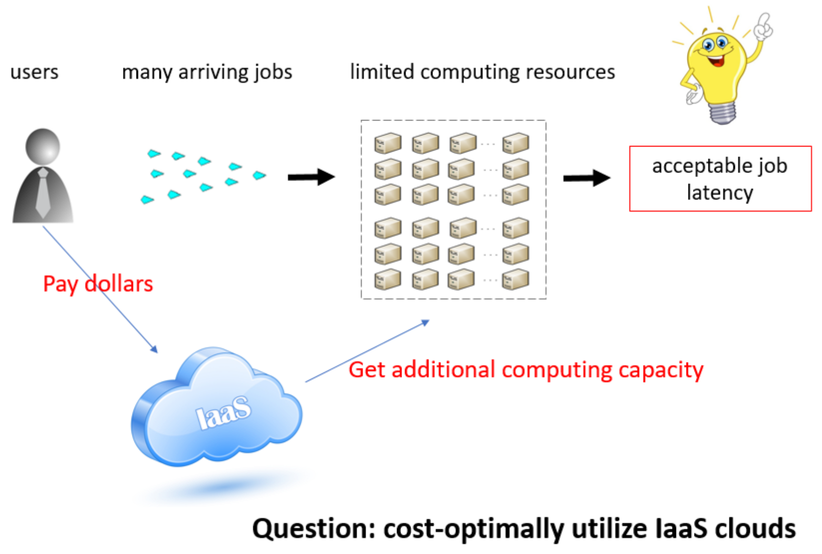
users
many arriving jobs
limited computing resources
acceptable job latency
Pay dollars
IaaS
Get additional computing capacity
Question: cost-optimally utilize IaaS clouds

Figure 6.2 – Our Question.

higher. Here, spot prices usually vary unpredictably over time and users will be charged the spot prices for their use [65]. Compared to on-demand instances, spot instances can reduce the cost by up to 50-90% [66].

Users purchasing instances on the cloud may have their own instances, referred to as self-owned instances, which can be used to process jobs but are insufficient at times (hence the need to purchase extra IaaS instances). They may also not have any self-owned instances (e.g., in the case of startups) and therefore need to buy from the cloud all necessary computing resources. In both cases though, the fundamental question for users is to determine how to purchase instances from IaaS clouds and utilize different instances to process their jobs in a way that minimizes their cost.

Tenants' jobs arrive over time and often have constraints that must be satisfied while trying to minimize cost [3], [6], [67], [68]. For example, one constraint is the parallelism bound and it specifies the maximum number of instances that could be utilized by a job simultaneously; another is on timing, i.e., a deadline by which to complete the job's workload. Subject to the parallelism constraint, an arriving job will be allocated instances of different types (self-owned, on-demand and spot) and the allocation can be updated at most once every hour (since billing is done per hour). The problem is then to find an allocation that minimizes cost while ensuring that every job will be completed by its deadline.

6.2 Challenges and Results

Challenges. In this part of the thesis, we make the natural assumption that self-owned instances are cheaper than spot instances, which are further cheaper than on-demand instances. So, to be cost-optimal, an allocation policy should allocate as many self-owned instances as possible, then spot instances, and finally on-demand instances. This is, however, a difficult task. For instance, *a naive policy* to achieve a high utilization of self-owned instances would be, when a job arrives, to assign as many remaining self-owned instances as possible to it. However, this policy turns out not to be good wrt cost. Indeed, it ignores the difference of jobs and treats all jobs equally when assigning instances, whereas we find that a good policy wrt cost needs instead to determine the allocations of self-owned instances to jobs according to their capabilities of utilizing spot instances alone to complete themselves by deadlines.

In particular, when self-owned instances are inadequate, actively assign self-owned instances to the jobs with poor capabilities and assign nothing to the others; otherwise, such poor jobs will have to consume more costly on-demand instances, and it also causes a waste of other rich jobs' capabilities together with self-owned instances (even if no self-owned instances are allocated, they can be completed by utilizing spot instances alone). When self-owned instances are adequate, assign them to jobs with both poor and strong capabilities such that after the allocations all jobs are expected to be completed by utilizing spot instances alone, eliminating the need of consuming costly on-demand instances.

After allocating self-owned instances, the left question is to identify a job's capacity for utilizing spot instances, i.e., the maximum workload that could be processed by spot instances, and propose an expected optimal policy to achieve such capacities of jobs, further escaping unnecessary consumption of on-demand instances.

Our Contributions. In this thesis, we propose a framework to design policies to allocate various instances [69], [70]. Based on the two principles that (i) self-owned instances should be allocated to maximize their utilization while maximizing the opportunity of all jobs utilizing spot instances and (ii) on-demand instances should be allocated to maximize the opportunity to utilize spot instances, we propose parametric policies for the allocation of self-owned, on-demand and spot instances that achieve near-minimal costs. To cope with the cloud market dynamic and the uncertainty of job's characteristics, we use the online learning technique in [6], [68] to infer the optimal parameters. More specifically:

— We propose a cost-effective policy for allocating self-owned instances that is smarter than the naive allocation mentioned above and hits a good trade-off between the utilization of self-owned instances and the opportunity of utilizing spot instances. We show in our numerical experiments that this policy improves the cost by up to 43.74% compared to the naive policy.

— We propose a cost-optimal policy for the utilization of on-demand and spot instances, based on a formulation of the original problem as an integer program to maximize the utilization of spot instances. This policy can be used both when the tenant has self-owned resources and when he does not. Our simulation results show that it improves the cost of previous policies in [6], [68] by up to 64.51%.

6.3 Related Work

In this thesis, we use online learning technique to learn the most-effective parameters for utilizing various instances. Jain *et al.* were the first to consider the application of this approach to the scenario of cloud computing [2] [6], [68]. However, they do not consider the problem of how to optimally utilize the purchase options in IaaS clouds and self-owned instances are also not taken into account. The online learning approach is interesting because it does not impose the restriction of a priori statistical knowledge of workload, compared to other techniques such as stochastic programming (see Chapter 8.4 for an introduction of online learning). However, it can achieve good performances only if the potentially optimal scheduling policies are identified among all possible policies.

Similar to our work and [6], [68], executing deadline-constrained jobs cost-effectively in IaaS clouds is also studied in [71], [72]. In particular, Zafer *et al.* characterize the evolution of spot prices by a Markov model and propose an optimal bidding strategy to utilize spot instances to complete a serial or parallel job by some deadline [71]. Yao *et al.* study the problem of utilizing reserved and on-demand instances to complete online batch jobs by their deadlines and formulate it as integer programming problems; then heuristic algorithms are proposed to give approximate solutions [72].

There have been substantial works on cost-effective resource provisioning in IaaS clouds [73], and we introduce some typical approaches. Built on the assumption of a priori statistical knowledge of the workload or spot prices, several techniques could be applied. For example, in [74], [75], the techniques of stochastic programming is applied to achieve the cost-optimal acquisition of reserved and on-demand instances; in [76], the optimal strategy for the users to bid for the spot instances are derived, given a predicted distribution over spot prices. However, a high computational complexity arises when implementing these techniques, though the statistical knowledge could be derived by the techniques such as dynamic programming [77].

2. The objective of this thesis corresponds to a special case of [6], [68] where the value of each job is larger than the cost of completing it.

Wang *et al.* use the competitive analysis technique to purchase reserved and on-demand instances without knowing the future workload [78], where the Bahncard problem is applied to propose a deterministic and a randomized algorithm. In [79], a genetic algorithm is proposed to quickly approximate the pareto-set of makespan and cost for a bag of tasks where on-demand and spot instances are considered. In [77], the technique of Lyapunov optimization is applied and it's said to be the first effort on jointly leveraging all three common IaaS cloud pricing options to comprehensively reduce the cost of users. The less interesting aspect of this technique is that a large delay will be caused when processing jobs; in order to achieve an $\mathcal{O}(\epsilon)$ close-to-optimal performance, the queue size has to be $\Theta(1/\epsilon)$ [80].

Chapter 7

Model, Assumption, and Objectives

Nothing has such power to broaden the mind as the ability to investigate systematically and truly all that comes under thy observation in life.

Marcus Aurelius

In this chapter, we introduce the cloud pricing models, define the operational space of a user to utilize various instances, and characterize the scheduling objective.

7.1 Pricing Models in the Cloud

We first introduce the pricing models in the cloud. The price of an *on-demand instance* is charged on an hourly basis and it is fixed and denoted by p. Even if on-demand instances are consumed for part of an hour, the tenant will be charged the fee of the entire hour, as illustrated in Fig. 7.1.

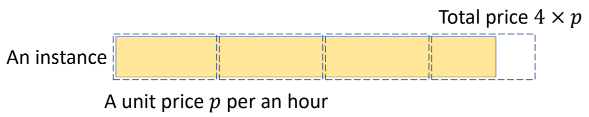

Figure 7.1 – On-demand price: users are charged on an hourly basis.

Furthermore, tenants can bid a price for *spot instances* and *spot prices are updated at regular time intervals/slots* (e.g., every $L = 5$ minutes in Amazon) [76]. Spot instances are assigned to a job and continue running if the spot price is lower than the bid price, as illustrated in Fig. 7.2. Since spot prices usually change unpredictably over time [65], once the spot price exceeds the bid price of a job, its spot instances will get lost suddenly and terminated immediately by the cloud; here, the termination occurs at the very beginning of a time slot. The tenant will be charged the spot prices for the maximum integer hours of execution. A partial hour of execution is not charged in the case where its instances are terminated by the cloud; in contrast, if spot instances run until a job is completed and then are terminated by the tenant, for the partial hour of execution, the tenant will also be charged for the full hour.

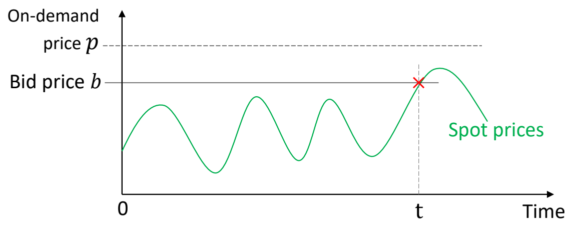

Figure 7.2 – Spot price: a user bid a price b for an instance at time 0 and can use it until time t.

Finally, a user might have its own computing instances, i.e., *self-owned instances*. The (averaged) hourly cost of utilizing self-owned instances is assumed to be p_1. We assume that it is the cheapest to use self-owned instances so that p_1 is without loss of generality assumed to be 0. An example of self-owned instances is academic private clouds, which are provided to researchers free of charge.

7.2 Jobs to be Processed

The job arrival of a tenant is monitored every time slot of L minutes (i.e., at the time points when spot prices change) and time slots are indexed by $t = 1, 2, \cdots$. Each job j has four characteristics: (i) *an arrival slot* a_j: If job j arrives at a certain continuous time point in $[(t-1) \cdot L, t \cdot L)$, then set a_j to t; (ii) *a relative deadline* $d_j \in \mathcal{Z}^+$: it is a time constraint on completing a job, that is, every job must be completed at or before time slot $a_j + d_j - 1$; (iii) *a job size* z_j (measured in CPU time slots that need to be utilized), i.e., the workload to complete j; (iv) *a parallelism bound* δ_j: the upper bound on the number of instances that could be simultaneously utilized by j. The tenant plans to rent instances in IaaS clouds to process its jobs and aims to minimize the cost of completing a set of jobs $\mathcal{J}$ (that arrive over a time horizon T) by their deadlines.

7.3 Operational Rules for Allocating Instances to Jobs

The pricing models define the rules of allocating instances to jobs and also the operational space of a user, i.e., (a) when the resource allocation to jobs is done and updated, and, (b) how various instances and especially spot instances are utilized by jobs at every allocation update. Generally, the allocation of various instances to every job will be simultaneously updated at most once every hour, since on-demand instances are charged on an hourly basis. Upon arrival of every job, the allocation to it is done immediately due to the time constraint. Now, we elaborate this.

7.3.1 On-demand and spot instances

We first consider the allocation of on-demand and spot instances alone, ignoring self-owned instances temporarily.

To meet deadlines, we assume that (**i**) *whenever a job j arrives at a_j, the allocation of spot and on-demand instances to it is done immediately.* The following rules apply to the case where j has flexibility to utilize spot instances. Given the fact that the tenant is charged on hourly boundaries, (**ii**) *the allocation of on-demand and spot instances to each job j is updated simultaneously every hour.* At the i-th allocation to j, the number of on-demand instances allocated to j is denoted by o_j^i and they can be utilized for the entire hour; we assume that (**iii**) *the tenant will bid a price b_j^i for a fixed number si_j^i of spot instances.* At the i-th allocation of j, b_j^i together with the spot prices determines whether j can successfully obtain spot instances and how long it can utilize them. Usually, spot instances are on average cheaper than on-demand instances, and we assume that (**iv**) *at every allocation the tenant will bid for the maximum number of spot instances under the parallelism constraint, i.e., $si_j^i = \delta_j - o_j^i$.* The crucial question is thus how to determine the proportion of on-demand and spot instances, i.e., o_j^i and si_j^i, that are acquired from the cloud.

Before the i-th allocation to j, we use z_j^i to denote the remaining workload of j to be processed, i.e., z_j minus the workload of j that has been processed, where $z_j^1 = z_j$, and we define the current slackness of j as

$$s_j^i = \frac{(d_j - (i-1) \cdot Len) \cdot \delta_j}{z_j^i}, \tag{7.1}$$

where $Len = 60/L$ is the number of slots per hour. Let $s_j = s_j^1 = (d_j \cdot \delta_j)/z_j$. The slackness can be used to measure the time flexibility that j has to utilize spot instances; the process of allocating on-demand and spot instances to j is in fact divided into two phases by the value of s_j^i:

Definition 7. *When spot instances get lost at the very beginning of slot t' and are not utilized for the entire hour at the i-th allocation of j, we say that, at the next allocation,*

1. j has flexibility to utilize spot instances, if $s_j^{i+1} \geq 1$;
2. j does not have such flexibility, otherwise.

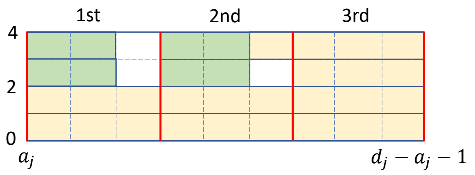

Figure 7.3 – Illustration of the process of allocating resource to j where $\delta_j = 4$, d_j equals 3 hours, $L = 5$ minutes and $z_j = 132$: the area between two red lines illustrates the workload processed at every allocation and the height of green (resp. yellow) areas denote the number of spot (resp. on-demand) instances.

Now, we illustrate Definition 7 by Fig. 7.3. As illustrated in Fig. 7.3, $z_j = z_j^1 = 132$ and, at the 1st allocation, $o_j^1 = si_j^1 = 2$; then $z_j^2 = 132 - 2 \cdot 12 - 2 \cdot 8 = 92$. At the 2nd update, o_j^2 and si_j^2 are still 2 and then $z_j^3 = 92 - 2 \cdot 12 - 2 \cdot 8 = 52$. Further, $s_j^3 = \frac{Len \cdot \delta_j}{z_j^3} < 1$ and there is no flexibility for j to utilize unstable spot instances at the 3rd allocation. We use i_j to index the last allocation of j after which there is no flexibility to utilize spot instances; in Fig. 7.3, $i_j = 2$. As a result, the decision on how to determine the $(i_j + 1)$-th allocation of instances to j has to be done earlier, since there exists an on-demand instance that has to be utilized for $\frac{4}{3}$ hours to satisfy the deadline constraint.

As illustrated above, the instance allocation is divided into two phases. In the first phase,

— the instance allocation is updated every hour (i.e., every Len slots).

At every i-th allocation of j, the remaining workload to be processed by spot and on-demand instances is z_j^i; on-demand instances are charged on an hourly basis and the workload that could be processed by on-demand instances is $Len \cdot o_j^i$. At every i-th allocation, as time goes by, there are two possible states for spot instances:

(i) if $z_j^i - Len \cdot o_j^i$ workload of j has been processed by spot instances, they will be terminated by the user itself;
(ii) if the bid price is below the spot price, the user will lose its spot instances immediately; otherwise, they will be utilized for an hour.

The first state occurs because j will be finally completed after the on-demand instances acquired at this allocation are consumed. If the second state occurs, we need to check whether or not job j still has flexibility to utilize spot instances using Definition 7: if there is such flexibility, the next allocation update of j is still in the first phase; otherwise,

— the next allocation of j (i.e., the $(i_j + 1)$-th allocation) needs to be done immediately after the spot instances of the i_j-th allocation get lost,

which is referred to as the second phase of instance allocation. In the second phase, only stable on-demand instances will be used to meet the deadline.

7.3.2 Self-owned instances

When self-owned instances are also taken into account, we assume that (**v**) *the allocation of self-owned instances to a job can be updated at most once at every allocation of j.* We denote by r_j^i the number of self-owned instances assigned to j at the i-th allocation; the parallelism constraint further translates to $o_j^i + si_j^i + r_j^i = \delta_j$. In this thesis, o_j^i and si_j^i denotes the numbers of on-demand and spot instances acquired at the i-th allocation and will be used to track the cost of completing j. As we will see in Chapter 8.2, the acquired on-demand instances may not be fully utilized for an entire hour at the i_j-th allocation, and, we use $o_j(t)$, $si_j(t)$ and $r_j(t)$ to denote the numbers of on-demand, spot and self-owned instances that are actually utilized by j at every slot $t \in [a_j, a_j + d_j - 1]$, where $r_j(t) = r_j^i$ for all $t \in [a_j + (i-1) \cdot Len, a_j + i \cdot Len - 1]$; then the parallelism constraint translates to $o_j(t) + si_j(t) + r_j(t) = \delta_j$.

As shown later, allocating properly self-owned instances enables escaping unnecessary consumption of on-demand instances that are more expensive than the others, which can be achieved by simply allocating j the same number of self-owned instances at every time slot, i.e., $r_j(t) = r_j$. So, the allocation of self-owned instances is done only once upon arrival of j; after the allocation, the job can could be viewed as a new job with a parallelism bound $\delta_j - r_j$, a size $z_j - r_j \cdot d_j$, and the same arrival time and deadline as j , and it will be completed by utilizing spot and on-demand instances alone.

7.4 Scheduling Objectives

7.4.1 Proposed principles for cost efficiency

We refer to the ratio of the total cost of utilizing a certain type of instances to the total workload processed by this type of instances as the average unit cost of this type of instances. As described in Chapter 7.1, we assume that

***Assumption* 7.4.1.** *The average unit costs of self-owned instances is lower than the average unit cost of spot instances, which is lower than that of on-demand instances.*

Accordingly, to be cost-optimal, we should consider allocating various instances to each arriving job in the order of self-owned, spot and on-demand instances. Further, in Principles 7.4.1 and 7.4.2, we give the objectives that should be achieved when considering allocating each type of instances to the arriving jobs.

***Principle* 7.4.1.** *The scheduler should make self-owned instances (i) fully utilized, and (ii) utilized in a way so as to maximize the opportunity that all jobs have to utilize spot instances.*

***Principle* 7.4.2.** *After self-owned instances are used, the scheduler should utilize on-demand instances in a way so as to maximize the opportunity that all jobs have to utilize spot instances.*

Table 7.1 – Main Notation

Symbol	Explanation
L	length of a time slot (e.g., 5 minutes)
Len	the number of time slots in an hour, i.e., $\frac{60}{L}$
$\mathcal{J}$	a set of jobs that arrive over time
j and a_j	a job of $\mathcal{J}$ and its arrival time
d_j	the relative deadline: j must be completed by a deadline $a_j + d_j - 1$
z_j	the job size of j, measured in CPU $\times$ time slots
δ_j	the parallelism bound, i.e., the maximum number of instances that can be simultaneously used by j
s_j	the slackness, i.e., $\frac{d_j}{z_j/\delta_j}$ where z_j/δ_j denotes the minimum execution time of j
T	the number of time slots, i.e., $\max_{j\in\mathcal{J}}\{a_j\}$
si_j^i, b_j^i, and o_j^i	the number of spot instances bid for, the bid price, and the number of on-demand instances acquired at the i-th allocation update of j
$r_j(t)$, $si_j(t)$ and $o_j(t)$	the number of self-owned, spot and on-demand instances utilized by j at a slot t
p_j^i	the spot price charged at the i-th allocation of j
z_j^i	the remaining workload of j to be processed at the i-th allocation update to j
s_j^i	the slackness at the i-th allocation update, i.e., $(d_j - (i-1)\cdot Len)\cdot \delta_j / z_j^i$
p and p_1	the price of respectively using an on-demand and self-owned instance for an hour
R	the number of self-owned instances
$\{\beta, \beta_0, b\}$	a tuple of parameters that defines a policy and determines the allocation of various instances to j at every allocation
$\mathcal{P}$	a set of parameterized policies, each indexed by π and defined by $\{\beta, \beta_0, b\}$
r_j	the number of self-owned instances allocated to a job j at every $t \in [a_j, a_j + d_j - 1]$
$N(t)$	the number of currently idle self-owned instances at a slot t
$m_{t_1}(t_2)$	the maximum number of self-owned instances idle at every slot in $[t_1, t_2]$, i.e., $\min\{N(t_1), \cdots, N(t_2)\}$

Principles 7.4.1 and 7.4.2 are intuitive under Assumption 7.4.1.

