國立臺灣大學電機資訊學院資訊工程學系

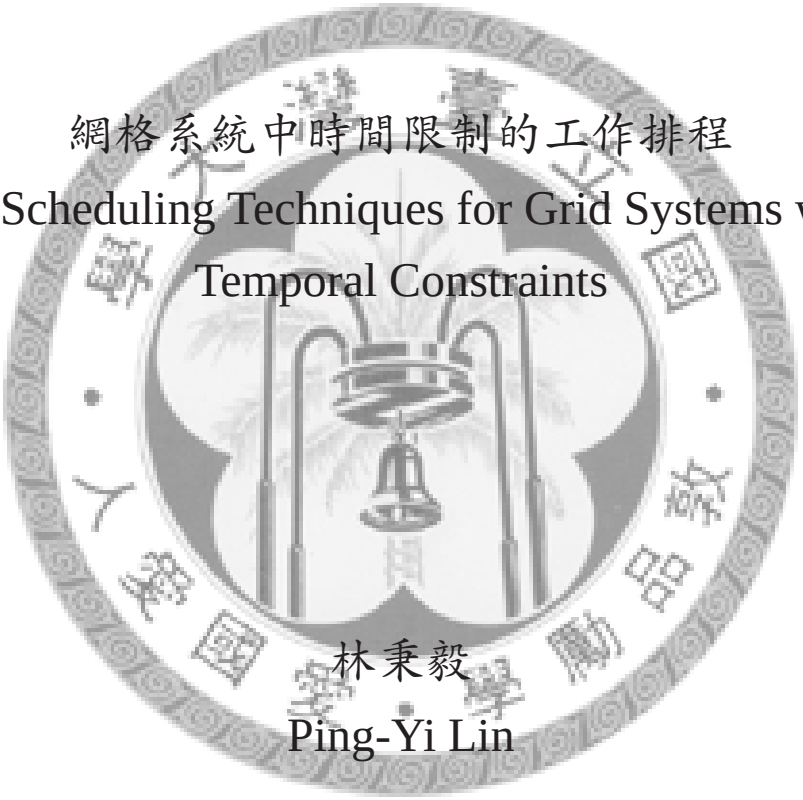
碩士論文

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

網格系統中時間限制的工作排程

Job Scheduling Techniques for Grid Systems with Temporal Constraints



林秉毅
Ping-Yi Lin

指導教授：劉邦鋒 博士
Advisor: Pangfeng Liu, Ph.D.

中華民國98年7月

July 2009

誌謝

首先感謝指導教授劉邦鋒老師的細心指導。自大四起的三年間，老師平時與學生們談笑風生，做研究時一絲不苟的認眞態度，在在都是學生的榜樣。偶爾對學生講解人生道理，或是做人處事的方式，學生必將銘記在心，在未來將這段時間所學好好應用。

感謝口試委員們的指教，以及口試時給予學生的各種建議。感謝實驗室的學長們，在剛開始懵懂的做研究時，給予學生鑽研的方向，並且在上台報告前與學生討論，使學生能夠更爲透徹了解，並且有信心的上台報告。

感謝實驗室同一屆的同學們。感謝秉誼，某些相同的樂趣讓我偶爾可以與你聊天解悶。感謝德禹，幾次的合作報告都讓我受益良多。感謝紘睿在論文寫作階段時的多次幫忙。感謝揚儒，從大學以來的一路陪伴，不知所措時讓我訴苦，茫然無助時對我伸出援手，你會是我一輩子的好朋友。

感謝尋云手語社的所有朋友們。手語社是我大學以來生活的重要歸屬，當我心情煩悶或是研究不順時，總是可以讓我暫時忘卻煩憂。感謝大學以及研究所時期認識的所有朋友們，偶爾見面時的鼓勵總是讓人窩心。

感謝我的父母，將我扶養長大，讓我安心求學，總是在遠方擔心我是否吃飽穿暖，給予我最大的付出與最多的愛。感謝我的姊姊，三不五時的捎來關心的訊息。

感謝所有所有的人，我會滿懷感謝地，向著未來繼續前進。

摘要

這篇論文介紹了將具有順序限制的工作排程到可使用時間被分割成一段段的處 理器上的方法。我們討論了兩種順序限制的狀況—樹狀及鏈狀。我們證明在樹狀限 制下的工作排程是一個NPC的問題，接著我們對於線性順序的工作排程提出動態規 劃演算法取得最佳解。該動態規劃演算法可以被推廣到異質環境中，亦可推廣以解 決似樹狀結構的工作排程問題。而爲了減少排程的時間，我們提出了三種不同的演 算法。實驗結果顯示，我們的三種演算法即使與以動態規劃取得的最佳解比較，仍 然可以得到近似最佳的排程。

