University of Alberta

Approximation Algorithms for Multi-processor Task Scheduling Problems on Identical Parallel Processors

by

Saber Khakpash

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2013
Edmonton, Alberta

Abstract

In this thesis we present approximation algorithms for some multi-processor task scheduling problems. In a scheduling problem, there is a set of processors P that can be used to process a set of tasks T and the goal is to find a feasible scheduling of the tasks on the processors, while optimizing an objective function. In a multi-processor task scheduling problem, tasks can be executed on several processors simultaneously.

In scheduling problems, tasks can be preemptive or non-preemptive. Preemptive tasks can be interrupted during their execution and resumed later with no cost. In contrast, a non-preemptive task cannot be interrupted during its execution. In Chapter 2 we propose polynomial time algorithms using linear programming to solve the preemptive scheduling problems for minimizing the maximum completion time, the maximum latency, and the maximum flow time. In Chapter 3 we consider two non-preemptive scheduling problems: the problem of minimizing the maximum completion time when tasks have minimum degree of parallelism, and problem of minimizing the maximum flow time.

In Chapter 4 we consider online scheduling of multi-processor tasks with minimum degree of parallelism. In online scheduling, the scheduler does not have access to the entire input instance initially and the scheduler will have access to tasks' characteristics over time. We show that it is not possible for an online scheduler to find a feasible scheduling for all input instances of the problem. We propose a bicriteria $(O(\log m), 1)$-approximation algorithm for the problem.

Table of Contents

List of Figures

Chapter 1

Introduction

Scheduling problems are some of the most well studied problems in computer science, with applications in computer resource allocation, networks, manufacturing, transport, and many other areas. In a scheduling problem, there is a set of processors $P = \{P_1, P_2, ..., P_m\}$ that can be used to process a set of tasks $T = \{T_1, T_2, ..., T_n\}$. The goal is to find a feasible scheduling of the tasks on the processors, while optimizing an objective function.

There are three different processor environments: identical processors, uniform processors, and unrelated processors. In an identical environment all processors have the same speed and the processing time of a task is processor independent. So when processors are identical, the processing time of task T_j can be denoted by p_j. In a uniform environment the speed of processors can be different. In this environment the processing time of task T_j on processor P_i is p_j/s_i, where s_i is the speed of processor P_i. In an unrelated environment the speed of a processor depends on the task it is executing. In unrelated environments we use p_{ij} to denote the processing time of task T_j when it is executed by processor P_i.

In scheduling problems, tasks can be preemptive or non-preemptive. Preemptive tasks can be interrupted during their execution and resumed later with no extra cost. In contrast, a non-preemptive task cannot be interrupted during its execution.

Scheduling problems can be divided into two groups: offline problems and online problems. In offline problems, the scheduler has access to all the characteristics of tasks at the beginning. But in online problems, the scheduler does not have access to the entire input instance initially and the scheduler will have access to the tasks' characteristics over time. Therefore, the scheduler does not know about the future tasks and should make its decisions online based on the currently available information.

1.1 Multi-processor task scheduling

With the advent of parallel computing systems, a new type of scheduling called *multi-processor task scheduling* emerged. In multi-processor task scheduling problems, the sched-

uler is allowed to schedule a task on several processors simultaneously. The number of processors assigned to a task T_j is called its degree of parallelism.

In a multi-processor task scheduling problem, processors can be dedicated or parallel. Dedicated processors are specialized for specific functionalities. So when processors are dedicated, a task may be processed by a specific set or a group of alternative sets of processors. On the contrary, parallel processors are similar and all processors are capable of processing all the tasks.

There are two types of multi-processor task scheduling problems for dedicated processors:

- Fixed set: For each task T_j a set Fix_j of processors is given. Fix_j denotes the subset of processors required by task T_j. The scheduler should schedule T_j on all of the processors in Fix_j simultaneously. The processing time of T_j is denoted by p_j.