7.4.2 Decision variables

Our main objective is to propose scheduling policies that can realize Principles 7.4.1 and 7.4.2. To do so, we will first determine the allocation of self-owned instances and then the allocation of on-demand and spot instances for every arriving job j. For every arriving job j, it will be first allocated r_j self-owned instances in $[a_j, a_j + d_j - 1]$. Then, as described in Chapter 7.3, the allocation of spot and on-demand instances will be updated per hour in the first phase and we need to determine the number of spot instances to be bid for and the number of on-demand instances to be purchased (i.e., si_j^i and o_j^i); once there is no flexibility for j to utilize spot instances, we need to determine the allocation of on-demand instances alone in order to complete j by deadline. Hence, the main decision variables are r_j, si_j^i, and o_j^i where $o_j^i + si_j^i + r_j = \delta_j$.

In this part of the thesis, we apply the online learning approach and it does not require the exact statistical knowledge on jobs and spot prices. At every allocation update of j in the first phase, only the current characteristics of j (i.e., z_j^i, δ_j, a_j, and d_j) and the amount of available self-owned instances are definitely known for a user to determine o_j^i and si_j^i.

The value of spot price is jointly determined by the arriving jobs of numerous users and the number of idle servers at a moment, usually varying over time unpredictably. In this thesis, it is assumed that the change of spot prices over time is independent of the job's arrival of a user [71], [76]. At the i-th allocation update of j, when a user bids some price for si_j^i spot instances, without considering the case where the spot instances of j is terminated by a user itself, the period in which j can utilize spot instances is a random variable and we assume that the expected time for which j could utilize spot instances is $\beta \cdot Len$ where $\beta \in [0, 1]$. Finally, Table 7.1 summarizes the main notation of this part of the thesis.

Chapter 8

The Design of Near-Optimal Policies

To become an academic expert takes years of studying. $\cdots\cdots$ They use case studies and observation to understand a subject.

Simon Sinek

In this chapter, we propose a theoretical framework to design (near-)optimal parametric policies, aiming at realizing Principles 7.4.1 and 7.4.2. Facing diverse users, the proposed policies should have good adaptability against the unknown statistics of the spot prices and each individual user's job characteristics; then, by applying the online learning technique, the best configuration parameter that corresponds to each user could be inferred to minimize its cost of processing jobs.

Upon arrival of a job j, the scheduler first considers the allocation of self-owned instances to it, aiming to realize the two goals in Principle 7.4.1. Next, as described in Chapter 7.3.1, the allocation of spot and on-demand instances is updated on an hourly basis.

8.1 Self-owned Instances

8.1.1 Challenge

We first show the challenges in cost-effectively utilizing self-owned instances by an example. Let $N(t)$ denote the number of self-owned instances that are currently idle at a slot t; let $m_{t_1}(t_2) = \min\{N(t_1), \cdots, N(t_2)\}$, where $t_1 \leq t_2$, and it represents the maximum number of self-owned instances idle at every slot in $[t_1, t_2]$. **An intuitive policy** would be, whenever a job j arrives, to allocate as many self-owned instances to j to make self-owned instances fully utilized, i.e.,

$$r_j = \min\{m_{a_j}(a_j + d_j - 1), z_j/d_j\}. \tag{8.1}$$

However, this intuitive policy may not maximize the opportunity that all jobs have to utilize spot instances as illustrated in the following example.

There are two self-owned instance available, and two jobs whose haved the same arrival time, relative deadline of 2 hours and parallelism bound of 4. Jobs 1 and 2 have a size of $4\times Len$ and $6\times Len$, respectively. It is expected that a job can utilize spot instances for $\beta = \frac{1}{2}$ hour ($\beta \cdot Len$ slots) at every allocation update. In Fig. 8.1, the green, blue and yellow areas denote the workload respectively processed by spot, self-owned and on-demand instances. Using the policy (8.1), the whole process of allocating instances is illustrated in Fig. 8.1 (left), where the user has to utilize two on-demand instances for 0.5 hour; however, it is not necessary to purchase more expensive on-demand instances if the allocation process is like Fig. 8.1 (right). In Fig. 8.1 (left), the cost of completing jobs 1 and 2 is $2 \cdot p$ while it is zero in Fig. 8.1; here, on-demand instances are charged on an hourly basis, and the fee of utilizing spot instances is zero when they are terminated by the cloud.

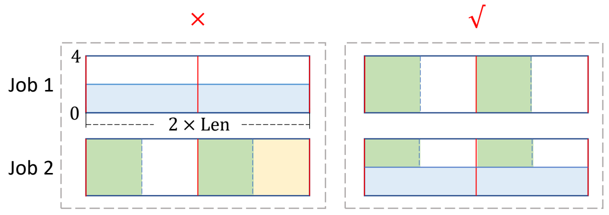

Figure 8.1 – The Challenge in Cost-Effectively Utilizing Self-owned Instances.

The above example reveals some challenges in designing cost-effective policies for allocating self-owned instances. For example, the policy should have the ability of (i) identifying the subset of jobs that can be expected to be completed by utilizing spot instances alone even if they are not allocated any self-owned instance, e.g., the job 1 in Fig. 8.1 (right), and (ii) properly allocating self-owned instances to the rest of jobs, when self-owned instances are inadequate. All in all, our aim is to realize Principle 7.4.1.

8.1.2 Policy Design

In the following, we propose a policy that has the abilities described above. In the subsequent analysis, the issue of rounding the allocations of a job to integers is ignored temporarily for simplicity; in reality, we could round the allocations up to integers, which does not affect the related conclusions much as shown by the analysis.

Recall the meaning of β in Chapter 7.4. For every job j, we will go to find a function $g_j(x) \in [0, \frac{z_j}{d_j}]$ that satisfies the following properties where $\frac{z_j}{d_j} \leq \delta_j$:

***Property* 8.1.1.** *$g_j(x)$ is non-increasing as x increases in $[0, 1)$.*

***Property* 8.1.2.** *$g_j(\beta)$ is the minimum number such that when a job j is assigned r_j self-owned instances in $[a_j, a_j + d_j - 1]$ where $r_j \geq g_j(\beta)$, it could be expected that*

— *job j could be completed by its deadline by utilizing spot instances alone if $\delta_j - r_j$ spot instances are bid for at every allocation update of j, where no on-demand instances is acquired.*

The value of $g_j(\beta)$ is an indicator of the capability that j has such that it can be completed by utilizing spot instances alone. By Property E.2, if $g_j(\beta) \leq 0$, it is expected that no self-owned or on-demand instances is needed in order to complete j and such jobs have strong capability to feed themselves with spot instances. Otherwise, $g_j(\beta)$ self-owned instances are needed, or j has to consume some amount of expensive on-demand instances in order to be completed by its deadline; for a job j, the larger the value of $g_j(\beta)$, the weaker its capability to feed itself with spot instances.

Let $\kappa_0 = \lceil \frac{d_j}{Len} \rceil - 1$, and we set

$$\overline{r}_j(x) = \begin{cases} r'_j(x) & \text{if } d_j - \kappa_0 \cdot Len > x \cdot Len, \\ r''_j(x) & \text{if } d_j - \kappa_0 \cdot Len \leq x \cdot Len, \end{cases}$$

where

$$r'_j(x) = \delta_j - \frac{d_j \cdot \delta_j - z_j}{d_j - (\kappa_0 + 1) \cdot Len \cdot x},$$

and

$$r''_j(x) = \begin{cases} 0 & \text{if } \kappa_0 = 0, \\ \delta_j - \frac{d_j \cdot \delta_j - z_j}{(1-x) \cdot \kappa_0 \cdot Len} & \text{if } \kappa_0 \geq 1. \end{cases}$$

We further set

$$g_j(x) = \max\left\{\overline{r}_j(x), 0\right\}. \tag{8.2}$$

When $x = 0$, $g_j(x) = \max\{r'_j(x), 0\} = \frac{z_j}{d_j}$. When $x \to 1$, $g_j(x) = \max\{r''_j(x), 0\}$ and we have that (i) if $\kappa_0 = 0$, $g_j(x) = 0$, (ii) if $\kappa_0 \geq 1$ and $d_j \cdot \delta_j = z_j$, $g_j(x) = \frac{z_j}{d_j}$, and (iii) if $\kappa_0 \geq 1$ and $d_j \cdot \delta_j > z_j$, $g_j(x) = 0$ since $r''_j(x) \to -\infty$. Now, we proceed to show that the particular $g_j(x)$ in (8.2) satisfies Properties E.2 and E.1.

Proposition 17. *The function $g_j(x)$ in (8.2) satisfies Property E.2.*

Proof. Assume that a job j is allocated r_j self-owned instances in $[a_j, a_j + d_j - 1]$. At each of the first κ_0 allocations of j, the expected time of utilizing spot instances is $\beta \cdot Len$. If a job can be expected to be completed by the deadline by totally utilizing spot instances after the allocation of self-owned instances, we have that (i) it could be expected that the workload processed by self-owned instances plus the workload processed by spot instances at every allocation of j is no less than z_j, and (ii) after the allocation of self-owned instances, the allocation of spot and on-demand instances is always in the first phase as described in Chapter 7.3, i.e., the allocation is updated every hour where only spot instances are bid for.

Now, we analyze two cases. The first one is $d_j - \kappa_0 \cdot Len > \beta \cdot Len$. In this case, at the (κ_0+1)-th allocation of j, the expected time of utilizing spot instances is $\beta \cdot Len$; then, it is expected that

$$r_j \cdot d_j + (\kappa_0 + 1) \cdot (\delta_j - r_j) \cdot Len \cdot \beta \geq z_j.$$

This leads to that $r_j \geq r'_j(\beta)$. The second case is $d_j - \kappa_0 \cdot Len \leq \beta \cdot Len$. In this case, at the (κ_0+1)-th allocation of j, the expected time of utilizing spot instances is $\min\{\beta \cdot Len, d_j - \kappa_0 \cdot Len\} = d_j - \kappa_0 \cdot Len$; then, it is expected that

$$r_j \cdot d_j + \kappa_0 \cdot (\delta_j - r_j) \cdot Len \cdot \beta + (d_j - \kappa_0 \cdot Len) \cdot (\delta_j - r_j) \geq z_j.$$

This leads to that $r_j \geq r''_j(\beta)$. As a summary of our analysis of both cases, the proposition holds. □

Proposition 18. *The function $g_j(x)$ in (8.2) satisfies Property E.1.*

Proof. When $x \in [0, \frac{d_j}{Len} - \kappa_0)$, $g_j(x) = \max\{r'_j(x), 0\}$; since $d_j \cdot \delta_j - z_j \geq 0$ and $(\kappa_0+1) \cdot Len > 0$, $r'_j(x)$ is a non-increasing function and so is $g_j(x)$. Similarly, when $x \in [\frac{d_j}{Len} - \kappa_0, 1)$, $g_j(x) = \max\{r''_j(x), 0\}$ is also non-increasing. In the rest of this proof, if suffices to show $g_j(x_1) \geq g_j(x_2)$ when $0 \leq x_1 < \frac{d_j}{Len} - \kappa_0 \leq x_2 < 1$. Given a job j, if $\kappa_0 = 0$, we have $g_j(x_1) \geq 0 = g_j(x_2)$. If $\kappa_0 \geq 1$ and $d_j \cdot \delta_j = z_j$, we have $g_j(x_1) = \delta_j = g_j(x_2)$. If $\kappa_0 \geq 1$ and $d_j \cdot \delta_j > z_j$, our analysis proceeds as follows. To prove $g_j(x_1) \geq g_j(x_2)$, it suffices to show $r''_j(x_2) \leq r'_j(x_1)$; the function $r''_j(x)$ itself is non-increasing when $x \in [0,1)$, and we have $r''_j(x_2) \leq r''_j(x_1)$. Hence, to prove $r''_j(x_2) \leq r'_j(x_1)$, it suffices to prove $r''_j(x_1) \leq r'_j(x_1)$, which can be proved by showing $A = (1-x_1) \cdot \kappa_0 \cdot Len \leq d_j - (\kappa_0+1) \cdot Len \cdot x_1 = B$. Since $x_1 \in [0, \frac{d_j}{Len} - \kappa_0)$, we have

$$B - A = d_j - (\kappa_0 + x_1) \cdot Len > 0.$$

Finally, the proposition holds. □

In this part of the thesis, we consider a set of jobs $\mathcal{T}$ that arrive over time and can have diverse characteristics. When x ranges in $[0,1)$, we illustrate the function $g_j(x)$ in Fig. 8.2 where $z_j = 240$, $L = 5$, $\delta_j = 20$, and $Len = 12$. The job's minimum execution time is $\frac{z_j}{\delta_j} = Len$ where j is assigned δ_j instances throughout its execution. The job's deadline reflects its ability to utilize spot instances and in Fig. 8.2 the solid curves from left to right represent $g_j(x)$ where d_j is respectively 5, 3, 2.1, 1.47, 1.25, 1.11, and 1.02 times Len: under the same x, the larger the deadline, the smaller the value of $g_j(x)$. Given z_j, δ_j and d_j, we can see in Fig. 8.2 that the function $g_j(x)$ is non-increasing as x ranges in $[0, 1)$.

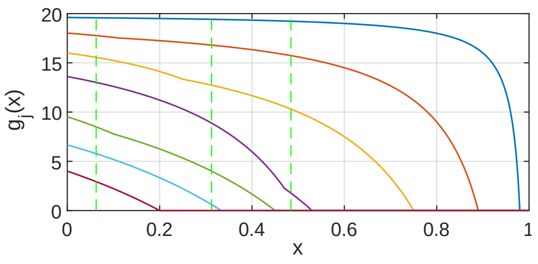

Figure 8.2 – As x ranges in $[0, 1)$, the function $g_j(x)$ for jobs respectively with different flexibility to utilize spot instances.

Proposed Policy. Based on Propositions 17 and 18, we propose the following policy for allocating self-owned instances. Upon arrival of every job j, it is allocated $r_j(\beta_0)$ self-owned instances where

$$r_j(\beta_0) = \min\{g_j(\beta_0), m_t(a_j + d_j - 1)\}, \tag{8.3}$$

where $\beta_0 \in [0, 1)$ is a parameter to be learned.

This policy achieves more cost-effective resource allocation as illustrated in Fig. 8.1 (right) where β_0 is set to $\beta = \frac{1}{2}$. Furthermore, this policy is also adaptive. For example, given another user who owns more instances (e.g., 5 instances), β_0 can be set to a value $< \beta$ (e.g., 0); then, both jobs are allocated more self-owned instances: $r_1 = 2$, and $r_2 = 3$. As a result, self-owned instances are fully utilized and there is no need purchasing spot or on-demand instances.

8.1.3 Explanation

Now, we further explain that the policy (8.3) has desired properties to realize Principle 7.4.1, which will also be validated by the simulations.

The allocations of self-owned instances to all jobs are based on the same function (8.3) whose value depends on a single parameter β_0. Together with Properties E.2 and E.1, the power of the proposed policy can be achieved by setting β_0 to a value properly small in $[0, 1)$. Now, we explain this.

High Utilization. As illustrated in Fig. 8.2, the function $g_j(x)$ is non-increasing; no matter how many self-owned instances a user possesses, a high utilization of them is achieved after

— we set β_0 to a small enough value in $[0, 1)$.

This is because every arriving job will be assigned a large number of self-owned instances when β_0 is small, as illustrated in Fig. 8.3.

Fair Allocation. Fair allocation means that the allocations of self-owned instances among jobs need to be balanced according to their capabilities of utilizing spot instances. Fair allocation avoids ignoring the difference among jobs and

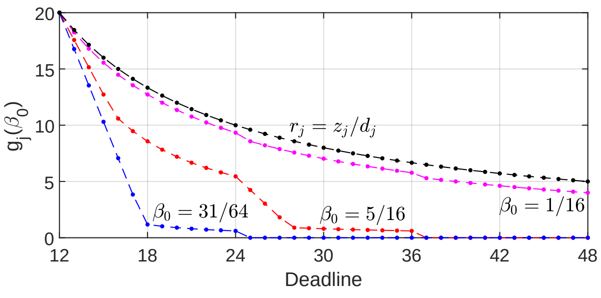

Figure 8.3 – As the (relative) deadline d_j increases from 12 to 48, the function $g_j(\beta_0)$ decreases respectively under $\beta_0 = \frac{31}{64}, \frac{5}{16}, \frac{1}{16}$, where $z_j = 240$, $\delta_j = 20$, and $Len = 12$.

treating them equally where a policy like (8.1) is used; together with Property E.2, the latter can lead to that "rich" jobs (i.e., jobs with strong capabilities where $g_j(\beta)$ is small) are consuming unnecessary self-owned instances, i.e.,

— $r_j > g_j(\beta)$, where r_j denotes the number of self-owned instances allocated to a job; the job's remaining $z_j - r_j \cdot d_j$ workload is expected to be processed by spot instances alone;

whereas the others (with large $g_j(\beta)$) are allocated poorly and still starving for more self-owned instances, i.e.,

— $r_j < g_j(\beta)$; here, on-demand instances are expected to be consumed.

Indiscriminate allocations of instances to jobs do harm to the process of achieving the capacity that jobs have for utilizing spot instances, causing unnecessary consumption of more on-demand instances and a higher cost of completing all jobs. In particular, for every rich job, only $g_j(\beta)$ self-owned instances are needed to complete its remaining workload without on-demand instances; the saved $r_j - g_j(\beta)$ self-owned instances can be used for those poorly allocated jobs so as to reduce their consumption of on-demand instances, which improves the cost-efficiency of instance utilization.

Now, we explain that the proposed policy achieves fair allocation by properly setting the value of β_0. The cost-optimal β_0, denoted by β_0^*, depends on the statistics of jobs and the amount of self-owned instances available; the online learning technique will be used subsequently in Chapter 8.4 to infer β_0^*. When $\beta_0^* = 0$, self-owned instances themselves are enough to complete all jobs by their deadlines where $g_j(\beta_0) = \frac{z_j}{d_j}$.

When there are adequate self-owned instances such that $\beta_0^* \in (0, \beta]$, every arriving job j will be allocated $\geq g_j(\beta)$ self-owned instances whenever the amount of idle self-owned is large (i.e., $m_{a_j}(a_j + d_j - 1) \geq g_j(\beta_0)$), according to the policy (8.3); this is illustrated in Fig. 8.3 where $\beta = \frac{5}{16}$ and $\beta_0^* = \frac{1}{16}$. Afterwards, the job j is expected to be completed by utilizing spot instances alone. No job will be allocated $< g_j(\beta)$ self-owned instances whenever possible, and fair allocation is achieved. Furthermore, the arriving jobs are allocated on a first come first served basis and we note that β_0 should be properly small but cannot be set to a value too small. If β_0 is too small, jobs that arrive earlier might consume too many self-owned instances and then the jobs that arrive late have less opportunity to get $\geq g_j(\beta)$ self-owned instances subject to the availability of these instances (i.e., the value of $m_{a_j}(a_j + d_j - 1)$).

When self-owned instances are deficient such that $\beta_0^* \in (\beta, 1)$, every arriving job will be allocated $< g_j(\beta)$ self-owned instances; this is illustrated in Fig. 8.3 where $\beta = \frac{5}{16}$ and $\beta_0^* = \frac{31}{64}$. Afterwards, the job j is expected to have to utilizing some amount of on-demand instances to meet its deadline. No job will be allocated $> g_j(\beta)$ self-owned instances, achieving fair allocation among jobs: if there exists such allocation, a waste of self-owned instances is caused since we can reduce this allocation to $g_j(\beta)$ and allocate these saved instances to other jobs to reduce the consumption of on-demand instances.

8.2 Spot and On-demand Instances

As described in Chapter 7.3.1, the instance allocation process is divided into two phases. Now, we analyze the expected optimal strategy to utilize spot instances.

8.2.1 First phase

In the first phase, the allocation of j is updated per hour and there is flexibility for j to utilize spot instances. Now, we analyze the expected optimal policy in the first phase. One of the following two cases will happen: (i) the job j is completed in the first phase, and (ii) At some allocation update of j (i.e., the i_j-th allocation in Chapter 7.3.1), after spot instances are terminated by the cloud, there is no flexibility for j to utilize spot instances.