關鍵字　時間限制、工作順序、動態規劃

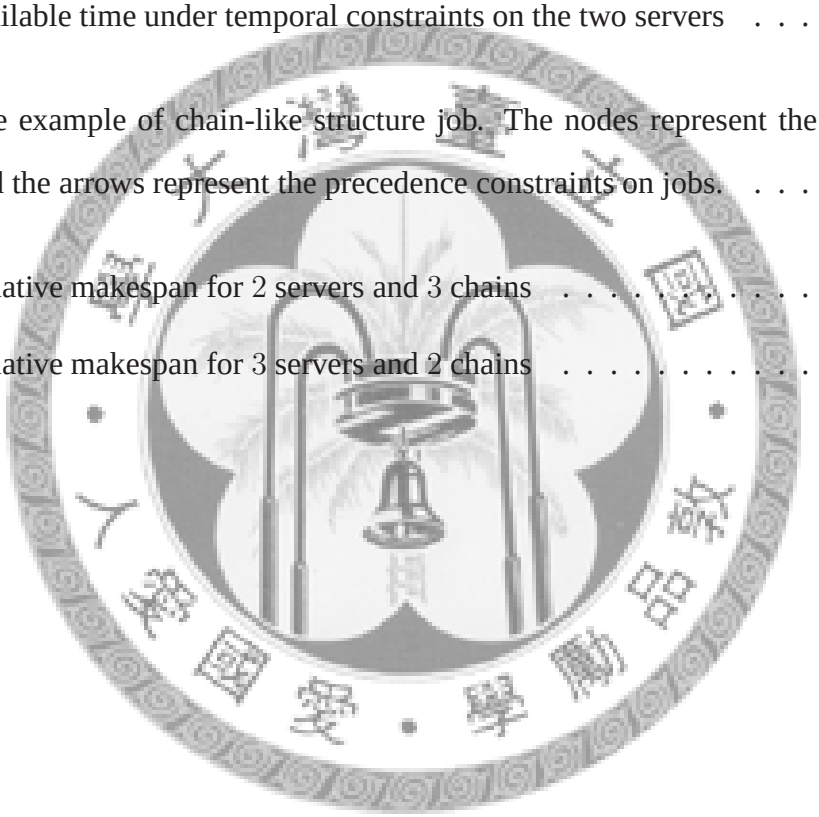
Abstract

This paper introduces techniques in scheduling jobs with dependency constraint to processors whose available time are fragmented into time slots. We discuss two job dependency patterns – tree and chains. We show that it is NP-complete to schedule jobs with a tree dependency pattern. Then we propose a dynamic programming algorithm to get the optimal schedule for assigning jobs with linear dependency. The dynamic programming can be generalized to heterogeneous environment and tree-like dependency structure. In order to reduce the time of scheduling we also propose three different heuristics. Experimental results indicate that these heuristics do provide near optimal schedules even when compared against the optimal solution found by the dynamic programming.

Keywords Temporal constraints, Job dependency, dynamic programming.

Contents

List of Figures



List of Tables

Chapter 1

Introduction

Grid computing is an important mechanism for utilizing computing resources that are distributed in different locations but organized into an integrated service. Specifically a grid system provides computing resources that enable users in different locations to utilize the CPU cycles of remote sites. In addition, users can access important data that are only available in certain locations, without the overheads of replicating them locally. These services are provided by an integrated grid service platform, which helps users access resources easily and effectively.

The primary objective of most existing Grid systems is to improve overall system performance, therefore the quality of service experienced by Grid users is of secondary consideration. In order to improve quality of service various scheduling techniques have been proposed, including advance reservation of resources.

Advance reservation of resources is one mechanism that is used to satisfy the quality-of-service requirements in Grid systems. Advance reservation is the ability of Grid scheduler to guarantee the availability of resources at a particular time in the future. Advance reservation increases the predictability of the system in order to match quality-of-service requirements imposed by users. Although advance reservation has such benefits, it tends to fragment the available resources and lead to poor utilization and system performance, which should be taken into account while we apply advance reservation scheduling techniques.

Another factor that should be taken into account while scheduling job in Grid systems is autonomous site policy. A Grid system usually consists of autonomous sites, and each of them have its own management policy. One important principle of grid system is *not* to interfere with local site policy. For example, a site may determine that it will only provide CPU at night and reserve the CPU during office hours for local users. The global grid scheduling must respect this local policy in order to maintain an integrated image of grid services.

Advance reservation and grid site local policy will fragment to available time of grid services. This paper focuses on this *temporal constraint*, by which we mean the available time of resource is not continuous in time domain. Rather there are time slots that we cannot utilize due to advance reservation or local policy. As a result our scheduling algorithm must take this temporal constraint into consideration, and develop new algorithm without assuming that the resource is available at all time.

As for job characteristics this paper focuses on jobs with linear dependency. A group of jobs is scheduled to run and the i-th job cannot start until the $i-1$-th job finishes. This linear dependency follows the logical consequence of the computation. For example, a computation phase cannot start until the preprocessing finishes, or the data cannot be rendered visually until all the pixels are given the correct values. The scheduling problem for linear dependency jobs is very difficult even without temporal constraints on processors – it is shown to be an NP-complete problem by Du et al. [1]

In this paper we proposed a dynamic programming algorithm for the scheduling problem under temporal constraint. We also propose efficient heuristic algorithms to achieve good solutions. Experimental results from the heuristic algorithm show that the proposed heuristic algorithms is scalable, and is close to the optimal solution.

The rest of the paper is organized as follows. Chapter 2 reviews related works on scheduling problems under other job dependency constraints, and works under temporal constraints on processor clusters. Chapter 3 describes our system model in details.

Chapter 4 and Chapter 5 describes our dynamic programming algorithm and heuristic algorithms for the scheduling problem. Chapter 6 gives experimental results and demonstrates the solution quality by comparing our heuristic algorithms to the optimal solution. Chapter 7 gives conclusions and discusses possible future works.

Chapter 2

Related Works

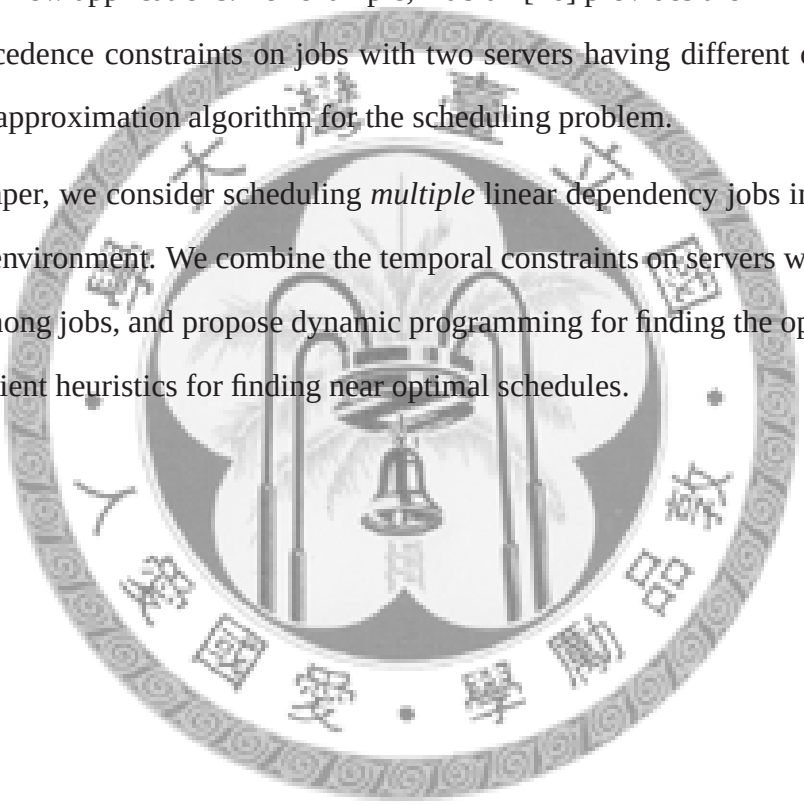
Scheduling job for execution for parallel and distributed systems has been studied for years and proven to be a very difficult problem. General scheduling problems on multiprocessor system have been shown to be NP-complete [2]. Even if there are only two processors with identical computing capacity and no data dependency between jobs, the problem remains NP-complete because it is a special case of 2-partition problem [3]. Du et. al. [1] proves that even for two processors with the same capacity, the scheduling problem remains NP-hard. Many heuristic algorithms have been proposed to achieve better performance for scheduling independent tasks in heterogeneous distributed systems. Braun et. al. [4] compares the results of eleven different heuristic algorithms for this problem under identical system assumption.