- Alternative sets: For each task T_j a set set_j of subsets of processors is given and the scheduler should schedule T_j on all of the processors in one of the subsets in set_j simultaneously. The processing time of T_j is denoted by p_j.

There are three types of multi-processor task scheduling problems for parallel processors:

- Rigid tasks: When tasks are rigid, the number of processors required for executing a task is fixed and given. In these problems we use $size_j$ to denote the number of processors required by task T_j. Task T_j should be scheduled simultaneously on exactly $size_j$ processors during its execution. The processing time of T_j is denoted by p_j.

- Moldable tasks: When tasks are moldable, the scheduler can schedule a task on any number of processors, but the number of processors cannot change during the execution of the task. The execution time of a task depends on the number of processors it is scheduled on. We denote the execution time of T_j on l processors by $p_j(l)$.

- Malleable tasks: When tasks are malleable, the scheduler can schedule a task on any number of processors and the scheduler is also allowed to change the number of processors assigned to a task during the execution of the task. We denote the processing time of task T_j on l processors by $p_j(l)$. Which means we can process a fraction of size $\frac{1}{p_j(l)}$ of T_j by assigning l processors to T_j in one unit of time.

In moldable and malleable scheduling problems, the processing time of a task depends on the number of processors assigned to it and it is denoted by a function $p_j(l)$. One of the special cases of this function considered in the scheduling literature is called *linear speed-up* function. A processing time function $p_j(l)$ is linear if $p_j(l) = p_j(1)/l$.

1.2 Notations and Preliminaries

In this section we introduce some notations used in this thesis. In Section 1.2.1 we give a brief introduction to approximation algorithms. In Section 1.2.2 we introduce some notations and definitions about linear programming. Finally in Section 1.2.3 we introduce scheduling problem notation of Graham *et al.* [31], extended to include multi-processor task systems.

1.2.1 Approximation algorithms

An optimization problem is the problem of finding a feasible solution with optimum objective value from a set of feasible solutions. An optimization problem can be defined as a pair (S, f), where S is the set of feasible solutions and $f : S \to \mathbb{R}$ is the objective function. If our goal is to minimize the objective value, we say the problem is a minimization problem and if our goal is to maximize the objective value, we say the problem is maximization problem.

Many optimization problems are NP-hard and it is not possible to find a polynomial time algorithm for them unless P=NP. So we try to approximate the solution. We say an algorithm is an α-*approximation algorithm* for a minimization problem, if it finds a feasible solution s in polynomial time such that $f(s) \leq \alpha OPT$, where OPT is the optimum objective value for the problem. Similarly, an α-*approximation* for a maximization problem is an algorithm that finds a solution s in polynomial time such that $f(s) \geq \frac{OPT}{\alpha}$. Sometimes α is called the *approximation ratio*, *approximation factor*, or *performance guarantee*.

For NP-hard optimization problems, it is not possible to find a 1-approximation algorithm unless P=NP. For these problems the best we can hope for is a $(1+\epsilon)$-approximation algorithm for an arbitrary small but fixed $\epsilon > 0$. We say an algorithm is a *polynomial time approximation scheme* (PTAS) if for any given fixed constant $\epsilon > 0$, it finds a feasible solution s such that $f(s) \leq (1+\epsilon)OPT$ in time polynomial in input size (but maybe exponential in $\frac{1}{\epsilon}$). We say an algorithm is an *asymptotic polynomial time approximation scheme* (APTAS) if for any given fixed constant $\epsilon > 0$, it finds a feasible solution s such that $f(s) \leq (1+\epsilon)OPT + c$ in time polynomial in input size (but maybe exponential in $\frac{1}{\epsilon}$), where c is a constant. An *asymptotic fully polynomial time approximation scheme* (AFPTAS) is an APTAS whose time complexity is polynomial in the input size and $\frac{1}{\epsilon}$.