As seen later, the value of β will be estimated by the online learning technique. If the previous allocation of self-owned instances r_j is $\geq g_j(\beta)$, it is expected that the first case will happen; then, by Property E.2, we conclude that

Proposition 19. *An expected optimal strategy is to bid for $\delta_j - r_j$ spot instances at every allocation of j.*

Next, we analyze the optimal strategy when the second case happens. Job j is allocated r_j self-owned instances at every $t \in [a_j, a_j + d_j - 1]$; afterwards, it can be viewed as a new job with a workload $z_j - \delta_j \cdot d_j$ and a parallelism bound $\delta_j - r_j$, as described in Chapter 7.3.2. So, without loss of generality, we just analyze the optimal strategy in the case where a job j is completed by utilizing on-demand and spot instances alone.

Our decision variables are o_j^i and si_j^i where $o_j^i + si_j^i = \delta_j$. Let κ_1 denote the total number of allocation updates in the first phase where j has flexibility for spot instances; let $\kappa_0 = \lceil d_j / Len \rceil$ denoting the maximum possible number of allocation updates of j and we have

$$\kappa_1 \leq \kappa_0. \tag{8.4}$$

At the i-th allocation of j where $i \in [1, \kappa_1]$, it is expected that the workloads processed by spot and on-demand instances are respectively $(\delta_j - o_j^i) \cdot Len \cdot \beta$ and $o_j^i \cdot Len$. By Definition 7, j has flexibility to utilize unstable spot instances at the κ_1-th allocation, i.e.,

$$s_j^{\kappa_1} = \frac{\delta_j \cdot (d_j - (\kappa_1 - 1) \cdot Len)}{z_j - \sum_{i=1}^{\kappa_1 - 1} \left(o_j^i \cdot Len + (\delta_j - o_j^i) \cdot Len \cdot \beta\right)} \geq 1,$$

and has no flexibility to utilize spot instances at the $(\kappa_1 + 1)$-th allocation, i.e.,

$$s_j^{\kappa_1 + 1} = \frac{\delta_j \cdot (d_j - \kappa_1 \cdot Len)}{z_j - \sum_{i=1}^{\kappa_1} \left(o_j^i \cdot Len + (\delta_j - o_j^i) \cdot Len \cdot \beta\right)} < 1.$$

They are respectively equivalent to the following relations:

$$\sum\nolimits_{i=1}^{\kappa_1 - 1} (\delta_j - o_j^i) \cdot Len \cdot (1 - \beta) \leq d_j \cdot \delta_j - z_j, \tag{8.5}$$

$$\sum\nolimits_{i=1}^{\kappa_1} (\delta_j - o_j^i) \cdot Len \cdot (1 - \beta) > d_j \cdot \delta_j - z_j. \tag{8.6}$$

For the condition that $s_j^{\kappa_1+1} < 1$, a special case is $\kappa_1 = \kappa_0$ where this condition holds trivially since $d_j - \kappa_1 \cdot Len \leq 0$; since $s_j^{\kappa_1} \geq 1$, the κ_1-th allocation of j is still in the first phase. In this subsection, our objective is to maximize the total workload processed by spot instances at the first κ_1 allocations, i.e.,

$$\text{maximize} \ \sum_{i=1}^{\kappa_1} (\delta_j - o_j^i) \cdot Len \cdot \beta, \tag{8.7}$$

subject to the constraints (8.4), (8.5), (8.6), and the constraint that o_j^i is an integer in $[0, \delta_j]$. Our decision variables are $o_j^1, \cdots, o_j^{\kappa_1}$.

Now, we give an optimal solution to (8.7).

Proposition 20. *An solution to (8.7) is optimal if it is of the following form: (i)* $\sum_{i=1}^{\kappa_1 - 1} (\delta_j - o_j^i) = \min\{\nu(z_j, d_j), (\kappa_0 - 1) \cdot \delta_j\}$, *and (ii)* $o_j^{\kappa_1} = 0$, *where*

$$\nu(z_j, d_j) = \left\lfloor \frac{d_j \cdot \delta_j - z_j}{Len \cdot (1 - \beta)} \right\rfloor.$$

Proof. Firstly, we prove by contradiction that the optimal value of $o_j^{\kappa_1}$ is 0. Assume that $\hat{o}_j^1, \cdots, \hat{o}_j^{\kappa_1}$ are an optimal solution to (8.7) where $\hat{o}_j^{\kappa} \geq 1$. The constraint (8.5) has no effect on the value of $o_j^{\kappa_1}$. We can reduce the value of $\hat{o}_j^{\kappa_1}$ to 0; such reduction can still guarantee that (8.6) is satisfied, and $\hat{o}_j^1, \cdots, \hat{o}_j^{\kappa_1 - 1}, o_j^{\kappa_1} = 0$ are a feasible solution to (8.7) under which (8.7) achieves a higher value, which contradicts that $\hat{o}_j^1, \cdots, \hat{o}_j^{\kappa_1}$ are an optimal solution to (8.7). Secondly, when $o_j^{\kappa_1} = 0$, the objective function (8.7) equals $(\sum_{i=1}^{\kappa_1 - 1} (\delta_j - o_j^i) + \delta_j) \cdot Len \cdot \beta$. Under constraint (8.5), $\sum_{i=1}^{\kappa_1 - 1} (\delta_j - o_j^i) \leq \frac{d_j \cdot \delta_j - z_j}{Len \cdot (1 - \beta)}$. Since $o_j^1, \cdots, o_j^{\kappa_1 - 1}$ are integers, the maximum possible value of $\sum_{i=1}^{\kappa_1-1}(\delta_j - o_j^i)$ is $\nu(z_j, d_j)$. On the other hand, since $\delta_j - o_j^i \leq \delta_j$, the constraint (8.4) indicates that $\sum_{i=1}^{\kappa_1-1}(\delta_j - o_j^i) \leq (\kappa_0 - 1)\cdot \delta_j$. Hence, the maximum possible value of $\sum_{i=1}^{\kappa_1-1}(\delta_j - o_j^i)$ is $\min\{\nu(z_j, d_j), (\kappa_0 - 1)\cdot\delta_j\}$. Now, we further show it is feasible. If $(\kappa_0 - 1)\cdot\delta_j \leq \nu(z_j, d_j)$, $\sum_{i=1}^{\kappa_1-1}(\delta_j - o_j^i) = (\kappa_0 - 1)\cdot\delta_j$ which leads to $\kappa_0 - 1 \leq \kappa_1 - 1$; to satisfy (8.4), we have $\kappa_1 = \kappa_0$. Then, constraint (8.6) holds trivially and constraint (8.5) is also satisfied. If $(\kappa_0 - 1)\cdot\delta_j > \nu(z_j, d_j)$, $\sum_{i=1}^{\kappa_1-1}(\delta_j - o_j^i) = \nu(z_j, d_j)$; in this case, we have $\kappa_1 - 1 \leq \kappa_0 - 1$. Furthermore, we also have $\nu(z_j, d_j) + \delta_j > \frac{d_j\cdot\delta_j - z_j}{Len\cdot(1-\beta)}$ and (8.6) is satisfied. Finally, the proposition holds. $\square$

Proposition 20 indicates the maximum number of spot instances that can be bid for in the first phase, i.e., the maximum value of $\sum_{i=1}^{\kappa_1}(\delta_j - o_j^i)$. As a corollary of Proposition 20, we conclude that

Proposition 21. *Given a job j, the expected maximum workload that can be processed by spot instances is*

$$(\min\{\nu(z_j, d_j), (\kappa_0 - 1)\cdot\delta_j\} + \delta_j)\cdot Len\cdot\beta.$$

Proposition 20 also implies an expected optimal strategy for spot instances.

Proposition 22. *Let $\kappa_2(z_j, d_j) = \lfloor\frac{\nu(z_j,d_j)}{\delta_j}\rfloor$ and $\kappa_3 = \frac{\nu(z_j,d_j)}{\delta_j}$. To maximize the total workload processed by spot instances, if $(\kappa_0 - 1)\cdot\delta_j \leq \nu(z_j, d_j)$, we can set $\kappa_1 = \kappa_0$ and* an expected optimal strategy *is to*

— *bid for δ_j spot instances at each allocation update of j.*

If $\nu(z_j, d_j) < (\kappa_0 - 1)\cdot\delta_j$, in the case that $\kappa_2(z_j, d_j) = \kappa_3$, we can set $\kappa_1 = \kappa_2(z_j, d_j) + 1$ and an expected optimal strategy *is to*

— *bid for δ_j spot instances at each of the first κ_1 allocations of j, i.e., $o_j^1 = \cdots = o_j^{\kappa_1} = \delta_j$;*

in the case that $\kappa_2(z_j, d_j) < \kappa_3$, we can set $\kappa_1 = \kappa_2(z_j, d_j) + 2$ and an expected optimal strategy *is to*

— *bid for δ_j spot instances at the 1st, $\cdots$, $(\kappa_1 - 2)$-th, κ_1-th allocations of j, i.e., $o_j^1 = \cdots = o_j^{\kappa_1-2} = o_j^{\kappa_1} = \delta_j$,*

— *bid for $\nu(z_j, d_j) - \kappa_2(z_j, d_j)\cdot\delta_j$ spot instances at the $(\kappa_1 - 1)$-th allocation of j, i.e., $o_j^{\kappa_1-1} = \nu(z_j, d_j) - \kappa_2(z_j, d_j)\cdot\delta_j$.*

Proof. We can check that when the strategy of utilizing spot instances is as above, $o_j^1, \cdots, o_j^{\kappa_1}$ are of the form in Proposition 20; hence, it is optimal. $\square$

We illustrate proposition 22 in Fig. 8.4 where the orange and green areas denote the workload processed respectively by spot and on-demand instances; in the grey areas, no workload of j is processed. We assume that $\beta = \frac{1}{2}$ and $L = 5$ where $Len = 12$; job j has $d_j = 42$ (3.5 hours), $z_j = 122$ and $\delta_j = 4$. Here, we have $\nu(z_j, d_j) = 7$ and $\kappa_2(z_j, d_j) = 1$. From the left to the right, the first four subfigures illustrate the expected optimal allocation. At the 1st allocation

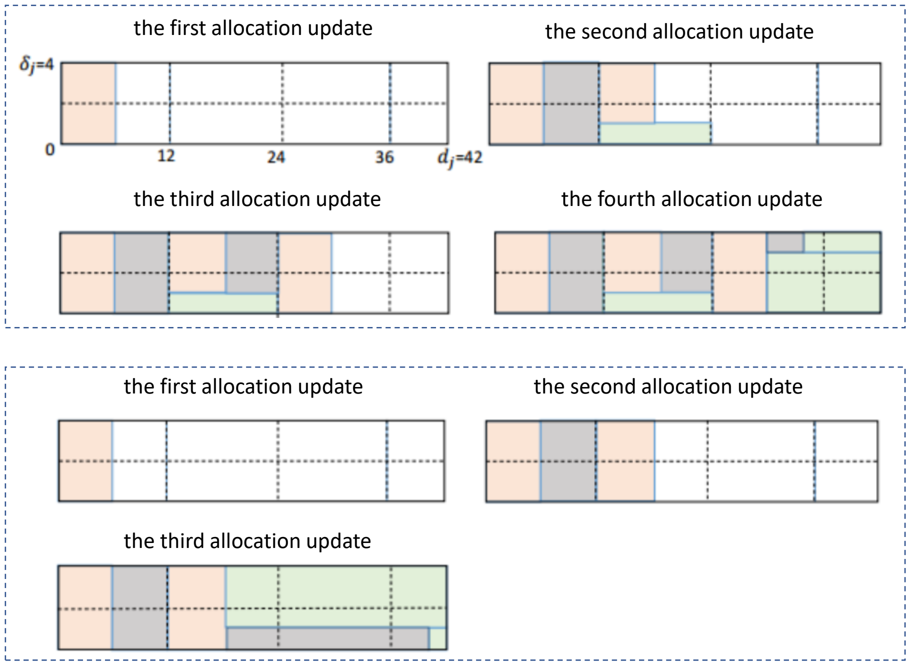
the first allocation update
δ_j=4
0
12
24
36
d_j=42
the second allocation update
the third allocation update
the fourth allocation update
the first allocation update
the second allocation update
the third allocation update

Figure 8.4 – Illustration of Proposition 22 in the case that $\nu(z_j, d_j) < (\kappa_0 - 1)\cdot\delta_j$ and $\kappa_2(z_j, d_j) < \kappa_3$.

of j, $\delta_j = 4$ spot instances are bid for and the expected execution time of spot instances is $\beta\cdot Len = 6$. At the 2-th allocation of j, $(\nu(z_j, d_j) - \delta_j\cdot\kappa_2(z_j, d_j)) = 3$ spot instances are bid for and one on-demand instance is purchased. So far, $\nu(z_j, d_j) = 7$ spot instances have been bid for. At the 3rd allocation of j, δ_j spot instances are bid for and after the execution of spot instances, j has no flexibility to utilize spot instances and it turns to totally utilize on-demand instances as illustrated by the fourth subfigure. In contrast, we also use the last three subfigures to illustrate an intuitive way to bid for spot instances where δ_j spot instances are bid for at every allocation of j when it has flexibility to utilize spot instances. After the execution of spot instances at the 2nd allocation of j, it does not have such flexibility and has to turn to utilize on-demand instances since $s_j^3 < 1$.

Based on Proposition 22, we propose Algorithm 12 to dynamically determine the numbers of on-demand and spot instances allocated to j at every i-th allocation update when there is flexibility to utilize spot instances. At every allocation of j that occurs at slot t, the remaining workload of j to be processed could be viewed as a new job with the arrival time t, workload z_j', parallelism bound δ_j, and relative deadline $a_j + d_j - t$; we always use Proposition 22 to determine the first allocation of this new job whose arrival time is t.

Algorithm 12: Proportion(j, β, b)

```
/* At the $i$-th allocation of $j$, its remaining workload is viewed as a new
   job with an arrival time $t$, and a relative deadline $a_j + d_j - t$      */
1  $\kappa_0(t) \leftarrow \left\lceil \frac{a_j+d_j-t}{Len} \right\rceil$
/* the case $(\kappa_0 - 1) \cdot \delta_j \leq \nu(z_j, d_j)$ in Proposition 22   */
2  if $(\kappa_0(t) - 1) \cdot \delta_j \leq \nu(z_j, a_j + d_j - t)$ then
3      $si_j^i \leftarrow \delta_j$,  $o_j^i \leftarrow 0$;
4  else
       /* both cases $\kappa_2(z_j, d_j) = \kappa_3$ and $\kappa_2(z_j, d_j) < \kappa_3$ where $\kappa_2(z_j, d_j) \geq 1$   */
5      if $\kappa_2(z'_j, a_j + d_j - t) \geq 1$ then
6          $si_j^i \leftarrow \delta_j$,  $o_j^i \leftarrow 0$;
       /* the case $\kappa_2(z_j, d_j) < \kappa_3$ where $\kappa_2(z_j, d_j) = 0$   */
7      if $\kappa_2(z'_j, a_j + d_j - t) = 0 \wedge \nu(z_j, a_j + d_j - t) > 0$ then
8          $si_j^i \leftarrow \nu(z_j, a_j + d_j - t)$,  $o_j^i \leftarrow \delta_j - si_j^i$;
       /* the case $\kappa_2(z_j, d_j) = \kappa_3 = 0$   */
9      if $\nu(z_j, a_j + d_j - t) = 0$ then
10         $si_j^i \leftarrow \delta_j$,  $o_j^i \leftarrow 0$;
11 $b_j^i \leftarrow b$;
12 at the $i$-th allocation update, bid a price $b_j^i$ for $si_j^i$ spot instances;
```

8.2.2 Second phase

As described in Chapter 7.3.1, once spot instances get lost at every allocation of j, the scheduler uses Definition 7 to check the flexibility to utilize spot instances. At the i_j-th allocation, when spot instances get lost at the beginning of some slot t'_1, there is no such flexibility; then, the instance allocation enters the second phase where only on-demand instances are utilized. Now, we analyze their optimal utilization.

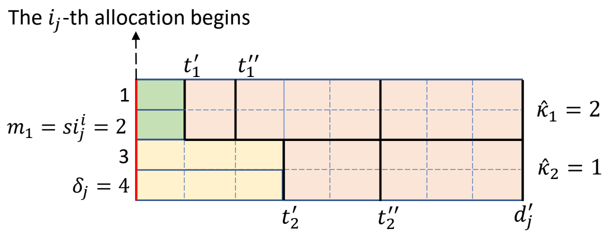

Figure 8.5 – The second phase of allocation where $i = i_j$: the orange area denotes the available space in the second phase; the green and yellow areas denote the workload processed at the i_j-th allocation by spot instances and on-demand instances that are utilized for an hour.

As shown in Algorithm 13, at every allocation of j in the first phase (including the i_j-th allocation), the number of on-demand instances allocated to j is either 0 (see lines 3, 6, 10) or > 0 (see line 8). Let $t_2' = a_j + i_j \cdot Len$, $d_j' = a_j + d_j - 1$, and we define two parameters that represent the maximum multiple of an hour (containing Len slots) respectively in time intervals $[t_1', d_j']$ and $[t_2', d_j']$:

$$\hat{\kappa}_1 = \left\lfloor \frac{d_j' - (t_1' - 1)}{Len} \right\rfloor, \text{ and } \hat{\kappa}_2 = \left\lfloor \frac{d_j' - t_2' + 1}{Len} \right\rfloor;$$

Let $t_i'' = d_j' - \hat{\kappa}_i \cdot Len + 1$ ($i \in \{1, 2\}$), and after deducting $\hat{\kappa}_1$ and $\hat{\kappa}_2$ hours respectively from the two intervals, the numbers of remaining slots in $[t_1', t_1'' - 1]$ and $[t_2', t_2'' - 1]$ are denoted by ϕ_1 and ϕ_2:

$$\phi_1 = t_1'' - t_1', \text{ and } \phi_2 = t_2'' - t_2',$$

where $0 \leq \phi_1, \phi_2 < Len$. The related notation is also illustrated in Fig. 8.5. Let

$$m_0 = si_j^i \cdot \hat{\kappa}_1 + o_j^i \cdot \hat{\kappa}_2, \;\; m_1 = si_j^i, \;\text{ and } m_2 = o_j^i,$$

where $i = i_j$. In Fig. 8.5, the available space in the second phase is the orange area and m_0 represents the maximum integer of instance hour that can be utilized by j.

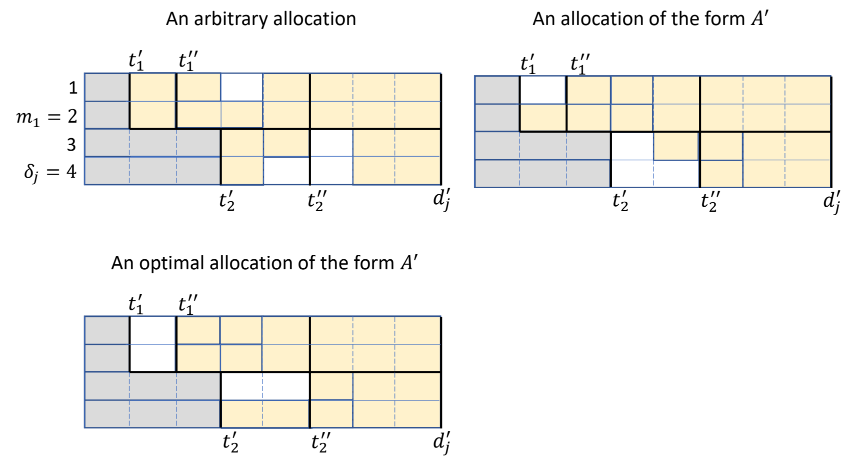

Figure 8.6 – Illustration for Proposition 23: the yellow area denotes the allocation of on-demand instances to j.

Since every on-demand instance is charged on an hourly basis, a cost-optimal strategy in the second phase is to minimize the integer instance hours (i.e., the number of on-demand instances $\times$ the time for which they are utilized). The following conclusion possibly is intuitive although a formal proof is also provided: whenever an instance is purchased for an hour, it should be utilized as long as possible with the space constraint.