Resource management policy is very important in meeting QoS requirement of users and achieving better performance when scheduling jobs on a parallel and distributed system. For example, *advance reservation* of resources is a mechanism that has been studied in [5, 6, 7, 8]. Previous works [9, 10] also give theoretical proofs that advance reservations indeed improves the performance predictability. However advance reservation does not scale well in these previous works, thus Castillo et al. [11, 12] uses concepts from computational geometry to propose scheduling algorithms that scale well in large Grid systems. Singh et al. [13] also propose a multi-objective genetic algorithm formulation for selecting the set of resources to be provisioned that optimizes the application perfor-

mance while minimizing the resource costs.

Most works mentioned above focus on *independent* job scheduling, however workflow applications have become popular in recent years because of the development of Grid computing. A workflow application can be represented as a directed acyclic graph (DAG), where the nodes represent individual jobs and the edges represent job dependencies. The challenge of scheduling workflow applications is discussed in [14], and several heuristics have been proposed and summarized in [15]. There are also results considering special cases of workflow applications. For example, Kubiak [16] provides the NP-hard proof of tree-like precedence constraints on jobs with two servers having different capacity, and proposes an approximation algorithm for the scheduling problem.

In this paper, we consider scheduling *multiple* linear dependency jobs in a time constraint Grid environment. We combine the temporal constraints on servers with linear dependency among jobs, and propose dynamic programming for finding the optimal schedule, and efficient heuristics for finding near optimal schedules.



Chapter 3

Problem Description

In this chapter we formally define the system model for scheduling multiple linear dependency jobs on temporal constraint grid systems.

The grid system model is as follow. We have a Grid system G with m processors $p_1, \ldots, p_m$. All processors are identical in terms of their processing capability. The available time of processors is fragmented, and we use *available* periods to denote time periods that can run jobs. We use I to denote the set of available periods, and each element $I_{i,j}$ in I indicates the j-th available periods on processor p_i. Each $I_{i,j}$ is a two-parameter tuple ($s_{i,j}$, $e_{i,j}$), where $s_{i,j}$ is the *starting time* of the available period, and $e_{i,j}$ is the *ending time* of the available period.

The system has n *job chains* $H_1, \ldots, H_n$. A job chain $H_i = \{J_{i,1}, \ldots, J_{i,h_i}\}$ is a chain of jobs that follows linear dependency, where h_i is the number of jobs in job chain H_i, and $J_{i,k}$ is the k-th job within job chain H_i. The linear dependency mandates that $J_{i,j}$ cannot start until $J_{i,j-1}$ finishes. We also use $t_{i,j}$ to denote the execution time job $J_{i,j}$. The system does not allow preemption, therefore once a job starts on a processor its execution will not be interrupted.

We summarize the notations used in the model in Table 3.1.

Notation	Description
G	the Grid system G
m	the amount of processors
p_i	the i-th processor
I	available period set
$I_{i,j}$	the j-th available periods on i-th processor
$s_{i,j}$	starting time of the available period $I_{i,j}$
$e_{i,j}$	ending time of the available period $I_{i,j}$
H	job chain set
n	the amount of job chains
H_i	the i-th job chain
h_i	amount of jobs of the i-th job chain
$J_{i,j}$	the j-th job of the i-th job chain
$t_{i,j}$	the processing time of $J_{i,j}$
S	a schedule

Table 3.1: Notations used in the system model

3.1 Schedule

A scheduler assigns jobs to processors without violating *temporal constraints* on processors and *linear dependency* on jobs. That is, we can only run a job during the available periods of a processor and we can only do so after we have finished its predecessor in the job chain some time before. The scheduler must determine a *schedule* S that maps jobs to processors in their available time. The goal is to determine a schedule that minimizes the total makespan, i.e., the time for the last job to finish.

When there is only one job chain H_1, we can use a simple greedy algorithm to obtain the optimal schedule. We consider each job on H_1 according to the order they appear in the chain, starting from the $J_{1,1}$. We choose the processor (and the time slot) that can finish $J_{1,1}$ at the earliest possible time. Then since the second job $J_{1,2}$ must wait until the first job $J_{1,1}$ finishes, we adjust the starting time of the time slots that overlap with the execution of $J_{1,1}$, and remove time slots that end before $J_{1,1}$ finishes. Then we assign the second job $J_{1,2}$ to a processor (and a time slot) that can finish $J_{1,2}$ at the earliest possible time. We repeat this process until all jobs are scheduled. It is easy to see that this greedy schedule is optimal.

We then consider more general job dependency workflow. First when there are more

than one chain this scheduling problem becomes NP-complete [1]. Second if the job dependency forms a tree and processors have temporal constraints, the scheduling problem is also NP-complete, as Theorem 1 suggests.

Theorem 1. *The scheduling problem for jobs having tree dependency and processors having temporal constraint is NP-complete.*

Proof. This scheduling problem is in NP since a non-deterministic Turing machine can guess a schedule consisting of the starting time of all jobs, and verify whether the job dependency constraint and the processor temporal constraint are met in polynomial time.