For some optimization problems, we may allow an approximation algorithm to violate some constraints within a factor, to achieve a better approximation ratio. This leads to a generalization of approximation algorithms. A *bicriteria* (α, β)-*approximation algorithm* is a polynomial time algorithm that finds a solution within a factor α of the optimum solution and violates a group of constraints within factor β.

1.2.2 Linear Programming

Linear programming (LP) is a problem that can be expressed in the following form [53].

$$\begin{aligned} \text{Minimize} \quad & \sum_{i=1}^{n} c_i x_i & \\ \text{Subject to} \quad & \sum_{i=1}^{n} A_{ji} x_i \geq b_j & \forall 1 \leq j \leq m, \\ & x_i \geq 0 & \forall 1 \leq i \leq n, \end{aligned}$$

where $x \in \mathbb{R}^n$ represents the vector of variables and $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$ are fixed known coefficients. The goal is to find a vector x which satisfies the inequalities and minimizes the objective function.

We say vector x is a feasible solution to the LP if x satisfies all of the constraints. We say vector x is a *Basic Feasible Solution* (BFS) if it is a feasible solution and it cannot be written as a convex combination of two other feasible solutions.

If an LP has a feasible solution, we say the LP is *feasible*. Otherwise, we say the LP is *infeasible*. If an LP is feasible, but the optimum value for the objective function is unbounded, we say the LP is *unbounded*. Otherwise, we say the LP has a bounded optimum. If an LP has a bounded optimum, then there exists a basic feasible solution that attains the optimum value.

We say an algorithm solves linear programming, if for a given linear program, the algorithm produces a basic feasible solution x minimizing the objective function, or determines the given LP is infeasible or unbounded. LP can be solved in time polynomial in n and m using *ellipsoid algorithm* [32]. In some cases, where there exists a *separation oracle* for an LP in time polynomial in n, it is possible to solve the LP in time polynomial in n even if the number of constraints is very large. A separation oracle is an algorithm that for a given vector x either determines that x is feasible or returns a violated constraint.

For every linear programming problem, referred to as a primal problem, there is a converted dual problem, which provides a lower bound to the optimum value of the primal problem. The dual LP has the following form.

$$\begin{aligned} \text{Maximize} \quad & \sum_{j=1}^{m} b_j y_j & \\ \text{Subject to} \quad & \sum_{j=1}^{m} A_{ji} y_j \leq c_i & \forall 1 \leq i \leq n, \\ & y_j \geq 0 & \forall 1 \leq j \leq m. \end{aligned}$$

Every feasible solution for a linear program gives a bound on the optimal value of the objective function of its dual. There are two fundamental theorems about duality: the weak duality theorem and the strong duality theorem.

The weak duality theorem states that the objective function value of the primal LP for any feasible solution is always greater than or equal to the objective function value of the dual LP for any feasible solution. In other words every feasible solution of the dual LP can be used as a bound for the optimal solution of the primal LP and vice versa.

The strong duality theorem states that if the primal has an optimal solution x, then the dual also has an optimal solution y, and $\sum_{i=1}^{n} c_i x_i = \sum_{j=1}^{m} b_j y_j$. Using this theorem, we can conclude that we can find the solution of the primal LP by solving the dual LP.

Integer Programming (IP) is presented the same way as linear programming with the additional constraint that x_i should be integer for every $1 \leq i \leq n$. A large number of combinatorial optimization problems can be formulated as integer programs. Among them are NP-complete problems that can be reduced to IP in polynomial time. So the decision version of integer programming is NP-complete and we cannot solve IP in polynomial time unless $P = NP$.

If we relax the constraints that x_i should be integer, then we have a linear program. We call this LP, the LP relaxation of the integer programming. Since every feasible solution for the integer programming instance is a feasible solution for the LP relaxation, then the optimum value of LP relaxation provides a lower bound for the optimum value of IP if it is a minimization problem or an upper bound for the optimum value of IP if it is a maximization problem.