Proposition 23. *Let $y = y_0 + y_1 + y_2$ be the minimum such that $y_0 \cdot Len + y_1 \cdot \phi_1 + y_2 \cdot \phi_2 \geq z_j^{i_j+1}$ subject to y_0, y_1, y_2 are non-negative integers and $y_0 \in [0, m_0]$, $y_1 \in [0, m_1]$, $y_2 \in [0, m_2]$. In the second phase, a cost-optimal strategy is to*

— *purchase on-demand instances for y instance hours.*

Proof. Let us consider an arbitrary allocation of on-demand instances to process the remaining $z_j^{i_j+1}$ workload, denoted by $\mathcal{A}$, also illustrated in the 1st subfigure of Fig. 8.6. These workload will be processed on δ_j instances, and let x_h denote the total workload processed at the h-th instance where

$$\sum\nolimits_{h=1}^{\delta_j} x_h \geq z_j^{i_j+1}, \tag{8.8}$$

$$\begin{aligned} x_1, \cdots, x_{m_1} &\in [0, d_j' - t_1' + 1], \\ x_{m_1+1}, \cdots, x_{\delta_j} &\in [0, d_j' - t_2' + 1]. \end{aligned} \tag{8.9}$$

The allocation $\mathcal{A}$ can be transformed into an allocation $\mathcal{A}'$ with the following form without increasing the total cost of utilizing instances: the x_h workload of the h-th instance is processed from the deadline d_j' towards earlier slots, i.e., in $[d_j' - x_h + 1, d_j']$, which is illustrated in the 2nd subfigure of Fig. 8.6. Hence, in the following, we only need to show the cost-optimal strategy of utilizing instances when the allocation is of the form $\mathcal{A}'$.

As illustrated in Fig. 8.5, let $\hat{\mathcal{I}}_1 = [t_1', d_j']$ and $\hat{\mathcal{I}}_2 = [t_2', d_j']$. From d_j' towards earlier slots in $\hat{\mathcal{I}}_1$ (resp. in $\hat{\mathcal{I}}_2$), let every Len slots constitute a time interval, i.e., $\mathcal{I}_i = [d_j' + 1 - i \cdot Len, d_j' - (i-1) \cdot Len]$; for $\hat{\mathcal{I}}_1$ the last interval is $\mathcal{I}_{\hat{\kappa}_1+1} = [t_1', t_1'' - 1]$ (resp. for $\hat{\mathcal{I}}_2$ the last is $\mathcal{I}_{\hat{\kappa}_2+1} = [t_2', t_2'' - 1]$). Now, we describe the cost structure when the allocation of j is of the form $\mathcal{A}'$. We use $x_{h,i}$ to denote the workload processed by the h-th instance in $\mathcal{I}_i$ where for all $h \in [1, m_1]$,

$$x_{h,1}, \cdots, x_{h,\hat{\kappa}_1} \in [0, Len], \;\; x_{h,\hat{\kappa}_1+1} \in [0, \phi_1], \tag{8.10}$$

and for all $h \in [m_1 + 1, \delta_j]$,

$$x_{h,1}, \cdots, x_{h,\hat{\kappa}_2} \in [0, Len], \;\; x_{h,\hat{\kappa}_2+1} \in [0, \phi_2]. \tag{8.11}$$

Let $\psi_h = \lceil \frac{x_h}{Len} \rceil$; under the allocation form of $\mathcal{A}'$, we have for all $h \in [1, \delta_j]$ that

$$\begin{aligned} & x_{h,1} = \cdots = x_{h,\psi_h - 1} = Len, \\ & x_{h,\psi_h} = x_h - (\psi_h - 1) \cdot Len, \\ & \text{the other } x_{h,i} = 0, \end{aligned} \tag{8.12}$$

and

$$\begin{aligned} x_h &= \sum\nolimits_{i=1}^{\hat{\kappa}_1+1} x_{h,i}, \text{ if } h \in [1, m_1] \\ x_h &= \sum\nolimits_{i=1}^{\hat{\kappa}_2+1} x_{h,i}, \text{ if } h \in [m_1 + 1, \delta_j] \end{aligned} \tag{8.13}$$

where $0 \leq x_{h,\psi_h} < Len$. We define the sign function $sgn(x)$: it equals 1 if $x > 0$ and 0 if $x = 0$. Let

$$y_{h,i} = sgn(x_{h,i}) \in \{0, 1\}, \tag{8.14}$$

and the price of utilizing the h-th instance is p times the sum of all $y_{h,i}$; here, by (8.12), the sum of all $y_{h,i}$ is ψ_h.

The cost minimization problem under the allocation form of $\mathcal{A}'$ is as follows, referred to as **$\mathcal{Q}$-I**:

$$\min \sum_{h=1}^{m_1}\sum_{i=1}^{\hat{\kappa}_1+1} p \cdot y_{h,i} + \sum_{h=m_1+1}^{\delta_j}\sum_{i=1}^{\hat{\kappa}_2+1} p \cdot y_{h,i} \tag{8.15}$$

subject to the constraints (8.8)-(8.14). $\mathcal{Q}$-I corresponds to another optimization problem: its objective function is also (8.15), subject to (8.8), (8.9), (8.13), (8.14), and for all $h \in [1, m_1]$

$$x_{h,1}, \cdots, x_{h,\hat{\kappa}_1} \in \{0, Len\}, \;\; x_{h,\hat{\kappa}_1+1} \in \{0, \phi_1\}, \tag{8.16}$$

and for all $h \in [m_1 + 1, \delta_j]$,

$$x_{h,1}, \cdots, x_{h,\hat{\kappa}_2} \in \{0, Len\}, \;\; x_{h,\hat{\kappa}_2+1} \in \{0, \phi_2\}. \tag{8.17}$$

The above mathematical problem is referred to as **$\mathcal{Q}$-II**. In the following, we prove that (i) any solution to $\mathcal{Q}$-I corresponds to a solution to $\mathcal{Q}$-II and their objective function (8.15) under these two solutions achieves the same value; then, (ii) an optimal solution to $\mathcal{Q}$-II corresponds to a solution to $\mathcal{Q}$-I, and their objective function under these two solutions also achieves the same value. The first point shows that the optimal value of $\mathcal{Q}$-II is a lower bound of the optimal value of $\mathcal{Q}$-I. The second point shows that there is a solution to $\mathcal{Q}$-I under which the value of (8.15) equals the optimal value of $\mathcal{Q}$-II; hence, this solution to $\mathcal{Q}$-I is optimal and we will give such an optimal solution while proving the two points above.

The decision variables of both $\mathcal{Q}$-I and $\mathcal{Q}$-II are the same, i.e., $\{y_{h,i} | h \in [1, m_1], i \in [1, \hat{\kappa}_1 + 1]\} \cup \{y_{h,i} | h \in [m_1 + 1, \delta_j], i \in [1, \hat{\kappa}_2 + 1]\}$. Given a solution to $\mathcal{Q}$-I denoted by Y, we set the decision variables of $\mathcal{Q}$-II to the same values. Now, we show Y is a feasible solution to $\mathcal{Q}$-II. Both in $\mathcal{Q}$-II and $\mathcal{Q}$-I, the same $x_{h,i}$ is set to non-zero and the others are set to zero by (8.14), and the non-zero's $x_{h,i}$ in $\mathcal{Q}$-II is $\geq$ the $x_{h,i}$ in $\mathcal{Q}$-I by (8.10), (8.11), (8.16), and (8.17). Since (8.8) holds in $\mathcal{Q}$-I where the value of x_h is defined in (8.13), we have (8.8) also holds in $\mathcal{Q}$-II. Hence, Y is feasible. Furthermore, $\mathcal{Q}$-I and $\mathcal{Q}$-II have the same objective function (8.15) that achieves the same value under the same Y. This finishes proving the first point above.

Now, we give an optimal solution to $\mathcal{Q}$-II. The physical meaning of $\mathcal{Q}$-II can be explained as follows. There are 3 types of items each with a weight p: (i) $\hat{\kappa}_1 \cdot m_1 + \hat{\kappa}_2 \cdot m_2$ items each with a size Len, (ii) m_1 items each with a size ϕ_1 ($< Len$), and (iii) m_2 items each with a size ϕ_2 ($< Len$); the objective is to select some items such that the total size of chosen items is $\geq z_j^{i_j+1}$ (satisfying (8.8)) while their total weight (i.e., (8.15)) is minimized. Since items have the same weight, an optimal solution is just to select the minimum number of items, e.g., the items with the largest sizes, to exactly satisfy the size requirement; correspondingly, an optimal solution to $\mathcal{Q}$-II is such that the value of $y_{h,i} \in \{0, 1\}$ satisfies

$$\begin{aligned} y_0 &= \sum_{h=1}^{m_1}\sum_{i=1}^{\hat{\kappa}_1} y_{h,i} + \sum_{h=m_1+1}^{\delta_j}\sum_{i=1}^{\hat{\kappa}_2} y_{h,i}, \\ y_1 &= \sum_{h=1}^{m_1} y_{h,\hat{\kappa}_1+1}, \;\; y_2 = \sum_{h=m_1+1}^{\delta_j} y_{h,\hat{\kappa}_2+1}, \end{aligned} \tag{8.18}$$

where y_0, y_1, y_2 are described in Proposition 23. We denote such a solution by OPT_2. Here, we set $x_{h,i}$ to non-zero if $y_{h,i} = 1$ and zero otherwise by (8.14); the particular value of $x_{h,i}$ depends on (8.16) and (8.17), and it determines the value of x_h by (8.13) that can satisfy (8.9); by (8.18), x_h can satisfy (8.8).

Next, we show OPT_2 corresponds to a solution OPT_1 to $\mathcal{Q}_1$-I, and their objective function (8.15) under OPT_1 and OPT_2 achieves the same value. In $\mathcal{Q}$-I, we set the value of x_h to the same value when the solution to $\mathcal{Q}$-II is OPT_2 where the constraints (8.8) and (8.9) in $\mathcal{Q}$-I are naturally satisfied; then, we use (8.12) to obtain feasible $x_{h,i}$ that will also satisfy (8.10) and (8.11); by (8.14), the value of $y_{h,i}$ in $\mathcal{Q}$-I can be set, deriving a feasible solution OPT_1 to $\mathcal{Q}$-I. In both $\mathcal{Q}$-I and $\mathcal{Q}$-II, we have the number of non-zero's $y_{h,i}$ is $\lceil x_h/Len \rceil$; hence, $\mathcal{Q}$-I under OPT_1 and $\mathcal{Q}$-II under OPT_2 achieve the same value. Finally, OPT is an optimal solution to $\mathcal{Q}$-I by the two points above. □

In the proof of Proposition 23, we have given an optimal solution OPT_1 to $\mathcal{Q}$-I; it is a particular cost-optimal allocation of on-demand instances, which is also illustrated in the 3rd subfigure of Fig. 8.6.

8.3 Scheduling Framework

As described above, a general policy is defined by a tuple $\{\beta_0, \beta, b\}$ and determines the amounts of self-owned, spot, and on-demand instances allocated to a job, and the bid price.

The instance allocation process has been described in Chapter 7.3. Based on this, at every slot t, if a job j just arrives or it has arrived before but not been completed yet, we propose a framework, presented in Algorithm 13, to determine the action of allocating instances to j after checking the state of j. Actions are needed in the following three states: (i) t is the arrival time of j, determining the allocation of self-owned instances, (ii), t equals $a_j + (i-1)\cdot Len$ where the i-th allocation update of spot and on-demand instances needs to be done, (iii) the spot instances of j get lost at t where we need to check whether j still has flexibility for spot instances. In Algorithm 13, z'_j denotes the remaining workload of j to be processed after deducting its current allocations from z_j; upon arrival of j, $z'_j = z_j$.

Algorithm 13: Dynalloc

```
Input : the job's current characteristics $\{a_j, d_j, z'_j, \delta_j\}$ where $z'_j$ is still
        $> 0$, and a parameterized policy $\{\beta_0, \beta, b\}$
/* allocate instances at the very beginning of slot $t$ */
1  if $a_j = t$ then
       // upon arrival of $j$, allocate self-owned instances to it
2      set the value of $r_j$ using Equation (8.3);
3      for $\bar{t} \leftarrow a_j$ to $a_j + d_j - 1$ do
4          $r_j(\bar{t}) \leftarrow r_j$;
5  $i \leftarrow \left\lfloor \frac{t-a_j}{Len} \right\rfloor + 1$// used to number the allocation update
6  if $\frac{t-a_j}{Len} = i - 1$ then
       // at the $i$-th allocation of $j$ where it has flexibility for spot
          instances
7      if $r_j \geq g_j(\beta)$ then
           // it is expected that $j$ will be completed by utilizing spot
              instances alone after allocating self-owned instances
8          apply the strategy in Proposition 19 here;
9      else
10         call Algorithm 12;
11 if the spot instances of $j$ get lost at the beginning of slot $t$ then
12     if $\frac{(\delta_j - r_j)\cdot(d_j - Len\cdot i)}{z'_j} < 1$ then
           // $j$ has no flexibility to utilize spot instances at the next
              allocation update by Definition 7
13         apply the strategy in Proposition 23 here;
       // otherwise, $j$ still has the flexibility at the next allocation update
          where $z'_j = z_j^{i+1}$
```

8.4 The Application of Online Learning

Upon arrival of a job j, the allocation process in Algorithm 13 is determined by parameters β_0, β, b. In this subsection, we show how online learning is applied to learn the most cost-effective parameters $\{\beta_0, \beta, b\}$.

The online learning algorithm that we adopt is the one in [6], [68], presented as Algorithm 14, and is also a form of the classic weighted majority algorithm. It runs as follows. There are a set of jobs $\mathcal{J}$ that arrive over time and a set of n parametric policies $\mathcal{P}$ each specified by $\{\beta_0, \beta, b\}$. Let $d = \max_{j\in\mathcal{J}}\{d_j\}$, i.e., the maximum relative deadline of all jobs. Let $\mathcal{J}_t \subseteq \mathcal{J}$ denote all jobs j that arrive at time slot t, i.e., $a_j = t$. There is also an initial distribution over n policies, e.g., a discrete uniform distribution $\{1/n, \cdots, 1/n\}$. Whenever a job $j \in \mathcal{J}_t$ arrives, the algorithm randomly picks a policy from $\mathcal{P}$ according to the distribution and bases the allocation of various instances to j on that policy. In the meantime, when $t > d$, if $\mathcal{J}_{t-d} \neq \emptyset$, since the history of spot prices in the time interval $[a_j - d, a_j - 1]$ has been known, we are enabled to compute the cost of each policy on a job in $\mathcal{J}_{t-d}$. Subsequently, the weight of each policy (i.e., its probability) are updated so that the lower-cost (higher-cost) polices of this job are re-assigned the enlarged (resp. reduced) weights. As more and more jobs are processed and the above process repeats, the most cost-effective policies of $\mathcal{P}$ will be identified gradually, i.e., the ones with the highest weights, well realizing Principles 7.4.1 and 7.4.2 and finally minimizing the total cost of completing all jobs.

Algorithm 14: OptiLearning

```
Input : a set $\mathcal{P}$ of $n$ policies, each $\pi$ parameterized for indexing so that
        $\pi \in \{1, 2, \cdots, n\}$; the set $\mathcal{J}_t$ of jobs that arrive at $t$;
1  initialize the weight vector of policies:
   $w_1 = \{w_{1,1}, \cdots, w_{1,n}\} = \{1/n, \cdots, 1/n\}$;
2  for $t \leftarrow 1$ to $T$ do
3      if $\mathcal{J}_t \neq \emptyset$ then
4          for each $j \in \mathcal{J}_t$, pick a policy $\pi$ with a probability $w_{j,\pi}$, being
           applied to $j$;
5      if $t \leq d$ then
6          $w_{j+1} \leftarrow w_j$;
7      else
8          while $\mathcal{J}_{t-d} \neq \emptyset$ do
9              $\eta_t \leftarrow \sqrt{\frac{2\log n}{d(t-d)}}$;
10             get a job $j$ from $\mathcal{J}_{t-d}$;
11             for $\pi \leftarrow 1$ to $n$ do
12                 $w'_{j+1,\pi} \leftarrow w_{j,\pi} \exp^{-\eta_t c_j(\pi)}$;
13             for $\pi \leftarrow 1$ to $n$ do
14                 $w_{j+1,\pi} \leftarrow \frac{w'_{j+1,\pi}}{\sum_{i=1}^{n} w'_{j+1,i}}$;
15             $\mathcal{J}_{t-d} \leftarrow \mathcal{J}_{t-d} - \{j\}$;
```

As modeled in Chapter 7.1, the cost of completing a job is from the use of spot and on-demand instances alone and is defined as their cost. For every job $j \in \mathcal{J}$, let π_j denote the policy defined by Algorithm 13 under which j is completed. Denote by $c_j(\pi)$ the cost of completing j under some policy $\pi \in \mathcal{P}$. Let $N' = |\cup_{t=d+1}^{T} \mathcal{J}_t|$, i.e., the number of all jobs that arrive in $[d+1, T]$, and, as proved in [6], we have that

Proposition 24. *For all $\delta \in (0,1)$, it holds with a probability at least $1-\delta$ over the random of online learning that*

$$\max_{\pi\in\mathcal{P}} \left\{ \sum_{t\in\cup_{t=d+1}^{T}\mathcal{J}_t} \frac{c_j(\pi_j) - c_j(\pi)}{N'} \right\} \leq 9\sqrt{\frac{2d\log(n/\delta)}{N'}}.$$

Proposition 24 says that, as an online learning algorithm runs, the actual total cost of completing all jobs is close to the cost of completing all jobs under a policy $\pi^* \in \mathcal{P}$ that generates the lowest total cost. Recall that a policy is defined by a tuple of parameters from $\mathcal{P}$.

Chapter 9

Evaluation

Nothing in the world can take the place of Persistence. Talent will not; nothing is more common than unsuccessful men with talent. Genius will not; unrewarded genius is almost a proverb. Education will not; the world is full of educated derelicts. Persistence and determination alone are omnipotent.

Calvin Coolidge

The main aim of our evaluations is to show the effectiveness of the proposed policies.

9.1 Simulation Setups

The on-demand price is $p = 0.25$ per hour. We set L to 5 (minutes) and all jobs have a parallelism bound of 20. Following [81], [82], we generate the jobs as follows. The job's arrival is generated according to a poisson distribution with a mean of 2. The size z_j of every job j is set to $12 \times 20 \times x$ where x follows a bounded Pareto distribution with a shape parameter $\epsilon = \frac{1}{1.01}$, a scale parameter $\sigma = \frac{1}{6.06}$ and a location parameter $\mu = \frac{1}{6}$; the maximum and minimum value of x is set to 1 and 10. The job's relative deadline is generated as $x \cdot z_j/\delta_j$, where x is uniformly distributed over $[1, x_0]$. x represents the slackness of a job; it affects the jobs' capability to utilize spot instances as shown by Proposition 21, and is a main factor that determines the performance. We consider three types of jobs respectively with a small, medium, and large slackness: the 1st, 2nd, 3rd types of jobs respectively with $x_0 = 3, 7, 13$. Spot prices are updated every time slot and their values can follow an exponential distribution where its mean is set to 1.1 [76].

Proposed Policies. The policies proposed in this thesis are parameterized: β and b are used for determine the allocation of spot and on-demand instances (see lines 5-13 of Algorithm 13), and β_0 is for self-owned instances (see lines 1-4 of Algorithm 13). The parameter β_0 is chosen in $\mathcal{C}_1 = \{\frac{i}{10} \mid 1 \leq 0 \leq 6\}$. As illustrated in Fig. 8.2, for jobs with $x_0 > 1.25$, the amount of self-owned instances allocated to jobs can be effectively controlled by selecting a value ≤ 0.6; for the others with little flexibility to utilize spot instances, they will be a large number of self-owned instances whenever possible to reduce the consumption of on-demand instances. The parameter β is chosen from $\mathcal{C}_2 = \{\frac{i}{10} \mid 0 \leq i \leq 9\} \cup \{0.9999\}$. The bid price b is chosen in $\mathcal{B} = \{b_i = 0.13 + 0.03 \cdot (i-1) \mid 1 \leq i \leq 6\}$. When only spot and on-demand instances are considered, let $\boldsymbol{\mathcal{P}} = \{(\beta, b) \mid \beta \in \mathcal{C}_2, b \in \mathcal{B}\}$, representing all the proposed policies to be evaluated; when self-owned instances are also taken into account, let $\boldsymbol{\mathcal{P}} = \{(\beta, b, \beta_0) \mid \beta_0 \in \mathcal{C}_1, \beta \in \mathcal{C}_2, b \in \mathcal{B}\}$.

Compared Policies. The proposed policies are compared with (i) the naive policy (8.1) for self-owned instances and (ii) the policy proposed in [6], [68] only for spot and on-demand instances (see Algorithm 1 in [6]). The latter randomly selects a parameter $\theta \in \Theta = \{\frac{i}{10} \mid 0 \leq i \leq 10\}$ for every job j: (i) the user will bid a price b for $\theta \cdot \delta_j$ spot instances and acquire $(1-\theta) \cdot \delta_j$ on-demand instances at every allocation update of j; (ii) it monitors at every slot t whether there is a risk of not completing the job by its deadline if only $(1-\theta) \cdot \delta_j$ on-demand instances are utilized in the remaining slots; (iii) if such risk exists, there is no flexibility for utilizing spot instances and it turns to utilize $\min\left\{\delta_j, \left\lceil z_j^{i_j+1}/Len \right\rceil\right\}$ on-demand instances alone until j is completed[1]. Let $\boldsymbol{\mathcal{P}'} = \{(\theta, b) \mid \theta \in \Theta, b \in \mathcal{B}\}$, representing all the policies of [6], [68].