We prove the NP-completeness of the scheduling problem by reducing 2-partition to this scheduling problem. A problem instance I from the 2-partition problem has k numbers $n_1, \ldots, n_k$, and we would like to know whether it is possible to partition the numbers into two groups of equal sum. That is, each group has a sum $\frac{\sum_{i=1}^{k} n_i}{2} = B$.

Given a problem instance of 2-partition, we construct an instance of our scheduling problem as follows. We construct a skewed binary tree of $2k$ nodes as in Figure 3.1. This tree consists of main "chain" of k jobs, each with execution time 1. These jobs will be denoted to as *light* jobs. And from each of the light job we grow a *heavy* job of execution time mn_i, where $m > k$. Figure 3.1 (a) shows an example of the constructed tree. The white nodes are light jobs with execution time 1, and the black ones are heavy jobs with execution time taken from the 2-partition problem instance.

We would like to schedule these jobs into two processors – server 1 and server 2. Server 2 has two available time – $(0, k)$ and $(mB, 2mB)$, and server 1 has only one available time $(mB, 2mB)$. Recall that $B = \frac{\sum_{i=1}^{k} n_i}{2}$, which is half of the sum of n_i in the 2-partition problem instance I. We would like to know whether the jobs can finish before time $2mB$, and we will refer to this problem instance as I'. Figure 3.1 (b) shows an example of the available time slots we constructed.

It is easy to see that if there is a solution for 2-partition problem instance I, there is a solution for our scheduling problem instance I'. All the light jobs on the main chain will

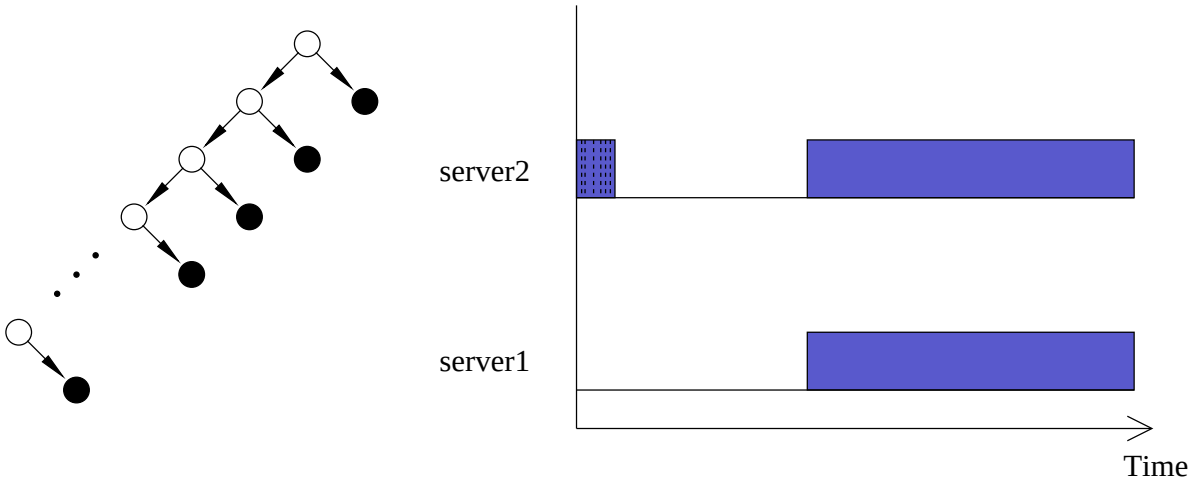


Figure 3.1: (a) An example of the constructed job dependency tree (b) An example of available time under temporal constraints on the two servers

be assigned to server 2. The available time $(0, k)$ is sufficient because there are k light jobs and each of them has execution time 1. The heavy jobs that grow out of the main chain will be assigned to either server 1 or 2, according to the solution from the 2-partition problem instance I. Since this solution of I partitions the numbers of two groups, each has a sum B, the heavy jobs will fit into the two available time slots.

Now we need to show that if there is a solution for I' then there is a solution for I. We first observe that no heavy job can fit into the first available time interval $(0, k)$ since $m > k$, so the first interval has all the light jobs. That also means all the heavy jobs must go into the two $(mB, 2mB)$ available time intervals. We also observe that there will be no precedence constraints among heavy jobs, since all the light jobs finish before time $mB > k$, therefore we can consider heavy jobs as independent. Also the sum of the lengths of the two intervals is exactly the total heavy job execution time, they must be precisely packed into these two intervals. As a result we can use the mapping from heavy jobs to processors as a solution to I. The theorem follows. ■

In this paper, we mainly focus on chain precedence constraint. We will propose a dynamic programming algorithm to find the optimal schedule, and efficient heuristics that find near optimal solutions.

Chapter 4

Dynamic Programming

In this chapter we present our dynamic programming for scheduling job chains in systems that have temporal constraints, then we propose efficient heuristic algorithms that achieve near optimal schedule.

4.1 Dynamic Programming Algorithm

The dynamic programming schedules jobs in phases. In each phase, the algorithm chooses one job and adds it into the "partial" schedule of this phase. For ease of explanation we use the *current schedule* to denote the partial schedule at this point. At the end the schedule will have assigned all jobs to processors.

We define *ready time* for a job chain, and *ready time sequence* for a set of job chains. The ready time of a job chain H_1, denoted by R_{H_1}, is the completion time of the last job being processed in a job chain. The ready time sequence of the set of job chains is the sequence of ready time all job chains in that set, and is denoted as $R_H = (R_{H_1}, \ldots, R_{H_n})$. Note that the ready time and ready time sequence are defined over the current schedule, and will change after a new job is added into the current schedule in every phase.

We define *last completion time* of a processor and *last completion time sequence* for all processors. The last completion time of a processors p is the completion time of the last job assigned to p, and is denoted as L_p. Similar we can define *last completion time*

sequence as the sequence of the last completion time of all processors, which is denoted as $L_G = (L_{p_1}, \ldots, L_{p_m})$. Note that if we assign job $J_{i,j}$ to processor p_k in the current phase, both the ready time of job chain H_i and the last completion time of processor p_k will be updated to the completion time of job $J_{i,j}$.

Finally we define the *finished job* vector for job chains. Let $U = (u_1, \ldots, u_n)$ be a finished job vector such that u_i is the numbers of *finished* jobs in job chain H_i.

Now we are ready to present our dynamic programming in three parts – table element definition, recursive formula, and the computing sequence.

4.1.1 Table Element Definition

We now define the table elements in our dynamic programming. Let L_G be a last completion time sequence for a grid system G and R_H be a ready time sequence for a job chain set H, and U be a finished job vector. We define $E(L_G, R_H, U)$ to be the *minimum* makespan for a last completion time sequence L_G and ready time sequence R_H, and a finished job vector U. By definition the optimal makespan of a given job chain set H for a processor set G is $E((0, \ldots, 0), (0, \ldots, 0), (0, \ldots, 0))$, that is, the last completion time of every processor is 0, the ready time of every job chain is 0, and the numbers of finished job of every job chain is 0.

4.1.2 Recursive Formula

We now describe how to update a last completion time sequence L_G and a ready time sequence R_H when we schedule a new job $J_{i,j}$ to processor p_k. First we note that only L_{p_k} in L_G and R_{H_i} in R_H need to be updated, i.e., the completion time corresponding to processor p_k in L_G and the ready time corresponding to job $J_{i,j}$ in job chain H_i. The new value for both L_{p_k} and R_{H_i} is the completion time of $J_{i,j}$, which can be determined by assigning $J_{i,j}$ to the *first* available time slot period, i.e. minimum l, such that the time slot $I_{k,l}$ on p_k satisfies the condition that $e_{k,l} - t_{i,j} \geq \max(s_{k,l}, L_{p_k}, R_{H_i})$. Recall that $e_{k,l}$ and

$s_{k,l}$ are the starting and ending time of time slot $I_{k,l}$. Formally we use $x(L_{p_k}, R_{H_i}, J_{i,j})$ to denote this minimum index l for time slots on processor k.

$$x(L_{p_k}, R_{H_i}, J_{i,j}) = \min\{l | e_{k,l} - t_{i,j} \geq \max(s_{k,l}, L_{p_k}, R_{H_i})\} \tag{4.1}$$

We describe the reasons of Equation 4.1 as follows.

1. The length of the available time slot must be large enough to run the job, i.e, $e_{k,l} - t_{i,j} \geq s_{k,l}$.
2. The starting time of job $J_{i,j}$ cannot be earlier than the completion time of its predecessor $J_{i,j-1}$ because of linear dependency within a job chain, therefore we have $e_{k,l} - t_{i,j} \geq R_{H_i}$.
3. We only consider time slots, or portion of time slot on processor p_k that are *after* L_{p_k}, the last completion time of p_k, and we have $e_{k,l} - t_{i,j} \geq L_{p_k}$.

After assigning a job of a job chain to a time slot of a processor we need to update the last completion time of that processor and the ready time of that job chain. Formally after assigning job $J_{i,j}$ to time slot $I_{k,l}$ on processor p_k we update the last completion time L_{p_k} for p_k and the ready time R_{H_i} for job chain H_i. For ease of notation we use l^* to denote the time slot index $x(L_{p_k}, R_{H_i}, J_{i,j})$ from Equation 4.1. Then the starting time of $J_{i,j}$ will be $\max(s_{k,l^*}, L_{p_k}, R_{H_i})$, and the processing time $t_{i,j}$, the completion time of $J_{i,j}$ will be $\max(s_{k,l^*}, L_{p_k}, R_{H_i}) + t_{i,j}$. Therefore, we update both L_{p_k} and R_{H_i} to $\max(s_{k,l^*}, L_{p_k}, R_{H_i}) + t_{i,j}$.

We now use a function $T(L_G, R_H, J_{i,j}, p_k) = (L'_G, R'_H)$ to describe the new last completion time and the new ready time after we assign $J_{i,j}$ to a processor p_k based on the old last completion time sequence L_G and the old ready time sequence R_H. That is, L_G and R_H are the last completion time sequence and the ready time sequence before the scheduling, and L'_G and R'_H are ones after the scheduling respectively. The definition of L'_G and R'_H are as follows.

$$l^* = \min\{l | e_{k,l} - t_{i,j} \geq \max(s_{k,l}, L_{p_k}, R_{H_i})\} \quad (4.2)$$

$$L'_{p_v} = \begin{cases} L_{p_v}, & \text{if } v \neq k \\ \max(s_{k,l^*}, L_{p_k}, R_{H_i}) + t_{i,j}, & \text{if } v = k \end{cases} \quad (4.3)$$

$$R'_{H_v} = \begin{cases} R_{H_v}, & \text{if } v \neq i \\ \max(s_{k,l^*}, L_{p_k}, R_{H_i}) + t_{i,j}, & \text{if } v = i \end{cases} \quad (4.4)$$

Now we describe how to update the finished job vector U after assigning a new job to a time slot. Note that in each round, we can only choose the first remaining job in each chain to execute because of the linear dependency, i.e., jobs $J_{1,u_1+1}, \ldots, J_{n,u_n+1}$. For example, if the remaining job vector is $(3, 0, 2)$, then we can only choose job $J_{1,4}$, $J_{2,1}$, or $J_{3,3}$ in this round. Therefore we define an increase-by-1 function $Q(U, e) = (u'_1, \ldots, u'_n)$ as follows.

$$Q(U, e) = (u'_1, \ldots, u'_n) \quad (4.5)$$

$$u'_i = u_i, \text{ if } i \neq e \quad (4.6)$$

$$u'_i = u_i + 1, \text{ if } i = e \quad (4.7)$$

Now we present the recursive formula for the optimal makespan function $E(L_G, R_H, U)$ as Equation 4.8. We consider the first remaining job of all job chains and all processors p_k in G, assign the job to the processor, and choose one that will in term has the minimum makespan.

$$E(L_G, R_H, U) = \min_{p_k \in G} \min_i E(T(L_G, R_H, J_{i,j}, p_k), Q(U, j)), \text{ s.t. } j = u_i + 1 < h_i \quad (4.8)$$

The terminal condition for $E(L_G, R_H, H)$ is when all jobs are scheduled. Formally we have $u_i = h_i, 1 \leq i \leq n$, which means the number of finished jobs is equal to the number of jobs in a job chain. In these cases the makespan is the maximum ready time within the ready time sequence R_H. Note that this maximum ready time is also equal to the maximum last completion time within the last completion time sequence L_G.

$$E(L_G, R_H, (h_1, \ldots, h_n)) = \max_{i \le n} R_{H_i} = \max_{k \le m} L_{p_k} \tag{4.9}$$

4.1.3 Time Complexity

We now analyze the time complexity of the dynamic programming. We assume that there are m processors, n job chains, and k jobs, where $k = \sum_{i=1}^{n} h_i$.