Performance Metric. Let π denote a policy in $\mathcal{P}$ or $\mathcal{P}'$. Given a set of jobs $\mathcal{J}$ that arrive over time, our aim is to minimize the cost of completing all jobs in $\mathcal{J}$; and a main performance metric is *the average unit cost of processing jobs* when the x_2-th type of jobs are processed with x_1 self-owned instances available, i.e.,

— the ratio of the total cost of utilizing various instances to the processed workload of jobs, denoted by α_{x_1,x_2}, where $\alpha_{x_1,x_2} = \sum_{j\in\mathcal{J}} c_j(\pi)/\sum_{j\in\mathcal{J}} z_j$.

When a policy in $\mathcal{P}$ or $\mathcal{P}'$ is applied to process all jobs, we denote by $\alpha_{x_1,x_2}(\pi)$ the corresponding average unit cost of processing jobs. Against the unknown statistics of spot prices and job's characteristics, there are some policies in $\mathcal{P}$ or $\mathcal{P}'$ that are the most cost-effective. We use α_{x_1,x_2} (resp. α'_{x_1,x_2}) to denote the minimum of the average unit costs of our policies (resp. the policies in [6], [68] and defined by (8.1)), where $x_2 = 1, 2$, e.g., $\alpha_{x_1,x_2} = \min_{\pi\in\mathcal{P}}\{\alpha_{x_1,x_2}(\pi)\}$.

The performance of the intuitive policy (8.1) (for self-owned instances) and the existing policy in [6], [68] (for spot and on-demand instances) are used

1. In [6], [68], the workload of j is measured in instance hours.

as *the baseline* to measure the performance of the proposed policies; so, one performance indicator can be as follows:

$$\rho_{x_1,x_2} = 1 - \frac{\alpha_{x_1,x_2}}{\alpha'_{x_1,x_2}};$$

it represents the performance improvement of the proposed policies $\mathcal{P}$ over the baseline, that is, the ratio in cost reduction. Moreover, in this thesis, the online learning algorithm, i.e., Algorithm 14, is run to actually select a policy for each arriving job. The selection is random according to a distribution that will be updated according to the cost of completing that job; after numerous jobs are processed, the policies that generate the lowest cost will be associated with the highest probability. In this case, we use $\overline{\alpha}_{x_1,x_2}(\mathcal{P})$ or $\overline{\alpha}_{x_1,x_2}(\mathcal{P}')$ to denote the average unit cost of processing jobs when $\mathcal{P}$ or $\mathcal{P}'$ is applied to Algorithm 14. When online learning is applied, the performance indicator can be as follows:

$$\overline{\rho}_{x_1,x_2} = 1 - \frac{\overline{\alpha}_{x_1,x_2}(\mathcal{P})}{\overline{\alpha}_{x_1,x_2}(\mathcal{P}')};$$

it represents the ratio in cost reduction when online learning is applied.

9.2 Results

In the following, we give the results of simulations that are taken over about 60000 jobs, mainly listed in Tables 9.1, 9.3, 9.6, and 9.7. In our simulations, all fractional solutions will be rounded up to the nearest integers.

Experiment 1. We aim to evaluate the effectiveness of the proposed policies $\mathcal{P}$ for spot and on-demand instances alone by means of comparisons with the policies $\mathcal{P}'$ in [6], [68], where $x_1 = 0$. The simulation results are listed in Table 9.1 and show a noticeable cost reduction by up to 64.51%.

Table 9.1 – Performance Improvements for Spot and On-Demand Instances

$\rho_{0,1}$	$\rho_{0,2}$	$\rho_{0,3}$
58.87%	60.84%	64.51%

There are a total of 66 policies in $\mathcal{P}$. In our simulations, every 11 policies are grouped together and they use the same bid price. We have in the same group of policies that the cost-optimal value of β (denoted by β^*) is the same even under different types of jobs; the particular results are illustrated in Table 9.2. So, in the rest of our simulations, the effective range of β will be defined in $\{0.5, 0.6, 0.7, 0.8, 0.9, 0.999999\}$, to which we reset the value of $\mathcal{C}_2$.

Table 9.2 – The Optimal β under a Bid Price b

b	0.13	0.16	0.19	0.22	0.25	0.28
β	0.7	0.8	0.9	0.9	0.999999	0.999999

Experiment 2. We aim to evaluate the proposed policy for self-owned instances, compared with the naive policy in (8.1); here, the allocation of spot and on-demand instances will use the same policy $\mathcal{P}$ proposed in this thesis. The simulation results are listed in Table 9.3, showing a noticeable cost reduction by up to 43.74%.

Table 9.3 – Performance Improvement for Self-Owned Instances

	ρ_{200,x_2}	ρ_{400,x_2}	ρ_{600,x_2}	ρ_{800,x_2}
$x_2 = 1$	15.73%	21.41%	27.07%	22.83%
$x_2 = 2$	27.25%	39.59%	34.04%	17.85%
$x_2 = 3$	33.05%	34.41%	43.74%	31.88%

The utilizations of self-owned instances under different policies are illustrated in Fig. 9.1, where the red, blue, magenta, and black stars are respectively in the case where $x_1 = 200, 400, 600$ and 800. The allocation of self-owned instances are determined by the policy (8.3) or (8.1). Given a set of jobs, the utilization of self-owned instances under the policy (8.3) only depends on the parameter β_0 since their allocation is before and independent of the allocation of spot and on-demand instances. The intuitive policy (8.1) is a special form of the policy (8.3) when $\beta_0 = 0$. In the case that $x_2 = 2$, when $x_1 = 200, 400, 600, 800$, the minimum average unit cost is generated when $\beta = 0.3, 0.2, 0.2, 0.1$ respectively; the corresponding utilizations are given in Table 9.4; the utilization of the intuitive policy (8.1) is illustrated in Table 9.5. We can see that, given a case of x_1 and x_2, the proposed policy achieves a lower utilization than the intuitive policy; even so, it still achieves a lower average unit cost as shown in Table 9.3 where $x_2 = 2$. This is because the proposed policy could effectively reduce the unnecessary consumption of on-demand instances as explained in Chapter 8.1.3.

Table 9.4 – The Instance Utilization of the Proposed Policy under Cost-Optimal β_0

(β_0, x_1)	(0.3, 200)	(0.2, 400)	(0.2, 600)	(0.1, 800)
Utilization	89.89%	92.41%	72.70%	96.39%

Table 9.5 – The Instance Utilization of the Intuitive Policy

x_1	200	400	600	800
Utilization	99.73%	99.57%	99.31%	98.89%

Experiment 3. Assume that there are some amount of self-owned instances, and we show the performance improvement of the proposed policies $\mathcal{P}$, compared with the policies that use $\mathcal{P}'$ for spot and on-demand instances and (8.1) for self-owned instances. The simulation is done under the 2nd type of jobs that

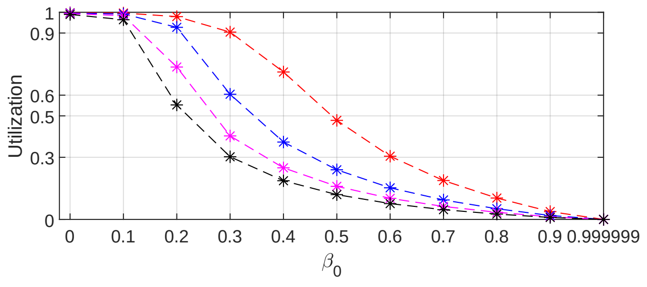

Figure 9.1 – The utilization of self-owned instances under different values of β_0.

have a medium slackness, and the results are listed in Table 9.6, showing the improvement of performance by up to 75.68%.

Table 9.6 – Performance Improvement for Three Types of Instances

$\rho_{200,2}$	$\rho_{400,2}$	$\rho_{600,2}$	$\rho_{800,2}$
71.30%	75.68%	72.83%	66.65%

Experiment 4. Now, we show the performance of the proposed policies when online learning is applied. The simulation setting is the same as Experiment 3 except that only the 2nd type of jobs is processed here. The related results are illustrated in Table 9.7, showing a cost reduction by up to 66.71%.

Table 9.7 – Performance Improvement under Online Learning

$\overline{\rho}_{0,2}$	$\overline{\rho}_{200,2}$	$\overline{\rho}_{400,2}$	$\overline{\rho}_{600,2}$	$\overline{\rho}_{800,2}$
60.89%	63.28%	66.71%	63.60%	51.11%

Chapter 10

Concluding Remarks

When you're trying to solve a problem, you initially spend a lot of time in the foothills. Your progress is very slow.

Then you reach a point where you can carry the problem with you, and your progress becomes very rapid. You quickly get to the top of the mountain. At this point you've solved the problem, and it's tempting to climb down and begin something new.

But you should stay on the top of the mountain. See how far you can go in different directions. Often there are discoveries you can make very rapidly because you've already done a lot of the climbing.

Donald Knuth

10.1 Conclusions

The problem of scheduling and pricing is central to the cloud computing field since resource efficiency and utility maximization are often the most significant concern for cloud providers and users. In this thesis, for the fundamental model of malleable tasks arising in cloud computing, we are the first to identify the optimal state such that the machines can be said to be optimally utilized by a type of malleable tasks that arises in the cloud and propose the first polynomial-time optimal algorithm to achieve the optimal state. Its importance can be perceived by the following fact: if the resource utilization state is not optimal in an algorithm, its performance can be improved by utilizing the machines optimally, allowing more tasks to be completed or reducing the overall completion times of tasks. The core results have provided a conceptual tool to enable proposing new analysis and design of algorithms or to enable improving existing algorithms for extensive scheduling objectives. These scheduling algorithms either improve the quality of service of the cloud provider or allow the cloud to serve more jobs, increasing its revenue.

The above tasks are assumed to be work-preserving, i.e., the speedup is linear and a task's workload remains constant when the number p of processors on which to execute this task is small and does not exceed a parallelism bound. We also consider a more general model to incorporate the case when a task is assigned a larger number of processors: when p exceeds the parallelism bound, the workload of a task increases but its execution time decreases as p increases; however, there is a threshold value that the number of assigned to a task cannot exceed. In this thesis, we propose scheduling algorithms to minimize the makespan or maximize the sum of values of tasks completed by a deadline.

On the other hand, in the pricing aspect, we study the problem of cost-efficiently utilizing self-owned instances and the instances (spot, on-demand) from public clouds such as Amazon EC2. The workload to be processed is assumed to be independent malleable tasks where there is a constraint of time by which to complete the task, and we propose (near-)optimal parametric policies to allocate different types of instances among the arriving tasks and the effectiveness of these policies is also validated by simulations. What is more, we identify and address in this thesis two underlying questions in the instance allocation process: to be cost-efficient, what properties should be kept in the policy for allocating self-owned instances and what policy can maximize the utilization of spot instances. This also enables us to extend the work of this thesis to the case where there are precedence constraints among tasks, which will be one of our future works.

10.2 Outlook of the Future

This thesis focuses on the performance improvement to cloud systems and the cost-effective ways for the user to utilize the computing resources under the current purchase options. There are still lots of future works to be done. In the scheduling aspect, for malleable tasks, future works include exploring the possibility of extending the definition in this thesis of the optimal state of executing malleable tasks on identical machines respectively to the case with release time and to the case in which each task consists of several subtasks with precedence constraints. Then, based on this, one may attempt to find the optimal schedule for those cases and to propose similar algorithms in this thesis for those extended cases. According to the results in [8], the optimal schedule that achieves the optimal resource utilization is also the key to the objective of minimizing the total weighted completion time of all the tasks. Hence, one can also consider how to apply the results in this thesis to improve the algorithms in [8].

In the pricing aspect, Infrastructure as a Service (IaaS) is a means of delivering value to users by facilitating user's access to computing capacities without the ownership and maintenance of infrastructures. The design of IaaS service obeys fundamental principles in service design: plan and organize people, infrastructure, communication and material components of the service, aiming at excellence along three dimensions, namely, the service quality, the system's efficiency, and the interaction between the service provider and its users. This requires a joint effort to understand both the needs of customers and implement efficient resource management.

Thanks to this thesis, our work on scheduling has made us better understand what task's characteristics can bring better resource efficiency [17], [33], [83]–[85] while our work on cost-efficiently utilizing public clouds enables us to better perceive what pricing models could be of high usability [69], [70]; the pricing models in Amazon EC2 define the process under which users can acquire them and are not very user-friendly, as indicated by some works [65], [76], [86]. We were considering what forms of computing service should be offered to users by the cloud so that the offered services can be more user-friendly while the revenue of the cloud is maximized; here, the forms of services offered require a corresponding resource management scheme used inside the cloud system. We believed that, QoS-differentiation is an important lens in order to efficiently allocate shared resources in many domains where both (i) users have potentially diverse preferences for QoS and (ii) different preferences take differentiable effects on resource efficiency; to realize this, appropriate pricing is needed to shape user's preferences, incentivizing potential users (whose demand of resources is of time elasticity) to express their demand/jobs in forms that enable maximizing the power of QoS-differentiation in resource management.

For example, at the moment of submitting this thesis, we have completed the performance analysis of a type of QoS-differentiated pricing in cloud computing via an analytical approach where the cloud offers multiple QoS classes: the jobs of each class will be completed with a finite waiting time and the smaller the waiting time, the higher the price; our numerical simulations have shown that the proposed architecture could improved the revenue of a cloud provider by up to a five-fold increase [87]. Other examples of QoS-differentiation including colocating two types of workloads on the same servers where one type has a definite delay requirement while the other only requires best-effort service, commonly used in traditional private computer systems to improve resource efficiency greatly. In the cloud computing context, the aspect of pricing shared resources needs to be incorporated and one could also use the analytical modeling method to study the related problems; here, the resource management model is very similar to the model in Amazon EC2, where the idle instances of the on-demand market are sold in the form of spot instances to improve the resource efficiency [88]. This form of QoS differentiation is also a promising direction that remains to be addressed in future. To sum up, QoS-differentiated pricing and resource management is still an important and rich area for future research in cloud computing.

Bibliography

Bibliography

[1] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: A technology tutorial", *IEEE Access*, vol. 2, pp. 652–687, 2014.

[2] N. Jain, I. Menache, J. S. Naor, and J. Yaniv, "A truthful mechanism for value-based scheduling in cloud computing", *Theory of Computing Systems*, vol. 54, no. 3, pp. 388–406, 2014.

[3] N. Jain, I. Menache, J. S. Naor, and J. Yaniv, "Near-Optimal Scheduling Mechanisms for Deadline-Sensitive Jobs in Large Computing Clusters", *ACM Transactions on Parallel Computing*, vol. 2, no. 1, 3:1–3:29, 2015.

[4] B. Lucier, I. Menache, J. S. Naor, and J. Yaniv, "Efficient online scheduling for deadline-sensitive jobs", in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA'13, ACM, 2013, pp. 305–314.

[5] Y. Azar, I. Kalp-Shaltiel, B. Lucier, I. Menache, J. S. Naor, and J. Yaniv, "Truthful Online Scheduling with Commitments", in *Proceedings of the Sixteenth ACM Conference on Economics and Computation*, ser. EC'15, ACM, 2015, pp. 715–732.

[6] I. Menache, O. Shamir, and N. Jain, "On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud", in *Proceedings of the 11th International Conference on Autonomic Computing*, ser. ICAC'14, USENIX Association, pp. 177–187.

[7] P. Bodík, I. Menache, J. S. Naor, and J. Yaniv, "Brief announcement: deadline-aware scheduling of big-data processing jobs", in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA'14, ACM, 2014, pp. 211–213.

[8] V. Nagarajan, J. Wolf, A. Balmin, and K. Hildrum, "Flowflex: Malleable scheduling for flows of mapreduce jobs", in *Proceedings of the ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, ser. Middleware'13, Springer, 2013, pp. 103–122.

[9] J. Wolf, Z. Nabi, V. Nagarajan, R. Saccone, R. Wagle, K. Hildrum, E. Pring, and K. Sarpatwar, “The X-flex Cross-platform Scheduler: Who’s the Fairest of Them All?”, in *Proceedings of the Middleware Industry Track*, ser. Industry papers, ACM, 2014, 1:1–1:7.

[10] E. L. Lawler, “A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs”, *Annals of Operations Research*, vol. 26, no. 1, pp. 125–133, 1990.

[11] D. Karger, C. Stein, and J. Wein, “Scheduling algorithms”, in *Algorithms and theory of computation handbook*, M. J. Atallah, Ed., CRC Press, 1998, ch. 35.

[12] W. Horn, “Some simple scheduling algorithms”, *Naval Research Logistics Quarterly*, vol. 21, no. 1, pp. 177–185, 1974.

[13] J. R. Jackson, *Scheduling a production line to minimize maximum tardiness*, management science research project, University of California, Los Angeles, 1955.

[14] E. L. Lawler and J. M. Moore, “A Functional Equation and Its Application to Resource Allocation and Sequencing Problems”, *Management Science*, vol. 16, no. 1, pp. 77–84, 1969.

[15] J. A. Stankovic, K. Ramamritham, and M. Spuri, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[16] T. White, *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[17] X. Wu and P. Loiseau, “Algorithms for scheduling deadline-sensitive malleable tasks”, in *Proceedings of the 53rd Annual Allerton Conference on Communication, Control, and Computing*, ser. Allerton’15, IEEE, 2015, pp. 530–537.

[18] ——, “Algorithms for Scheduling Malleable Cloud Tasks”, *arXiv:1501.04343v8 (Preprint)*, 2015.

[19] J. Chuzhoy, S. Guha, S. Khanna, and J. S. Naor, “Machine Minimization for Scheduling Jobs with Interval Constraints”, in *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, ser. FOCS’04, IEEE, 2004, pp. 81–90.

[20] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*. Prentice-Hall, Inc., 1996.

[21] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[22] G. Even, “Recursive Greedy Methods”, in *Handbook of Approximation Algorithms and Metaheuristics*, T. F. Gonzalez, Ed., CRC Press, 2007, ch. 5.

[23] G. Mounie, C. Rapine, and D. Trystram, "Efficient Approximation Algorithms for Scheduling Malleable Tasks", in *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA'99, ACM, 1999, pp. 23–32.

[24] G. Mounie, C. Rapine, and D. Trystram, "A $\frac{3}{2}$-approximation algorithm for scheduling independent monotonic malleable tasks", *SIAM J. Comput.*, vol. 37, no. 2, pp. 401–412, 2007.

[25] R. A. Dutton, W. Mao, J. Chen, and W. W. III, "Parallel Job Scheduling with Overhead: A Benchmark Study", in *Proceedings of the 2008 International Conference on Networking, Architecture, and Storage*, ser. NAS'08, IEEE, 2008, pp. 326–333.

[26] Wikipedia Contributors, *Embarrassingly parallel*, available online at `https://en.wikipedia.org/wiki/Embarrassingly_parallel` [accessed on June 17, 2018].

[27] ——, *Multigrid Method*, available online at `https://en.wikipedia.org/wiki/Multigrid_method` [accessed on June 17, 2018].

[28] A. Mariano, S. Timnat, and C. Bischof, "Lock-Free GaussSieve for Linear Speedups in Parallel High Performance SVP Calculation", in *Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD'14, IEEE, 2014, pp. 278–285.

[29] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu, "SIFT Implementation and Optimization for Multi-Core Systems", in *IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[30] D. Darriba, G. L. Taboada, R. Doallo, and D. Posada, "ProtTest 3: Fast Selection of Best-Fit Models of Protein Evolution", *Bioinformatics*, vol. 27, no. 8, pp. 1164–1165, 2011.

[31] P. Maruzewski, D. L. Touzé, G. Oger, and F. Avellan, "SPH high-performance computing simulations of rigid solids impacting the free-surface of water", *Journal of Hydraulic Research*, vol. 48, no. S1, pp. 126–134, 2010.

[32] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On Parallel Scalable Uniform SAT Witness Generation", in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'15, Springer, 2015, pp. 304–319.

[33] X. Wu, P. Loiseau, and H. Esa, "Efficient Algorithms for Scheduling Moldable Tasks", *arXiv:1609.08588v8 (Preprint submittted to European Journal of Operational Research)*, 2017.

[34] J. T. Havill and W. Mao, "Competitive online scheduling of perfectly malleable jobs with setup times", *European Journal of Operational Research*, vol. 187, no. 3, pp. 1126–1142, 2008.