We analyze the time complexity of the dynamic programming by counting the number of terminal cases. One can think of the dynamic programming as a tree where $E((0, \ldots, 0), (0, \ldots, 0), (0, \ldots, 0))$ is the root and the terminal cases are leaves, therefore the time complexity is proportional to the number of edges in this tree. Since the number of edges in the tree is roughly the number of nodes, and the number of nodes is bounded by a multiple of the number of leaves, we can bound the execution time by the number of leaves in this tree.

We count the number of terminal cases by first fixing a sequence S of scheduling jobs and consider the number of ready time sequences and last completion time sequences S could derive. Since a job $J_{i,j}$ can be allocated to any processor, regardless the position it is in the sequence S, we have m choices to allocate $J_{i,j}$. Thus a given sequence S has a total number of m^k to allocate all jobs.

Now we consider all possible job scheduling sequences. Because of the precedence constraint on jobs, jobs on the same chain must appear in order, therefore jobs in chain H_1 must appear in S like $(\ldots, J_{1,1}, \ldots, J_{1,2}, \ldots, J_{1,n}, \ldots)$. Therefore we have $\frac{k!}{\prod_{i=1}^{n} h_i!}$ different schedule sequences for all jobs in H, and the total number of terminal cases for all S is $m^k \frac{k!}{\prod_{i=1}^{n} h_i!}$.

From the calculation in Equation 4.10 we conclude that the time complexity of the dynamic programming is $O(n^k m^k)$. The last inequality follows from the fact that $\frac{(k!)}{((\frac{k}{n}!))^n n^k} < 1$.

$$m^k \frac{(k!)}{\prod_{i=1}^{n}(h_i!)} \leq m^k \frac{(k!)}{\prod_{i=1}^{n}(\frac{k}{n}!)} \leq m^k \frac{(k!)n^k}{((\frac{k}{n}!))^n n^k} = O(n^k m^k) \tag{4.10}$$

4.1.4 Generalization

Heterogeneous processors Our dynamic programming can be generalized to heterogeneous processor environments in which the computation speed of processors are different. We just need to take the speed of processors into consideration in the dynamic programming algorithm and slight modify the definition of the processing time of jobs. We describe the details as follow.

We now review the speed of processors and execution time in the heterogeneous processor model. Without lose of generality we assume that the slowest processor has speed 1, and the speed of processor p_k is c_k. The execution time of job $J_{i,j}$ ($t_{i,j}$) is the time it takes to run $J_{i,j}$ on the slowest processor, therefore the execution time of $J_{i,j}$ on processor p_k now becomes $\frac{t_{i,j}}{c_k}$. We modify Equation 4.1 accordingly as follows.

$$x(L_{p_k}, R_{H_i}, J_{i,j}) = \min\{l | e_{k,l} - \frac{t_{i,j}}{c_k} \geq \max(s_{k,l}, L_{p_k}, R_{H_i})\} \tag{4.11}$$

The dynamic programming can now solve the scheduling problem under heterogeneous environment. We use Equation 4.2 and Equation 4.5 to update the last completion time sequence, the ready time sequence, and the fished job vector. Then we use the recursive formula in Equation 4.8 to compute the makespan function under the same terminal conditions.

Chain-like structure Our dynamic programming can also be generalized for certain chain-like workflow problem. For example, we have a root node job J_s with n job chain children $\{H_1, \ldots, H_n\}$, all of which are then followed by the same job J_e. Please refer to Figure 4.1 for an illustration.

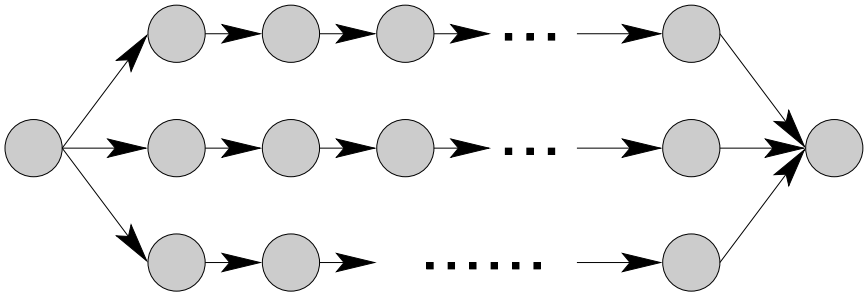


Figure 4.1: The example of chain-like structure job. The nodes represent the jobs, and the arrows represent the precedence constraints on jobs.

We can schedule the workflow in Figure 4.1 by three steps: First, we place the root J_s by a greedy algorithm. We try J_s on each processor and assign it to the processor with the earliest completion time. Second, we use the dynamic programming to find the optimal solution for the n job chains. We only need to consider time slots or portion of time slots that are *after* the completion time of job J_s, and apply the dynamic programming. Finally, we schedule job J_e using the greedy algorithm again. It is easy to see that by using the three steps we can achieve optimal makespan for this chain-like structure.

Chapter 5

Heuristics

We propose two classes of heuristic algorithms for our scheduling problem. First we propose the *chain ordering heuristic*, then the *longest job first heuristic*.

5.1 The Chain Ordering Heuristic

The chain ordering heuristic is a greedy heuristic that schedules one job chain at a time according to a particular order. This order of job chains is determined by the characteristic of job chains. In this paper we consider two characteristics of job chains – *average execution time* and *makespan*.

5.1.1 Average Execution Time

We use *average execution time* (denoted by a_i) of job chains to determine the scheduling order among job chains. The average execution time of a job chain is the average execution time of jobs within this job chain, i.e., $a_i = \frac{\sum_{j=1}^{h_i} t_{i,j}}{h_i}$. We sort the average execution time a_i in *descending* order, and schedule one job chain onto processors in each iteration using the earliest completion time first algorithm describing in Chapter 3.

We think scheduling job chains in descending average execution time might reduce the makespan for the following reasons. If we assign jobs with smaller average execution time first, we tend to delay the execution of some long jobs. Those long jobs can easily

become the bottleneck of their job chains.

If we schedule long jobs first we might have more options to place them, and it is more likely to find suitable and earlier time slots for them, hence reducing the makespan. The pseudo code of the algorithm is in Algorithm 1.

Algorithm 1: Chain Ordering Heuristic Algorithm

```
Input: Job Chain Set H
Output: A Schedule S
begin
    for i ← 1 to n do
        a_i ← average execution time of H_i
    end
    sort H_i in descending order of a_i
    for i ← 1 to n do
        R_{H_i} ← 0
        for j ← 1 to h_i do
            assign J_{i,j} to available period I computed by ECF(J_{i,j}, t_{i,j}, R_{H_i})
            R_{H_i} = max(R_{H_i}, starting time of I) + t_{i,j}
        end
    end
    return S
end
```

5.1.2 Expected Makespan

We can use *expected makespan* of job chains to decide the scheduling order among them. The expected makespan of a job chain is the makespan that we assume that the system only has the current job chain to schedule, and all processors are available. The optimal makespan can be calculated using the greedy method we described for one job chain case in Section 3.1. After we determine the job chain order we then schedule job chains by *increasing* makespan order.

The reason for scheduling job chains in increasing expected makespan order is as follow: A job chain H_b could have a large makespan because one or several jobs in H_b cannot find suitable time slots, and this causes large gaps among themselves and after their predecessors. If we schedule job chains with shorter makespan first we may delay

Procedure ECF(*Job J, Job processing time t, Time point T*)

Input: Job J, Job processing time t, Time point T
Output: An Idle Period
begin
  $l \leftarrow \infty$
  $I \leftarrow \varnothing$
  for $i \leftarrow 1$ **to** m **do**
    $j \leftarrow 1$
    while $e_{i,j} - t < \max(T, s_{i,j})$ **do**
      $j++$
    end
    if $\max(T, s_{i,j}) + t < l$ **then**
      $l \leftarrow \max(T, s_{i,j}) + t$
      $I \leftarrow I_{i,j}$
    end
  end
  return I
end

the execution of the beginning part of chain H_b. However, the delay may get rid of the gaps within this job chain, and will not increase the overall makespan. Based on this intuition we schedule job chains according to increasing expected makespan order.

The scheduling algorithm using expected makespan is similar to the heuristic using average execution time. We only need to sort jobs according to the expected makespan. The pseudo code is in Algorithm 1.

5.2 Longest Job First Heuristic

The two heuristics described earlier are easy to implement, but do not perform well in some special cases. For example, if a job chain H_1 has a long job $J_{1,j}$ and many short jobs, and another job chain H_2 has jobs that have nearly the same execution time, and the average execution time a_1 is smaller than a_2. The average execution time heuristic in Section 5.1.1 will schedule H_2 first. However, the long job $J_{1,j}$ will be the bottleneck of its chain, and this may cause average execution time heuristic not perform well under this situation.

To handle such special cases we propose *longest job first heuristic* that schedules one job at a time, instead of one job chain at a time. In every iteration we choose the job with the longest processing time from the next jobs of all job chains, and assign it to the processor with earliest completion time. By choosing one job at a time, we make decision based on each job, rather than on job chains. The intuition is that we might handle special cases well because we are not required to schedule the entire chain. The pseudo code is in Algorithm 3.

Algorithm 3: Longest Job First Heuristic Algorithm

Input: Job Chain Set H
Output: A Schedule S
begin
  $U \leftarrow (0, \ldots, 0)$
  $R_H \leftarrow (0, \ldots, 0)$
  while *one of* u_i *is not* h_i **do**
    take $J_{i,j}$ which has longest processing time in every J_{i,h_i+1}
    assign $J_{i,j}$ to available period I computed by $ECF(J_{i,j}, t_{i,j}, R_{H_i})$
    $u_i \leftarrow u_i + 1$
    $R_{H_i} = \max(R_{H_i}$, starting time of $I) + t_{i,j}$
  end
  return S
end

Chapter 6

Experiments

In this chapter we evaluate the performance of our heuristic algorithms by comparing the results against the optimal solutions found by the dynamic programming.

6.1 Environment Settings

The setting of the number of jobs chains and processors in our experiment is as follows. We consider two cases of job chains and processors. In the first case we have 2 processors and 3 job chains, and in the second case we have 3 processors and 2 job chains. In the first case job chains compete the available time slots more often than in the second case.

Other parameters in our experiments are described as follows. In the first case the number of jobs per job chain is from 1 to 5. In the second case, the number of jobs per job chain is from 3 to 8. The length of available time slots and the gaps between two time slots are randomly chosen from 1 to 50, therefore the processors are available about half of time. The execution time of job is randomly chosen from 1 to 40. We run the experiment for 100 times for each parameter set, and calculate the average results. We list the experimental parameters in Table 6.1.

number of processors m	2	3
number of job chains n	3	2
number of job per job chain h_i	1,. . .,5	3,. . .,8
time slot length $I_{i,j}$	1–50	
job execution time $t_{i,j}$	1–40	

Table 6.1: Experiment environment parameters

6.2 Performance Evaluation

In this section we compare the performance of different heuristics, using the optimal solution from the dynamic programming as a base. We use *relative makespan* as the measurement of performance. The relative makespan from a heuristic is the makespan divided by the optimal makespan from the dynamic programming. The relative makespans are shown in Figure 6.1 and Figure 6.2.

In Figure 6.1 we compare the relative makespan from three heuristics when we have 2 processors and 3 job chains. The heuristics are chain-ordering using average execution (*order-avg*), chain-ordering using expected makespan (*order-mk*), and longest job first (*LJF*) described in Section 5.1.1, Section 5.1.2, and Section 5.2. First we find that the relative makespans of all heuristics are no more than 1.25 for any number of jobs per chain. Second, we find that longest job first heuristic outperforms the other two heuristics. That is because the longest job first heuristic considers one job at a time job so that the scheduling is more flexible in choosing jobs.