[35] R. A. Dutton and W. Mao, "Online Scheduling of Malleable Parallel Jobs", in *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, ser. PDCS'07, ACTA Press, 2007, pp. 136–141.

[36] N. Kell and J. Havill, "Improved upper bounds for online malleable job scheduling", *Journal of Scheduling*, vol. 18, no. 4, pp. 393–410, 2015.

[37] S. Guo and L. Kang, "Online scheduling of malleable parallel jobs with setup times on two identical machines", *European Journal of Operational Research*, vol. 206, no. 3, pp. 555–561, 2010.

[38] Wikipedia Contributors, *Supercomputer Architechture*, available online at `https://en.wikipedia.org/wiki/Supercomputer_architecture` [accessed on June 17, 2018].

[39] Y. Aridor, T. Domany, O. Goldshmidt, Y. Kliteynik, J. Moreira, and E. Shmueli, "Open Job Management Architecture for the Blue Gene/L Supercomputer", in *Proceedings of the 11th International Conference on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP'05, Springer, 2005, pp. 91–107.

[40] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg", in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys'15, ACM, 2015, 18:1–18:17.

[41] K. Jansen and G. Zhang, "Maximizing the total profit of rectangles packed into a rectangle", *Algorithmica*, vol. 47, no. 3, pp. 323–342, 2007.

[42] K. Jansen and G. Zhang, "Maximizing the Number of Packed Rectangles", in *the 9th Scandinavian Workshop on Algorithm Theory*, Springer, 2004, pp. 362–371.

[43] A. V. Fishkin, O. Gerber, K. Jansen, and R. Solis-Oba, "Packing Weighted Rectangles into a Square", in *Proceedings of the 30th International Conference on Mathematical Foundations of Computer Science*, ser. MFCS'05, Springer, 2005, pp. 352–363.

[44] J. Blazewicz, M. Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz, "Preemptable Malleable Task Scheduling Problem", *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 486–490, 2006.

[45] J. Błażewicz, M. Machowiak, J. Węglarz, M. Y. Kovalyov, and D. Trystram, "Scheduling malleable tasks on parallel processors to minimize the makespan", *Annals of Operations Research*, vol. 129, no. 1, pp. 65–80, 2004.

[46] J. Blazewicz, M. Machowiak, G. Mounie, and D. Trystram, "Approximation Algorithms for Scheduling Independent Malleable Tasks", in *Proceedings of the 7th International European Conference on Parallel Processing*, ser. Euro-Par'01, Springer, 2001, pp. 191–197.

[47] D. Trystram, "Scheduling Parallel Applications Using Malleable Tasks on Clusters", in *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, ser. IPDPS'01, IEEE, 2001, pp. 2128–2135.

[48] P.-F. Dutot and D. Trystram, "Scheduling on Hierarchical Clusters Using Malleable Tasks", in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA'01, ACM, 2001, pp. 199–208.

[49] R. Lepère, G. Mounié, and D. Trystram, "An approximation algorithm for scheduling trees of malleable tasks", *European Journal of Operational Research*, vol. 142, no. 2, pp. 242–249, 2002.

[50] K. Jansen and R. Thöle, "Approximation algorithms for scheduling parallel jobs", *SIAM Journal on Computing*, vol. 39, no. 8, pp. 3571–3615, 2010.

[51] R. Harren, K. Jansen, L. Prädel, and R. Van Stee, "A $(5/3+\epsilon)$-approximation for strip packing", *Computational Geometry: Theory and Applications*, vol. 47, no. 2, pp. 248–267, 2014.

[52] K. Jansen and H. Zhang, "Scheduling malleable tasks with precedence constraints", *Journal of Computer and System Sciences*, vol. 78, no. 1, pp. 245–259, 2012.

[53] K. Jansen, "A$(3/2+\epsilon)$ Approximation Algorithm for Scheduling Moldable and Non-moldable Parallel Tasks", in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA'12, ACM, 2012, pp. 224–235.

[54] K. Jansen and H. Zhang, "An Approximation Algorithm for Scheduling Malleable Tasks Under General Precedence Constraints", *ACM Trans. Algorithms*, vol. 2, no. 3, pp. 416–434, 2006.

[55] K. Jansen and F. Land, "Scheduling monotone moldable jobs in linear time", in *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS'18, IEEE, 2018, pp. 172–181.

[56] J. Y. Leung, *Handbook of scheduling: algorithms, models, and performance analysis.* CRC Press, 2004.

[57] J. Turek, J. L. Wolf, and P. S. Yu, "Approximate Algorithms Scheduling Parallelizable Tasks", in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA'92, ACM, 1992, pp. 323–332.

[58] W. T. Ludwig, "Algorithms for Scheduling Malleable and Nonmalleable Parallel Tasks", UMI Order No. GAX95-33447, PhD thesis, Madison, WI, USA, 1995.

[59] W. Ludwig and P. Tiwari, "Scheduling Malleable and Nonmalleable Parallel Tasks", in *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA'94, Society for Industrial and Applied Mathematics, 1994, pp. 167–176.

[60] A. Steinberg, "A Strip-Packing Algorithm with Absolute Performance Bound 2", *SIAM Journal on Computing*, vol. 26, no. 2, pp. 401–409, 1997.

[61] K. Jansen and L. Porkolab, "Linear-time approximation schemes for scheduling malleable parallel tasks", *Algorithmica*, vol. 32, no. 3, pp. 507–520, 2002.

[62] B. Korte and J. Vygen, "Fractional Knapsack and Weighted Median Problem", in *Combinatorial Optimization: Theory and Algorithms*, CRC Press, 1998, ch. 17.1.

[63] *Cisco Maintains Lead in Public Cloud Infrastructure while HP Leads in Private*, available online at `https://www.srgresearch.com/articles/cisco-maintains-lead-public-cloud-infrastructure-while-hp-leads-private` [accessed on Sept. 13, 2018].

[64] *Sizing the Public Cloud Computing Market*, available online at `http://softwarestrategiesblog.com/2011/06/01/sizing-the-public-cloud/` [accessed on Sept. 13, 2018].

[65] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 Spot Instance Pricing", *ACM Trans. Econ. Comput.*, vol. 1, no. 3, 16:1–16:20, 2013.

[66] *Amazon EC2 Spot Instances Pricing*, available online at `http://aws.amazon.com/ec2/purchasing-options/spot-instances/` [accessed on Sept. 13, 2018].

[67] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed Job Latency in Data Parallel Clusters", in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys'12, ACM, 2012, pp. 99–112.

[68] N. Jain, I. Menache, and O. Shamir, "Allocation of computational resources with policy selection", US Patent No.: 9652288, May 2017.

[69] X. Wu, P. Loiseau, and E. Hyytiä, "Towards designing cost-optimal policies to utilize IaaS clouds with online learning", in *Cloud and Autonomic Computing (ICCAC), 2017 International Conference on*, IEEE, 2017, pp. 160–171.

[70] X. Wu, P. Loiseau, and H. Esa, "Towards designing cost-optimal policies to utilize IaaS clouds with online learning", *arXiv:1607.05178v6 (Preprint submittted to IEEE Transactions on Parallel and Distributed Systems)*, 2016.

[71] M. Zafer, Y. Song, and K.-W. Lee, "Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs", in *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, ser. CLOUD'12, IEEE, 2012, pp. 75–82.

[72] M. Yao, P. Zhang, Y. Li, J. Hu, C. Lin, and X. Y. Li, "Cutting your cloud computing cost for deadline-constrained batch jobs", in *Proceedings of the 2014 IEEE International Conference on Web Services*, ser. ICWS'14, IEEE, 2014, pp. 337–344.

[73] S. S. Manvi and G. K. Shyam, "Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey", *Journal of Network and Computer Applications*, vol. 41, pp. 424–440, 2014.

[74] Y.-J. Hong, J. Xue, and M. Thottethodi, "Dynamic server provisioning to minimize cost in an IaaS cloud", in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ser. Sigmetrics'11, ACM, 2011, pp. 147–148.

[75] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of Resource Provisioning Cost in Cloud Computing", *IEEE Trans. Serv. Comput.*, vol. 5, no. 2, pp. 164–177, 2012.

[76] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang, "How to Bid the Cloud", in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM'15, ACM, 2015, pp. 71–84.

[77] S. Shi, C. Wu, and Z. Li, "Cost-Minimizing Online VM Purchasing for Application Service Providers with Arbitrary Demands", in *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing*, ser. CLOUD'15, IEEE, 2015, pp. 146–154.

[78] W. Wang, B. Liang, and B. Li, "Optimal online multi-instance acquisition in IaaS clouds", *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 12, pp. 3407–3419, 2015.

[79] A. Vintila, A.-M. Oprescu, and T. Kielmann, "Fast (re-) configuration of mixed on-demand and spot instance pools for high-throughput computing", in *Proceedings of the first ACM workshop on Optimization techniques for resources management in clouds*, ACM, 2013, pp. 25–32.

[80] L. Huang, X. Liu, and X. Hao, "The Power of Online Learning in Stochastic Network Optimization", in *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS'14, ACM, 2014, pp. 153–165.

[81] J. Chen, C. Wang, B. B. Zhou, L. Sun, Y. C. Lee, and A. Y. Zomaya, "Tradeoffs Between Profit and Customer Satisfaction for Service Provisioning in the Cloud", in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC'11, ACM, 2011, pp. 229–238.

[82] L. Zheng, C. Joe-Wong, C. G. Brinton, C. W. Tan, S. Ha, and M. Chiang, "On the Viability of a Cloud Virtual Service Provider", in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, ser. Sigmetrics'16, ACM, 2016, pp. 235–248.

[83] E. Hyytiä, R. Righter, O. Bilenne, and X. Wu, "Dispatching Fixed-sized Jobs with Multiple Deadlines to Parallel Heterogeneous Servers", in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS'17, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2017, pp. 161–161.

[84] ——, "Dispatching fixed-sized jobs with multiple deadlines to parallel heterogeneous servers", *Performance Evaluation*, vol. 114, pp. 32–44, 2017.

[85] ——, "Dispatching discrete-size jobs with multiple deadlines to parallel heterogeneous servers", in *Systems modeling: methodologies and tools*, ser. EAI/Springer Innovations in Communications and Computing, A. Puliafito and K. Trivedi, Eds., Springer, 2018.

[86] T. Sandholm, J. Ward, F. Balestrieri, and B. A. Huberman, "Qos-based pricing and scheduling of batch jobs in openstack clouds", *arXiv:1504.07283 (Preprint)*, 2015.

[87] X. Wu and F. De Pellegrini, "On the Benefits of QoS-Differentiated Posted Pricing in Cloud Computing: An Analytical Model", *arXiv:1709.08909 (Preprint)*, 2017.

[88] N. R. Devanur, "A Report on the Workshop on the Economics of Cloud Computing", *ACM SIGecom Exchanges*, vol. 15, no. 2, pp. 25–29, 2017.

Appendix A

Résumé en français

What I cannot create, I do not understand.

Richard Feynman

Dans l'annexe, nous donnons un aperçu de la thèse, récapitulons ses résultats principaux, et expliquons les idées de haut niveau derrière ces résultats.

A Introduction

Cette thése aborde les problèmes liés à la ordonnancement et à la tarification dans le cloud computing. Le cloud computing est un modèle qui permet un accès omniprésent à la demande à un pool partagé de ressources informatiques configurables. Les solutions de stockage et d'informatique en nuage offrent aux utilisateurs diverses capacités pour stocker et traiter leurs données dans des centres de données tiers. Les ressources sont partagées par différents utilisateurs pour assurer la cohérence et des économies d'échelle similaires à celles d'un service public (comme le réseau) sur un réseau. Par conséquent, il se concentre sur l'optimisation de l'efficacité des ressources partagées. D'autre part, le cloud computing permet aux utilisateurs d'‘éviter les coûts d'‘infrastructure initiaux et au service informatique d'adapter plus rapidement les ressources pour répondre aux demandes changeantes et imprévisibles de l'entreprise. Il présente les avantages d'une puissance de calcul élevée, d'un coût élevé, de l'évolutivité, de l'accessibilité et de la disponibilité, tout en réduisant la charge de l'utilisateur liée au déploiement et à la gestion des infrastructures informatiques. Les fournisseurs de cloud utilisent généralement une tarification basée sur l'utilisation, parfois en association avec une tarification dynamique pour vendre les ressources restantes à travers des enchères. Cela peut entraîner des frais plus élevés si les utilisateurs ne s'adaptent pas au modèle de tarification en nuage. D'autre part, maximiser l'utilité des ressources est souvent l'une des préoccupations les plus importantes des fournisseurs de cloud.

Les problèmes d'ordonnancement suivants sont importants pour les fournisseurs de cloud. Compte tenu de la capacité fixe d'un fournisseur de cloud, comment le fournisseur de cloud pourrait-il desservir autant de locataires que possible tout en respectant leurs délais; lorsque les valeurs des tâches sont différentes et que la capacité d'un fournisseur de cloud est limitée, comment le fournisseur de services cloud peut-il sélectionner un sous-ensemble de tâches pour maximiser la valeur totale des tâches? En théorie de la ordonnancement, ces questions correspondent aux objectifs de ordonnancement: "machine minimization" and "social welfare maximization". Compte tenu de la capacité du cloud, comment le cloud pourrait-il optimiser les objectifs de ordonnancement, tels que la réduction des retards dans l'exécution des tâches? Ces problèmes d'ordonnancement sont essentiels pour gérer les revenus d'un fournisseur de cloud et améliorer la qualité de service des utilisateurs; ils constituent les principales questions à aborder dans cette thèse.

En ce qui concerne la tarification, nous abordons le problème suivant: compte tenu des modèles de tarification actuels, comment un utilisateur peut-il acquérir des ressources informatiques dans le cloud de manière rentable? La motivation pour cette question est la suivante. De nombreuses entreprises possèdent une petite infrastructure qu'elles peuvent utiliser pour leurs tâches informatiques; ils ont souvent besoin d'acheter des ressources informatiques supplémentaires dans les clouds. L'infrastructure en tant que service (IaaS) permet aux utilisateurs d'ajuster dynamiquement leur capacité de calcul en fonction de la demande, qui varie dans le temps. Du point de vue de l'utilisateur, cela élimine le besoin d'acheter des serveurs pour répondre à la demande de pointe, sans provoquer une latence inacceptable. L'utilisation rentable des clouds IaaS est donc une préoccupation majeure des utilisateurs.

B Un aperçu de la thèse et des principaux résultats

La thèse comporte deux parties. La première partie aborde les questions de ordonnancement et étudie deux types de tâches. La deuxième partie aborde le problème de l'utilisation rentable des ressources du cloud public.

B.1 Partie I: vue d'ensemble et résultats

Vue d'ensemble de la partie II. Dans la première partie de cette thèse, nous posons des bases théoriques sur le problème de la ordonnancement en informatique en nuage.

- Chapitre 2

  Dans ce chapitre, nous introduisons l'importance des problémes d'ordonnancement dans le cloud computing et du modèle de tâches à prendre en compte dans ce scénario.
- Chapitre 3

  Dans ce chapitre, nous identifions l'état d'utilisation optimale des ressources de plusieurs machines sur lesquelles un ensemble de tâches avec des délais stricts est planifié; nous proposons ensuite un algorithme de ordonnancement pouvant atteindre un tel état optimal.
- Chapitre 4

  Les résultats du Chapitre 3 fournissent un outil conceptuel pour proposer de nouvelles conceptions d'algorithmes et l'amélioration des algorithmes existants en fonction de divers objectifs de ordonnancement. En particulier, on obtient les résultats algorithmiques suivants:

  (i) un algorithme gourmand optimal pour la maximisation du bien-être social (maximiser la somme des valeurs des tâches accomplies dans leurs délais),

  (ii) le premier algorithme de programmation dynamique exact pour la maximisation du bien-être social avec une complexité de calcul pseudo-polynomiale,

  (iii) un algorithme exact pour la minimisation de la machine (minimiser le nombre de machines nécessaires pour produire un programme réalisable d'un ensemble de tâches où chaque tâche est terminée avant la date limite),

  (iv) un algorithme amélioré ayant pour objectif de minimiser le temps d'achèvement pondéré maximal.
- Chapitre 5

  Dans ce chapitre, nous introduisons une variante du type de tâches ci-dessus. A travers une analyse algorithmique, nous proposons des algorithmes de ordonnancement pour la minimisation de makespan et pour la maximisation de la somme des valeurs des tâches exécutées avant une date limite.

Pour les problèmes d'ordonnancement ci-dessus, nous visons à proposer de bons algorithmes. Pour un ensemble arbitraire de tâches, si un algorithme génère toujours une planification de tâches sur des machines m avec des performances optimales, il est optimal. Si l'algorithme optimal ne peut pas être obtenu, l'algorithme peut être mesuré par un rapport de performance: le rapport entre la performance de l'algorithme proposé et la performance d'un algorithme idéalement optimal (inconnu de nous); ce ratio est généralement appelé ratio d'approximation. Formellement, on dénoter les performances d'un algorithme et de l'algorithme optimal respectivement par $A(\mathcal{T})$ et $OPT(\mathcal{T})$, et un algorithme š'appelle un algorithme ρ-approché s'il existe une valeur ρ: pour un ensemble arbitraire $\mathcal{T}$, (i) quand l'objectif est de minimiser quelque chose

$$\frac{A(\mathcal{T})}{OPT(\mathcal{T})} \leq \rho \ (\rho \geq 1),$$

et (ii) lorsque l'objectif est de maximiser quelque chose

$$\frac{A(\mathcal{T})}{OPT(\mathcal{T})} \geq \rho \ (0 \leq \rho \leq 1).$$

Dans cette thèse, notre objectif final est de proposer des algorithmes d'ordonnancement atteignant des rapports d'approximation proches de 1.

Principaux résultats. Le premier type de tâches est traité dans les Chapitres 2 à 4 et s'appelle **tâches malléables**; ils sont assumé être malléables:

- lors de l'exécution d'une tâche, le nombre de machines attribuées peut varier dans une limite de parallélisme (ce qui amène l'opération consistant à préempter l'exécution d'une tâche),
- la charge de travail d'une tâche est constante et ne change pas avec le nombre de machines qui lui sont affectées.

Avant nos travaux, elles avaient été bien étudiées par d'autres techniques algorithmiques telles que le "dual-fitting" et "the rounding of linear programs".

Plus récemment, Jain *et al.* a utilisé la technique du dual-fitting pour proposer un algorithme glouton qui permet d'obtenir un rapport d'approximation $\frac{C-k}{C} \cdot \frac{s-1}{s}$ [3]; ici, C est le nombre de machines, k est une limite supérieure du parallélisme, s est le slackness minimum de toutes les tâches où le slackness d'une tâche est défini comme étant le rapport entre son date limite et son temps d'exécution minimal (lorsqu'une tâche est toujours affectée au nombre maximal de machines tout au long de l'exécution). Intuitivement, s caractérise l'urgence ou la flexibilité de l'affectation des ressources, par exemple, $s = 1$ signifie que, pour rencontrer l'échéance, une tâche doit toujours utiliser le nombre maximal de machines qu'elle peut utiliser du début à la fin.

Les principaux résultats de cette thèse ont été résumés lorsque nous introduisons les Chapitres 3 à 4 ci-dessus. En particulier, l'algorithme glouton proposé dans cette thèse a un rapport approximatif $\frac{s-1}{s}$. L'algorithme glouton de Jain *et al.* [3] et le nÃ´tre représentent une classe d'algorithmes gloutons. Dans cette classe, les tâches sont considérées dans l'ordre décroissant de leurs valeurs marginales (c'est-à-dire le rapport entre la valeur d'une tâche et sa taille); si une tâche peut être complétée avant son échéance en fonction des machines disponibles, il sera accepté et entièrement alloué selon un certain algorithme d'allocation; sinon, il sera rejeté. Nous montrons en outre que

- $\frac{s-1}{s}$ est la meilleure garantie de performance possible que cette classe d'algorithmes glouton pourrait réaliser.
- par conséquent, l'algorithme glouton proposé dans cette thèse est le meilleur possible parmi cette classe d'algorithmes gloutons.

Le deuxième algorithme du Chapitre 4 est une application de la procédure de programmation dynamique. Cependant, avant notre travail, comment activer cette application est un problème ouvert, comme indiqué dans [2], [3]. Ceci est principalement dê à l'absence de notion de l'état optimal d'utilisation de la machine lorsque des tâches malléables avec des délais sont considérées, et l'absence d'algorithme permettant d'atteindre un tel état. En revanche, l'état optimal dans le cas d'une seule machine peut être obtenu par l'algorithme EDF. Les principaux résultats de cette thèse permettent d'appliquer la procédure de programmation dynamique au scénario de tâche malléable. Les troisième et quatrième algorithmes ci-dessus sont obtenus en appliquant respectivement les résultats centraux du Chapitre 3 à une procédure de recherche binaire et les résultats correspondants dans [8].