In Figure 6.2 we compare the relative makespan from the three heuristics when we have 3 processors and 2 job chains. Job chains now do not need to compete for the available periods as often as in the previous case, therefore we observe that all three relative makespan are now reduced to be less than 1.2. We also find that the relative makespans of the three heuristics now become closer to each other. Nevertheless the longest job first heuristic still has the best performance.

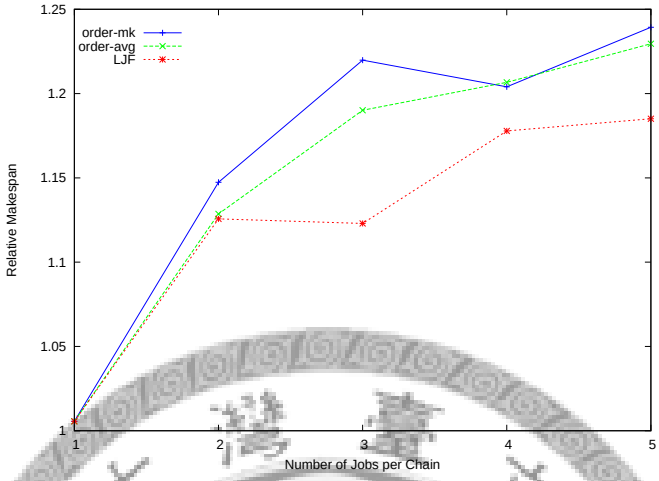


Figure 6.1: Relative makespan for 2 servers and 3 chains

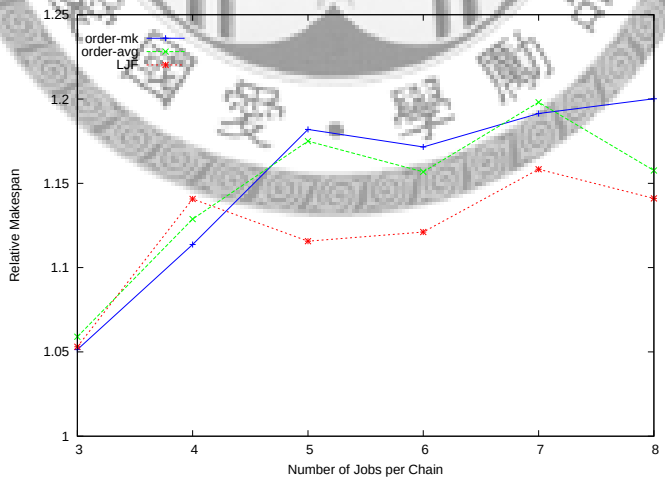


Figure 6.2: Relative makespan for 3 servers and 2 chains

Chapter 7

Conclusion

This paper introduces techniques in scheduling jobs with dependency constraint to processors whose available time are fragmented into time slots. We discuss two job dependency patterns – tree and chains. We show that it is NP-complete to schedule jobs with a tree dependency pattern. Then we propose a dynamic programming algorithm to get the optimal schedule for assuaging jobs with linear dependency, The dynamic programming can be generalized to heterogeneous environment and tree-like dependency structure. In order to reduce the time of scheduling we also propose three different heuristics. Experimental results indicate that these heuristics do provide near optimal schedules even when compared against the optimal solution found by the dynamic programming.

We are investigating a more precise estimate on the hardness of the scheduling jobs with linear dependency, since at this moment we do not have a NP-complete proof, or a polynomial time algorithm. We also would like to generalize our linear dependency dynamic programming to other workflow patterns, so that they can take advantage of this dynamic programming to find good schedules.

Bibliography

[1] Jianzhong Du, Joseph Y-T. Leung, and Gilbert H. Young. Scheduling chain-structured tasks to minimize makespan and mean flow time. *Inf. Comput.*, 92(2):219–236, 1991.

[2] J. D. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[3] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman and Co. New York, NY, USA., 1979.

[4] T. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.

[5] Rashid Al-Ali Kaizar, Rashid J. Al-ali, Kaizar Amin, Gregor Von Laszewski, Omer F, David W. Walker, Mihael Hategan, and Nestor Zaluzec. Analysis and provision of qos for distributed grid applications. *Journal of Grid Computing*, 2:163–182, 2004.

[6] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems. *Software Practice and Experience*, 32:135–164, 2002.

[7] Warren Smith, Ian Foster, and Valerie Taylor. Scheduling with advanced reservations. In *In Proceedings of IPDPS00*, pages 127–132, 2000.

[8] Anthony Sulistio and Rajkumar Buyya. A grid simulation infrastructure supporting advance reservation, 2004.

[9] Andrew Stephen Mcgough, Ali Afzal, John Darlington, Nathalie Furmento, Anthony Mayer, and Laurie Young. Making the grid predictable through reservations and performance modelling. *Comput. J.*, 48(3):358–368, 2005.

[10] M. Wieczorek, M. Siddiqui, A. Villazon, R. Prodan, and T. Fahringer. Applying advance reservation to increase predictability of workflow execution on the grid. pages 82–82, 2006.

[11] C. Castillo, G.N. Rouskas, and K. Harfoush. On the design of online scheduling algorithms for advance reservations and qos in grids. pages 1–10, 2007.

[12] C. Castillo, G.N. Rouskas, and K. Harfoush. Efficient resource management using advance reservations for heterogeneous grids. pages 1–12, 2008.

[13] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 117–126, New York, NY, USA, 2007. ACM.

[14] Holly Dail, Otto Sievert, Fran Berman, Henri Casanova, Asim Yarkhan, Sathish Vadhiyar, Jack Dongarra, Chuang Liu, Lingyun Yang, Dave Angulo, and Ian Foster. Scheduling in the grid application development software project, 2003.

[15] Jia Yu, Rajkumar Buyya, and Kotagiri Ramamohanarao. Workflow scheduling algorithms for grid computing. In *Studies in Computational Intelligence*. Springer, 2008.

[16] Peter Brucker. *Scheduling Algorithms*. Springer, 2007.