Le deuxiéme type de tâches est étudié au Chapitre 5. En particulier, motivés par de récentes études de référence, nous introduisons la notion de (δ, k)-monotonic tâches:

— les tâches sont moulables,

et pour chaque tâche T_j assignée à p processeurs, nous avons

(i) lorsque p est petit et compris entre $[1,\ \delta]$, sa charge de travail $D_{j,p}$ reste constante et l'accélération est linéaire;

(ii) lorsque p est grand et que sa valeur est comprise dans $[\delta+1,\ k]$, la charge de travail $D_{j,p}$ ne diminue pas lorsque p augmente alors que son temps d'exécution diminue et commence même à augmenter lorsque p dépasse un certain seuil;

(iii) le nombre maximal de processeurs pouvant être affectés à T_j est k.

Le (δ, k)-monotonic tâches sont le deuxième type de tâches que nous avons considáráes dans cette thèse; ici, "moldable" signifie qu'une tâche peut être assignáe un nombre flexible de machines dans $[1,\ k]$; cependant, après que le nombre soit dáterminá avant l'exácution de la tâche, ce nombre ne peut pas changer dans toute son exácution. Pour le deuxième type de tâches, nous proposons un algorithme d'átablissement du programme pour ráduire au minimum le makespan, dont la performance dápend de la valeur de δ; dans des scánarios ráalistes, δ can va de 5 à 64 et son rapport d'approximation va de $\frac{4}{3}$ à $\frac{11}{10}$; ici, si le rapport d'approximation d'un algorithme est ρ, la makespan atteinte par cet algorithme n'est pas toujours supárieure à ρ fois la makespan d'un algorithme optimal. En tant que sous-produit, nous fournissons ágalement un algorithme de planification qui maximise la somme des valeurs des tâches terminées avant une date limite.

B.2 Partie II: vue d'ensemble et résultats

Vue d'ensemble de la partie II. Dans la deuxième partie de cette thèse, nous considérons comment acquérir la ressource informatique du nuage d'une manière rentable.

— Chapitre 6

Dans ce chapitre, nous présentons les modèles actuels de tarification du marché du cloud, tels que les modèles d'Amazon Elastic Cloud Compute (EC2); deux options d'achat sont envisagées: les instances ponctuelles bon marché et les instances coûteuses á la demande (machines virtuelles). Les utilisateurs peuvent également avoir leurs propres instances, appelées instances auto-possédé.

— Chapitre 7

Dans ce chapitre, nous proposons des stratégies paramétriques rentables pour affecter différents types d'instances á des travaux [1]. Les circonstances sont dynamiques: les prix des "spot instances" varient de manière imprévisible dans le temps et les statistiques sur les emplois sont inconnues; ensuite, une approche d'apprentissage en ligne est appliquée pour estimer les paramètres de configuration optimaux en termes de coûts des politiques.

— Chapitre 8

Dans ce chapitre, nous évaluons l'efficacité des politiques paramétriques proposées au moyen de simulations approfondies, en particulier une réduction des coûts pouvant aller jusqu'á 64,51% lorsque les instances spot et on-demand sont prises en compte et jusqu'á 43,74% lorsque des instances "auto-possédées" sont considérées, par rapport aux stratégies proposées précédemment ou intuitives.

Résultats principaux. Dans la deuxième partie de cette thèse, nous examinons le problème de l'utilisation rentable des instances auto-possédées et les instances des clouds publics. Les deux options d'achat courantes dans le cloud sont les instances on-demand et spot. Les premiers sont toujours disponibles avec un prix fixe et les locataires [2] ne payez que pour la période au cours de laquelle les instances sont consommées á un taux horaire. Les utilisateurs peuvent également proposer un prix pour les instances spot et ne peuvent les obtenir avec succès que si leur prix est supérieur au prix au comptant. Les instances spot seront alors exécutées tant que l'offre sera supérieure au prix spot, mais elles seront résiliées si le prix spot devient plus élevé. Ici, les prix spot varient généralement de manière imprévisible dans le temps et les utilisateurs devront payer les prix spot pour leur utilisation. Par rapport aux instances on-demand, les instances spot peuvent réduire les coûts de 50% á 90%.

Les utilisateurs qui achètent des instances sur le cloud peuvent disposer de leurs propres instances, appelées instances auto-possédées, qui peuvent être utilisées pour traiter des travaux mais sont parfois insuffisantes (d'où la nécessité d'acheter des instances IaaS supplémentaires). Ils peuvent également ne pas avoir d'instances auto-possédées (par exemple, dans le cas de startups) et doivent donc acheter dans le cloud toutes les ressources informatiques nécessaires. Dans les deux cas, la question fondamentale pour les utilisateurs est de déterminer comment acheter des instances auprès de clouds IaaS et utiliser différentes instances pour traiter leurs travaux de manière à minimiser leurs coûts.

Les emplois des locataires arrivent avec le temps et ont des contraintes à satisfaire. Par exemple, une contrainte est liée au parallélisme et spécifie le nombre maximal d'instances pouvant être utilisées simultanément par un travail; un autre est sur la synchronisation, c.-à-d., une date-butoir avant laquelle le emplois doit être accompli.

1. Dans cette thèse, nous utilisons indifféremment "emplois" et "tâches".
2. Dans cette thèse, nous utilisons indifféremment "utilisateurs" et "locataires".

Dans le processus d'allocation d'instance, deux questions théoriques sous-jacentes sont bien traitées: pour économiser de l'argent, quelles propriétés doivent être conservées dans la politique d'allocation des instances self-owned et quelle politique peut maximiser l'utilisation des instances spot après les instances self-owned sont utilisées, évitant ainsi une consommation inutile d'instances à on-demand coûteuses. Sur cette base, nous proposons de bonnes stratégies paramétriques pour affecter différents types d'instances à des travaux. L'efficacité des politiques proposées est également validée par des simulations approfondies. Dans cette thèse, les emplois à traiter sont supposés être des emplois indépendants malléables. Il convient de noter que les deux questions théoriques seront également une clé pour l'extension des résultats de cette thèse au cas des emplois avec contraintes de précédence.

C Partie I: l'ordonnancement de tâches malléables

Nous présentons maintenant les idées principales de la conception algorithmique pour les tâches malléables.

C.1 Motivation

Nous présentons d'abord le movitation en cours de dériver les résultats relatifs. Les tâches malléables sont une généralisation des tâches de préemption qui peuvent seulement être exécutées sur une machine simple. Dans le passé, les problèmes relatifs de programmer des tâches de préemption sur une machine simple ont été intensivement étudiés [10], [11]. Quand chaque tâche doit être accomplie par une certaine date-butoir, les résultats précédents dans le cas particulier d'une machine ont déjà impliqué cela l'état d'utiliser de façon optimale des machines joue une fonction clé En particulier, la politique d'EDF peut réaliser l'état optimal d'utilisation de ressource, c.-à-d., donné un ensemble de tâches, s'il y a un programme faisable de ces tâches avec des dates-butoirs sur une machine, la politique EDF (Earliest Deadline First) peut produire un programme faisable.dans la conception et l'analyse des algorithmes d'établissement du programme pour plusieurs objectifs [11]. De nombreuses applications de la règle EDF ont été trouvées pour concevoir, par exemple, (rmnum 1) une planification qui permet d'atteindre l'état optimal d'utilisation des ressources, chaque tâche étant en outre associée à une heure de libération [12], (ii) un algorithme exact permettant de minimiser le retard maximal d'une tâche (c'est-à-dire le temps d'achèvement de la tâche moins la date d'échéance) [13], et (iii) un algorithme exact pour les tâches avec des délais afin de minimiser le nombre total de tâches en retard (c'est-à-dire des tâches dont les délais ne sont pas respectés) [14].

De même, nous avons cru que, une l'ordonnancement qui permet d'atteindre un tel état d'utilisation optimale des ressources est également fondamentale pour la ordonnancement de tâches malléables, et peut être bénéfique pour la conception et l'analyse d'algorithmes de ordonnancement. Ici, l'intuition sous-jacente est que, si l'utilisation des ressources n'est pas optimale dans un algorithme, ses performances peuvent être améliorées en utilisant les machines de manière optimale, ce qui permet d'achever davantage de tâches ou de réduire le temps d'achèvement global des tâches. Toutes ces considérations nous ont motivés à développer le cadre théorique de cette thèse.

C.2 L'état optimal de utilisation machine

Le temps est divisé en créneaux discrets et chaque créneau peut contenir un nombre fixe de minutes. Il y a des machines C et un ensemble de tâches $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$. Chaque tâche T_i une charge de travail D_i, une date limite d_i, et une limite de parallélisme k_i; T_i doit être complété avant la fente d_i et il peut utiliser simultanément au plus k_i machines. Laisser $\{\tau_1, \tau_2, \cdots, \tau_L\} = \{d_i \,|\, T_i \in \mathcal{T}\}$, dénotant toutes les échéances des tâches.

Laissez $\mathcal{S}$ dénoter un sous-ensemble arbitraire de $\mathcal{T}$. In this thesis, nous identifions l'état d'utilisation optimale des ressources de plusieurs machines sur lesquelles un ensemble de tâches avec des délais stricts est planifié. L'état est dérivé au-dessous de trois contraintes: dates-butoirs, limites de parallélisme, et la ressource maximum disponible (capacité). Premièrement, nous ignorons la contrainte de capacité et prenons en considération les contraintes de date-butoir et de parallélisme:

(i) chaque tâche T_i peut seulement utiliser les machines dans $[1, d_i]$,

(ii) T_i peut utiliser tout au plus des machines de k_i à une fente.

Sous de telles contraintes, nous définissons la charge de travail maximale de $\mathcal{S}$ pouvant être traitée dans $[\tau_{L-m}+1, \tau_L]$, dénoté par $\lambda_m(\mathcal{S})$ où $m \in [1, L]$; trivialement, $\tau_0 = 0$. De plus, nous prenons en considération la contrainte de capacité:

(iii) il y a seulement des machines de C disponibles,

et définissez la charge de travail maximum de $\mathcal{S}$ qui pourraient être traités dedans $[\tau_{L-m}+1, \tau_L]$, dénoté par $\lambda_m^C(\mathcal{S})$. Notre définition est récursive. Laissez $\lambda_0^C(\mathcal{S}) = 0$ trivialement. Avec la contrainte de capacité, le $\lambda_{m-1}^C(\mathcal{S})$ est la charge de travail maximum de $\mathcal{S}$ qui pourraient être traités dedans $[\tau_{L-(m-1)}+1, \tau_L]$; $\lambda_m^C(\mathcal{S})$ est soit $\lambda_m(\mathcal{S})$ ou la somme de $\lambda_{m-1}^C(\mathcal{S})$ et $C \cdot (\tau_{L-m+1} - \tau_{L-m})$, c'est à dire,

— $\lambda_m^C(\mathcal{S}) \leftarrow \lambda_{m-1}^C(\mathcal{S}) + \min\big\{\lambda_m(\mathcal{S}) - \lambda_{m-1}^C(\mathcal{S}),\, C \cdot (\tau_{L-m+1} - \tau_{L-m})\big\}$.

Nous définissons $\mu_m^C(\mathcal{S}) = \sum_{T_i \in \mathcal{S}} D_i - \lambda_{L-m}^C(\mathcal{S})$ comme charge de travail (minimum) demeurante $\mathcal{S}$ ce doit être traité après que $\mathcal{S}$ ait au maximum utilisé des machines de C dedans $[\tau_m + 1, \tau_L]$ pour tous $m \in [L-1]$. Dans cette thèse, nous appelons les inégalités suivantes la condition aux frontière

$$\mu_m^C(\mathcal{S}) \leq C \cdot \tau_m, \text{ pour tous } m \in [0,\, L-1]$$

L'algorithme optimal d'ordonnancement

Dans ce subsubsection, nous montrons cela, si $\mathcal{S}$ remplit la condition de frontière ci-dessus, alors, là existe un algorithme LDF ($\mathcal{S}$) qui produit un programme faisable de $\mathcal{S}$, réalisant l'état optimal d'utilisation de ressource.

Laissés $\mathcal{S}_m$ dénotent toutes les tâches de $\mathcal{S}$ avec la date-butoir τ_m. À la fente t, nous dénotons le nombre de machines qui ont été assignées aux tâches par $W(t)$; $\overline{W}(t) = C - W(t)$ dénote le nombre de machines disponible à t. Au commencement, $W(t) = 0$ et $\overline{W}(t) = C$. LDF($\mathcal{S}$) fonctionne comme suit:

1. les tâches de $\mathcal{S}$ sont considérées dans l'ordre décroissant de leurs dates-butoirs, i.e., dans les l'ordre de $\mathcal{S}_L$, $\mathcal{S}_{L-1}$, $\cdots$, $\mathcal{S}_1$, là où les tâches dans le même ensemble sont considérées dans l'ordre aléatoire;
2. quand une tâche T_i est considérée, l'algorithme Allocate-B(i) s'appelle pour faire T_i entièrement assigné sous les contraintes de date-butoir et de parallélisme.

Nous prouverons que, seulement si $\mathcal{S}$ remplit la condition de frontière et l'état d'utilisation de ressource de machines satisfait quelques propriétés sur chaque achèvement Allocate-B(i), toutes les tâches dans $\mathcal{S}$ seront entièrement assignées après LDF ($\mathcal{S}$) finit. Dans LDF($\mathcal{S}$), quand une tâche T_i est considérée, supposent que T_i appartient à $\mathcal{S}_m$ et dénotent par $\mathcal{S}' \subseteq \mathcal{S}_L \cup\ cdots \cup \mathcal{S}_m$ les tâches qui ont été entièrement assignées jusqu'ici; les tâches de $\mathcal{S}'$ sont considérées avant T_i. Ici, $\mathcal{S}$ remplit la condition de frontière; tout son sous-ensemble comprenant $\mathcal{S}'$ et $\mathcal{S}' \cup \{T_i\}$ remplissent également la condition de frontière. Avant que l'exécution de Allocate-B(i), nous supposons que l'état d'attribution de ressources satisfait les deux propriétés suivantes.

La première propriété est que, pour les machines C, l'état optimal d'utilisation des ressources est obtenu par l'allocation actuelle à $\mathcal{S}'$.

***Property* C.1.** *Pour toute $l \in [1, L]$, la quantité de la charge de travail de $\mathcal{S}'$ qui est traité dedans $[\tau_{L-l}+1, d]$ est $\lambda_l^C(\mathcal{S}')$.*

La deuxième propriété est que, l'état d'utilisation de ressource dans $[1, \tau_m]$ a une forme faite un pas.

***Property* C.2.** *Si là existe une fente $t \in [1, \tau_m]$ tels que $\overline{W}(t) > 0$, a laissé t_0 soit la dernière fente dans $[1, \tau_m]$ où $\overline{W}(t_0) > 0$; alors nous avons $\overline{W}(1) \geq \overline{W}(2) \geq \cdots \geq \overline{W}(t_0)$.*

Si les deux propriétés ci-dessus sont satisfaisantes, nous montrerons en Chapitre 3.2.2 et 3.2.3 cela, là existe un algorithme Allocate-B(i): après avoir terminé Allocate-B(i), les deux propriétés suivantes sont satisfaites:

***Property* C.3.** *T_i est entièrement assigné.*

***Property* C.4.** *L'allocation de ressources à $\mathcal{S}' \cup \{T_i\}$ satisfait Property C.1 et Property C.2.*

Dans le cas que ce qui précède Allocate-B(i) existe, seulement si $\mathcal{S}$ remplit la condition de frontière, $\mathcal{S}$ peuvent être entièrement assigné par LDF($\mathcal{S}$). La raison de ceci peut être expliquée par induction. Quand la première tâche T_i dans $\mathcal{S}$ est considérée, $\mathcal{S}'$ est vide, et, avant l'exécution Allocate-B(i), Properties C.1 et C.2 sont satisfaits trivialement. De plus, après avoir terminé Allocate-B(i), T_i sera entièrement alloué par Allocate-B(i) en raison de Property C.3, et Property C.4 est conservé. Ensuite, supposons que $\mathcal{S}'$ n'est pas vide et que les Properties C.1 et C.2 soient conservées. Lorsque la tâche T_i est considérée par LDF($\mathcal{S}$), elle est toujours entièrement allouée et les Properties C.3 et C.4 sont conservées après Allocate-B(i) est terminé. Par conséquent, toutes les tâches de $\mathcal{S}$ seront finalement entièrement attribuées à la fin de LDF($\mathcal{S}$).

C.3 Algorithme glouton

Nous associons chaque tâche T_i à une valeur v_i; cette valeur est obtenue lorsque T_i est terminé avant la date limite. Dans cette sous-section, nous proposons un algorithme pour maximiser le bien-être social, c'st-à-dire pour maximiser la somme des valeurs des tâches accomplies avant leur date limite. Dans cette section, nous illustrons l'application des résultats ci-dessus à l'algorithme avide pour la maximisation de bien être social. La forme générale d'un algorithme glouton est la suivante [20], [22]: il tente de construire une solution en exécutant de manière itérative les étapes suivantes jusqu'à ce qu'il ne reste plus aucun élément à prendre en compte: (1) norme de sélection: de manière gourmande, choisissez et considérez une tâche localement optimale en fonction de certains critères; (2) condition de faisabilité: quand une tâche est considérée, acceptez-la si elle remplit une certaine condition et rejetez-la autrement.

Le critère de sélection est lié à la fonction objective et aux contraintes, et est habituellement le rapport du 'avantage' au 'coût'; le ratio mesure l'efficacité d'une tâche. Dans le problème de cette thèse, la contrainte vient de la capacité à tenir les tâches choisies et l'objectif est de maximiser le bien-être social; donc, le critère de sélection ici est le rapport de la valeur d'une tâche à sa charge de travail, appelé la valeur marginale de cette tâche. Formellement, la valeur marginale d'une tâche T_i est définie as $v_i' = \frac{v_i}{D_i}$, i.e., la valeur obtenue à partir de par l'unité de la charge de travail quand T_i est accompli avant sa date-butoir. Etant donné la forme générale d'algorithme avide, nous définissons une classe des algorithmes avides qui fonctionnent comme suit, dénoté par GREEDY:

1. Considère les tâches dans l'ordre décroissant des valeurs marginales; assumez sans perte de généralité cela $v_1' \geq v_2' \geq \cdots \geq v_n'$;
2. Dénoter l'ensemble des tâches qui ont été acceptées par $\mathcal{A}$; quand une tâche T_i est considérée, on l'accepte et est entièrement assigné si là existe un programme faisable pour exécuter $\mathcal{A} \cup \{T_i\}$ sur des C machines.

Dans le suivant, nous nous référons à l'algorithme générique dans GREEDY comme Greedy. Le meilleur rapport d'approximation qu'un algorithme avide dans GREEDY peut réaliser est $\frac{s-1}{s}$.

Greedy considérera des tâches séquentiellement. La première tâche sera acceptée certainement et alors elle emploiera l'condition de faisabilité pour déterminer si accepter ou rejeter la prochaine tâche, selon les ressources disponibles actuelles et les caractéristiques de ceci tâche. Pour décrire le processus dans lequel Greedy accepte ou des tâches de rejets, nous définissons les ensembles de tâches admises et rejetées consécutives $\mathcal{A}_1, \mathcal{R}_1, \mathcal{A}_2, \cdots$. Spécifiquement, laissez $\mathcal{A}_m = \{T_{i_m}, T_{i_m+1}, \cdots, T_{j_m}\}$ être le m-th réglé des tâches adjacentes qui sont acceptées par avide où $i_1 = 1$, tandis $\mathcal{R}_m = \{T_{j_m+1}, \cdots, T_{i_{m+1}-1}\}$ est l'ensemble de m-th des tâches adjacentes qui sont rejetées après l'ensemble $\mathcal{A}_m$, où $m \in [K]^+$ pour un certain nombre entier E. Le nombre entier K représente la dernière étape : dans le K-th étape, $\mathcal{A}_K \neq \emptyset$ et $\mathcal{R}_K$ peut être vide ou non vide. Nous dénotons également par c_m que la date-butoir maximum de toutes les tâches rejetées dans le premier m met en phase, i.e.,

$$c_m = \max_{T_i \in \bigcup_{l=1}^{m} \mathcal{R}_l} \{d_i\},$$

et par c'_m la date-butoir maximum de toutes les tâches admises dans aux premières m-th phases, i.e.,

$$c'_m = \max_{T_i \in \bigcup_{l=1}^{m} \mathcal{A}_l} \{d_i\}.$$

Tandis que les tâches dans $\mathcal{A}_m \cup \mathcal{R}_m$ sont considérées, nous disons cela Greedy est dans la m-th phase. Avant l'exécution Greedy, nous disons cela Greedy est pendant la 0-th phase. À la fin de la m-th phase de Greedy, nous définissons un paramètre B de seuil comme suit

(i) si $c_m \geq c'_m$, laisser $t_m^{\text{th}} = c_m$, et

(ii) si $c_m < c'_m$, laisser t_m^{th} être un fente dans $[c_m, c'_m]$;

Dans cette thèse, nous prouvent que dès que l'attribution de ressources faite par Greedy satisfera quelques caractéristiques, son rapport d'approximation peut être déduit immédiatement. Les caractéristiques sont comme suit:

(i) pour tous $m \in [1, K]$ l'attribution aux tâches admises pendant les premières m phases (i.e., $\cup_{l=1}^{m} \mathcal{A}_l$) réalise une utilisation $\geq r$ in $[1,\, t_m^{\text{th}}]$ où $r \in [0,\, 1]$, et

(ii) la charge de travail maximale de $\cup_{l=1}^{m} \mathcal{A}_l$ est traitée dans $[t_m^{\text{th}} + 1, d]$ après l'achèvement de Greedy, où $d = \max_{T_i \in \mathcal{T}} \{d_i\}$.

La conclusion est comme suit: Greedy réalise un rapport d'approximation r. Puis, en appliquant l'algorithme qui réalise l'état optimal d'utilisation de machine, nous pourrions concevoir un algorithme greedy qui réalise un rapport d'approximation $\frac{s-1}{s}$.

D Partie I: l'ordonnancement de tâches (δ, k)-monotonic

Maintenant, nous expliquons les idées principales pour (δ, k)-monotonic tâches, là où deux objectifs de établissement du programme sont adressés séparément :

réduisez au minimum le makespan et maximisez la somme de valeurs des tâches accomplies par une date-butoir; ce dernier désigné sous le nom de la maximisation de bien être social.

D.1 Minimisation de makespan

Maintenant, nous présentons l'idée principale dans l'algorithme proposé pour la minimisation makespan. Laissez $\gamma(j,d)$ dénoter le nombre minimal de machines requises afin d'accomplir une tâche T_j par le temps d; $t_{j,p}$ est le moment d'exécution où T_j est assigné des p machines. Avant que nous présentions la notion des (δ,k)-monotonic tâches, beaucoup d'efforts avaient jamais fait pour étudier des tâches monotoniques: la charge de travail augmente pendant qu'une tâche est assignée plus de machines, alors que son temps d'exécution diminue ; jusqu'ici, le meilleur résultat est un algorithme de $(\frac{3}{2}+\epsilon)$-approximation [24]. Notre idée pour des (δ,k)-monotonic tâches provient de l'observation suivante pour des tâches monotoniques; ici, nous avons

$$d \geq t_{j,\gamma(j,d)} > \frac{\gamma(j,d)-1}{\gamma(j,d)} \cdot d. \tag{D.1}$$

Supposez que là existe un ordonnancement de toutes les tâches, dénoté par *Sched*, dont makespan est d et cela réalise une utilisation r de ressource dans $[0,d]$; dans la condition que la charge de travail minimum de chaque tâche T_j est traitée (c.-à-d., assigné des $\gamma(j,d)$ processeurs), le ordonnancement *Sched* sera un algorithme de $\frac{1}{r}$-approximation pour la minimisation makespan. La raison est comme suit. Dénotez par d^* le makespan d'un ordonnancement optimal a dénoté par $Sched^*$, où $d^* \leq d$; ainsi, la charge de travail de T_j quand $\gamma(j,d^*)$ processeurs assignés $D_{j,\gamma(j,d^*)} \geq D_{j,\gamma(j,d)}$. Dans le ordonnancement $Sched^*$, la charge de travail de chaque tâche est $\geq D_{j,\gamma(j,d^*)}$ puisque le nombre de processeurs assignés à une tâche T_j est au moins $\gamma(j,d^*)$; ainsi toute la charge de travail de toutes les tâches est $\leq m \cdot d^*$ mais $\geq$ ses homologues dans *Sched* qui est $\geq r \cdot m \cdot d$. Donc, nous déduisons que le makespan optimal d^* est $\geq \frac{r \cdot m \cdot d}{m} = r \cdot d$, c'est-à-dire, $\frac{d}{d^*} \leq \frac{1}{r}$.

Maintenant, seulement si nous pourrions concevoir un tel ordonnancement *Sched* avec une utilisation $r > 2/3$, un algorithme mieux que celui dans [24] pourrait être obtenu. Notre premier problème est de donner le ordonnancement *Sched*; pour réaliser ceci, un défi résulte de l'existence des tâches avec petit $\gamma(j,d)$. En particulier, donné un nombre entier $H \geq 4$, nous appelons les tâches avec $\gamma(j,d) \geq H$ comme tâches avec le grand $\gamma(j,d)$. Chaque tâche avec grand $\gamma(j,d)$ a un temps d'exécution $> \frac{H-1}{H} \cdot d$ par Inequality (D.1) quand $\gamma(j,d)$ processeurs de assignés; ces processeurs ont pu réaliser une utilisation $\geq \frac{H-1}{H} \geq \frac{3}{4}$ dans $[0,d]$. Pour faire face aux tâches avec petit $\gamma(j,d) \leq H-1$, nous présentons la notion des (δ,k)-monotonic tâches où $H-1 \leq \delta$, permettant une classification générique de ces derniers des tâches (voir Chapitre 5.4.1); ici, chaque tâche sera assignée le même nombre de processeurs $(\leq \delta)$ et sa charge de travail minimum est traitée. Nous pouvons proposer ainsi un tel ordonnancement *Sched* dont l'utilisation r rapproche $\frac{H-1}{H}$ (voir Chapitre 5.4.2 et 5.4.3).

Comme est vu plus tard, notre deuxième problème est que l'utilisation de *Sched* peut être dérivée seulement quand les processeurs de m ne sont pas assez pour traiter toutes les tâches par le temps d où quelques tâches sont rejetées; cependant, de ce que nous avons besoin est l'utilisation quand toutes les tâches sont programmées. Pour adresser ceci, laissez U et L être tel que *Sched* peut produire un programme faisable de toutes les tâches par le temps U mais ne fait pas ainsi par le temps L, et nous nous appliquons une procédure de recherche dichotomique à *Sched*. Après que les fins de procédure, pour un tel U et L, nous aient $U \leq L \cdot (1+\epsilon)$ et r dénote l'utilisation de *Sched* quand $d = U$. Après une analyse prolongée de notre observation préliminaire, nous pourrions dériver que le ordonnancement final de toutes les tâches par le temps U est un algorithme de $\frac{1}{r} \cdot (1+\epsilon)$-approximation (voir Chapitre 5.5.1).

D.2 Maximisation de bien-être social

Notre idée pour la maximisation de bien être social est comme suit. Nous donnons d'abord un algorithme (glouton) générique qui définira l'ordre dans lequel des tâches sont acceptées pour l'établissement du programme ; ici, l'algorithme final acceptera seulement une partie de tâches dues à la contrainte de capacité. Alors nous analysons rétrospectivement cet algorithme et définissons quels paramètres détermineront sa performance. En conséquence, puisque la charge de travail minimum de chaque tâche admise est traitée de notre procédure de établissement du programme dans Chapitre 5.4, une application directe de cette procédure à cet algorithme avide mène à un algorithme dont le rapport d'approximation est son utilisation.

E Partie II: utilisation rentable des clouds publics

Maintenant, nous expliquons les idées principales pour utiliser de manière rentable les clouds publics.

E.1 Défis

Dans la présente partie de la thèse, nous faisons l'hypothèse naturelle qui les instances self-owned sont meilleur marché que les instances de spot, qui sont encore meilleur marché que instances de on-demand. Ainsi, pour être coût-optimale, une politique naÃ¯f devrait assigner autant de instances self-owned comme possible, alors instances de spot, et finalement des instances on-demand. C'est, cependant, une tâche difficile. Par exemple, une politique naÃ¯ve pour réaliser une utilisation élevée des instances self-owned serait, quand un travail arrive, d'assigner autant de instances self-owned demeurants comme possible lui. Cependant, cette politique s'avère ne pas être bonne en termes de coût. En effet, elle ignore la différence des travaux et traite tous les travaux également en assignant des instances, tandis que nous constatons qu'une bonne politique en termes de coût doit à la place déterminer les attributions des instances self-owned aux travaux selon leurs capacités d'utiliser seuls des instances de spot pour s'accomplir par des dates-butoirs.

En particulier, quand instances self-owned sont insuffisants, activement assignez instances self-owned aux emploi avec des capacités pauvres et n'assignez rien aux autres ; autrement, de tels emploi pauvres devront consommer instances on-demand plus coûteux, et il cause également un gaspillage des capacités de d'autres emploi riches, ainsi que instances self-owned (même si aucun instances self-owned n'est assigné, ils peuvent être accomplis en utilisant seulement instances spot). Quand instances self-owned sont assez, assignez-les aux emplois avec pauvre et les capacités fortes tels qu'après les attributions tous les emplois sont prévus d'être accompli en utilisant seulement la instances spot, éliminant le besoin de consommer coûteux instances on-demand.

Après attribution instances self-owned, la question gauche est d'identifier la capacité d'un travail pour utiliser la instances spot, c.-à-d., la charge de travail maximum qui pourrait être traitée par instances spot, et propose une politique optimale ordonnancement pour réaliser telles capacités des emplois, eliminating unnecessary consumption of instances on-demand.

E.2 Notre solution

Le temps est les fentes discrètes divisées et chaque fente ont un nombre fixe de minutes. Les travaux arrivent au fil du temps. Chaque travail j a une heure d'arrivée a_j, une taille z_j, une limite du parallélisme δ_j et une date-butoir relative d_j ; le emploi j doit être accompli par la fente $a_j + d_j - 1$. Sur l'arrivée d'un emploi j, l'attribution des instances self-owned est prise et finie immédiatement; l'attribution de sur on-demand and spot instances est prise sur son arrivée et a puis mis à jour chaque heure. Le coût d'utiliser instances self-owned est plus petit que le coût d'utiliser la instances spot, qui est plus petite que le coût d'utiliser instances on-demand; pour être coût-optimal, j utilisera d'abord instances self-owned dans $[a_j, d'_j]$ où $d'_j = a_j + d_j - 1$; puis, si instances self-owned ne sont pas assez pour accomplir j, l'attribution de spot et on-demand instances est prise.

Self-owned instances. Les prix de spot varient au fil du temps; chaque fois que, après que l'attribution de j soit mise à jour, le moment prévu pour lequel j pourrait utiliser des instances spot est $\beta \cdot Len$ où $\beta \in [0, 1]$. Laissez r_j dénotent le nombre instances self-owned assignés à j; pour chaque emploi j, nous irons trouver une fonction $g_j(x) \in [0, \frac{z_j}{d_j}]$ qui satisfait les propriétés suivantes où $\frac{z_j}{d_j} \leq \delta_j$:

***Property* E.1.** *$g_j(x)$ est une fonction non-croissante à mesure que x augmente dans $[0, 1)$.*

***Property* E.2.** *$g_j(\beta)$ est le nombre minimal tels que quand un travail j est assigné r_j instances self-owned dans $[a_j, a_j + d_j - 1]$ où $r_j \geq g_j(\beta)$, il pourrait être prévu cela*

- *emploi j pourrait être accompli par sa date-butoir sans utiliser instances on-demand, si $\delta_j - r_j$ instances spot sont offerts pour à chaque mise à jour de l'attribution à j, où aucun instances on-demand n'est acquis.*

La valeur de $g_j(\beta)$ est un indicateur de la capacité que j doit accomplir lui-même en utilisant seulement la instances spot. Par Property E.2, si $g_j(\beta) \leq 0$, il est prévu qu'aucun self-owned ou on-demand instances n'est nécessaire afin d'accomplir j et de tels travaux ont la capacité forte pour s'alimenter avec la instances spot. Autrement, $g_j(\beta)$ instances self-owned sont nécessaires, ou j doit consommer une certaine quantité instances on-demand chers afin d'être complet elle-même par la date-butoir ; pour un emploi j, une plus grande valeur de $g_j(\beta)$ signifie que une capacité plus faible pour alimenter un emploi j avec la instances spot.

En conséquence, nous pouvons employer la fonction $g_j(x)$ pour commander de manière rentable l'attribution instances self-owned à chaque j. En particulier, sur l'arrivée de j à t, laissez le $N(t)$ dénotent le nombre instances self-owned disponibles à t ; laissez $N_t(d'_j)$ être le minimum de $N(t), \cdots, N(d'_j)$, c.-à-d., le nombre maximum instances self-owned disponibles à chaque fente dans le $[t, d'_j]$. Laissez β_0 être un paramètre et le emploi j est assigné r_j instances self-owned où

$$r_j = min\{g_j(\beta_0), N_t(d'_j)\}.$$

Quand il y a assez instances self-owned, nous pourrions placer β_0 à une valeur plus petit que β et assigner un plus grand nombre instances self-owned à chaque travail j ; puis, instances self-owned seront entièrement utilisés et après l'attribution à j, il pourrait prévoir que les besoins de j seulement d'utiliser la instances spot pour s'accomplir par la date-butoir, éliminant la consommation inutile des instances on-demand coûteux. Quand il n'y a pas assez instances self-owned, nous pourrions placer β_0 à une valeur pas plus petit que β et assigner r_j instances self-owned à j où $r_j \leq g_j(\beta)$; en conséquence, le travail ne consommera pas plus que $g_j(\beta)$ instances self-owned, évitant les déchets des instances self-owned.

Spot and on-demand instances. Après l'attribution instances self-owned, la charge de travail restante dont la taille est $z_j - r_j \cdot d_j$ doit être accomplie par d'_j avec une limite du parallélisme $\delta_j - r_j$. Chaque fois que, après que l'attribution de j soit mise à jour, le moment prévu pour lequel j pourrait utiliser des instances spot est $\beta \cdot Len$; de plus, nous pouvons dériver la quantité prévue de instances spot qui pourrait être utilisée par j et la quantité minimum instances on-demand requis afin de satisfaire la contrainte de date-butoir. Basé sur ceci, nous pourrions concevoir une politique coût-optimale ordonnancement pour utiliser la instances spot et instances on-demand.

F Remarques finales

F.1 Conclusion

Le problème de la ordonnancement et de la tarification est au coeur du domaine du cloud computing, étant donné que l'utilisation optimale des ressources et la maximisation de l'utilité sont souvent la principale préoccupation des fournisseurs et des utilisateurs de cloud. Dans cette thèse, pour le modèle fondamental des tâches malléables, nous sommes les premiers à identifier l'état optimal, dans lequel plusieurs machines sont utilisées de manière optimale par un ensemble de tâches malléables avec des délais, et propose le premier algorithme de ordonnancement optimal pour atteindre l'état optimal. Son importance peut être perçue par le fait suivant: si l'état d'utilisation des ressources n'est pas optimal dans un algorithme, ses performances peuvent être améliorées en utilisant les machines de manière optimale, ce qui permet de terminer davantage de tâches ou de réduire les temps de réalisation globaux des tâches. Les résultats ci-dessus ont fourni un outil conceptuel pour proposer de nouvelles conceptions d'algorithmes ou améliorer les algorithmes existants. Ces algorithmes d'ordonnancement améliorent la qualité de service du fournisseur de cloud ou permettent au cloud de servir plus de travaux, augmentant ainsi les revenus.

Pour les tâches malléables, l'accélération est linéaire: la charge de travail d'une tâche reste constante lorsque le nombre de processeurs sur lesquels exécuter cette tâche est petit et ne dépasse pas une limite de parallélisme. Pour les tâches malléables, l'accélération est linéaire: la charge de travail d'une tâche reste constante lorsque le nombre p de processeurs sur lesquels exécuter cette tâche est petit et ne dépasse pas une limite de parallélisme.

Nous considérons également un modèle plus général qui incorpore le cas où une tâche est affectée à un plus grand nombre de processeurs: lorsque p dépasse la limite de parallélisme, la charge de travail d'une tâche augmente mais son temps d'exécution diminue à mesure que p augmente; cependant, il existe une valeur seuil que le nombre de machines affectées à une tâche ne peut pas dépasser. Dans cette thèse, nous proposons des algorithmes de planification pour minimiser le makespan ou pour maximiser la somme des valeurs des tâches accomplies avant une date limite.

D'autre part, en ce qui concerne la tarification, nous avons étudié les moyens économiques d'utiliser des instances self-owned et des instances (spot, on-demand) de clouds publics tels qu'Amazon EC2. La charge de travail à traiter est supposée être une tâche indépendante et malléable, pour laquelle il existe une contrainte de temps, et nous proposons des les politiques paramétriques rentables pour affecter différents types d'instances à des tâches. L'efficacité de ces politiques est également validée par des simulations. Ici, nous identifions et abordons deux questions sous-jacentes dans le processus d'allocation d'instance: pour être rentable, quelles propriétés doivent être conservées dans la stratégie d'allocation d'instances self-owned et quelle stratégie peut maximiser l'utilisation de d'instances spot. Cela nous permet également d'étendre les travaux de cette thèse au cas suivant: il existe une préséance entre les tâches, ce qui constituera l'un des travaux futurs.

F.2 Perspectives d'avenir

Cette thèse se concentre sur l'amélioration de représentation pour opacifier des systèmes et les manières rentables pour que l'utilisateur utilise les ressources informatiques sous les options actuelles d'achat. Il reste encore beaucoup de travaux futurs à faire.

En ce qui concerne la tarification, l'IaaS est un moyen de créer de la valeur pour les utilisateurs en facilitant leur accès aux capacités de calcul sans la propriété et la maintenance des infrastructures. La conception du service IaaS obéit aux principes fondamentaux de la conception de service: planifier et organiser les personnes, l'infrastructure, la communication et les composants matériels du service, visant l'excellence selon trois dimensions: la qualité du service, l'efficacité du système et l'interaction entre le service fournisseur et ses utilisateurs. Cela nécessite un effort commun pour comprendre les besoins des clients et mettre en place une gestion efficace des ressources.

Avec cette thèse, nos travaux sur la ordonnancement nous ont permis de mieux comprendre quelles caractéristiques des tâches peuvent conduire à une efficacité élevée des ressources, tandis que nos travaux sur l'utilisation des clouds publics nous permettaient de mieux comprendre quels modèles de tarification sont faciles à utiliser. Les modèles de tarification dans Amazon EC2 définissent le processus selon lequel les utilisateurs peuvent acquérir des ressources informatiques et ne sont pas très conviviaux, comme l'indiquent certains travaux [65], [76], [86]. Actuellement, nous examinions quelles formes de service informatique devraient être offertes aux utilisateurs par le cloud afin que les services proposés puissent être plus conviviaux, tout en maximisant les revenus du cloud; ici, les formes de services doivent être liées à un système de gestion des ressources correspondant utilisé dans le système en nuage. Nous avons cru que, la QoS-différenciation est une lentille importante afin d'allouer efficacement les ressources partagées dans beaucoup de domaines là que (i) les utilisateurs ont des préférences potentiellement diverses pour QoS et (ii) les différentes préférences prennent des effets différentiables sur l'efficacité de ressource; pour ce faire, une tarification appropriée est nécessaire pour inciter les utilisateurs potentiels à exprimer leur demande/leurs emplois sous des formes permettant d'optimiser la puissance de la différenciation QoS dans la gestion des ressources.

Par exemple, au moment de soumettre cette thèse, nous avons achevé l'analyse des performances d'un système de tarification différencié en fonction de la QoS dans le cloud computing via une approche analytique, le cloud offrant plusieurs classes de QoS: les emplois de chaque classe seront complétés par un temps d'attente fini; plus le temps d'attente est petit, plus le prix est élevé; nos simulations numériques ont prouvé que l'architecture proposée pourrait a amélioré le revenu d'un fournisseur de nuage jusqu'à de 500% [87]. D'autres exemples de QoS-différenciation comprenant exécuter deux types de charges de travail sur les mêmes serveurs: un type de charges de travail a une condition définie de retard tandis que l'autre exige seulement le service de meilleur-effort, ce qui est utilisé généralement dans des systèmes informatiques privés traditionnels pour améliorer l'efficacité de ressource considérablement. Dans le contexte du cloud computing, il convient d'intégrer l'aspect tarification des ressources partagées; ici, le modèle de gestion des ressources est très similaire au modèle Amazon EC2, dans lequel des instances tourner au ralenti du marché à la demande sont vendues comme des instances spot pour améliorer l'efficacité des ressources [88]. Cette forme de différenciation de QoS est également une direction prometteuse qui reste à adresser à l'avenir.