Given an LP-relaxation for a minimization problem Π, let $OPT(I)$ denote the cost of an optimal solution to a given instance I of Π, and let $OPT_f(I)$ denote the cost of an optimal solution to the LP-relaxation. The *integrality gap* of the LP-relaxation is defined to be

$$\sup_I \frac{OPT(I)}{OPT_f(I)}.$$

In the case of a maximization problem, the integrality gap is defined to be the infimum of this ratio.

1.2.3 3-Field Notation

In a scheduling problem, there is a set of processors $P = \{P_1, P_2, ..., P_m\}$ that can be used to process a set of tasks $T = \{T_1, T_2, ..., T_n\}$. Each processor can process one task at a time, but a task may be allowed to be executed by several processors simultaneously. The goal is to find a feasible scheduling of tasks, while optimizing an objective function.

A scheduling problem can be specified with three elements: the processor set characteristics, the task system characteristics, and the optimality criterion. Graham *et al.* [31] introduced a three-field representation for scheduling problems. In this representation a scheduling problem is denoted by a three-field descriptor $\alpha|\beta|\gamma$, where α defines the processor set characteristics, β defines the task system characteristics, and γ specifies optimality criterion.

The first field α defines processors characteristics. $\alpha = 1$ means there is only one processor. $\alpha = P$ denotes identical processors. $\alpha = Q$ denotes uniform processors with different speeds and $\alpha = R$ denotes unrelated processors, which means the execution time of a task depends on the processor it is scheduled on.

In the case of parallel processors it is possible to specify the number of processors. For example $\alpha = P3$ means there are three identical processors and $\alpha = Rm$ means there is a constant number m of unrelated parallel processors, but m is not specified.

The second field β represents task system characteristics. This field can be a subset of the following items.

- r_j: Each task has a release time and it can be scheduled only after its release time.
- s_j: Task T_j has a start time s_j and it must be started at s_j.
- d_j: Task T_j has a deadline d_j and should be finished by its deadline.
- *prec* : Tasks have precedence constraints denoted by a precedence graph, which is an arbitrary directed acyclic graph.
- *pmtn*: Preemption is allowed. A task may be interrupted during its execution and resumed later at no cost.
- *online-list*: The scheduler must schedule tasks from some queue or list one by one. The characteristics of a task becomes known after scheduling the previous tasks on the list.
- *online-time*: Each task has a release time and the scheduler is unaware of the existence of a task and its characteristics before its release time.
- *online-time-nclv*: Each task has a release time and the scheduler is unaware of the existence of a task and its characteristics before its release time. The scheduler does not know tasks' processing times even after their release times.

For multi-processor task scheduling problems, the following items are used to specify the characteristics of multi-processor tasks.

- fix_j: Task T_j must be processed by a specific set of dedicated processors.
- set_j: Task T_j may be executed by some family of alternative dedicated processor sets.
- $size_j$: Tasks are rigid. Task T_j should be executed by exactly $size_j$ parallel processors simultaneously.
- *any*: Tasks are moldable. Tasks can be executed on any number of processors.

- var: Tasks are malleable. The number of processors executing a task can change during its execution.
- lin: In a malleable or moldable task system, the processing time functions of tasks are linear.
- δ_j: Tasks have maximum degree of parallelism. Task T_j cannot be scheduled on more that δ_j processors simultaneously.
- min_j: Tasks have minimum degree of parallelism. Task T_j should be scheduled on at least min_j processors during its execution.

The last field specifies the optimality criterion. There are two general groups of optimality criteria: min-max criteria and min-sum criteria. In min-max criteria the goal is to minimize $\max_j f(T_j)$, where $f(T_j)$ is an arbitrary function. In contrast, in min-sum criteria the goal is to minimize $\sum_j f(T_j)$, where $f(T_j)$ is an arbitrary function.

Given a schedule, we define C_j to be the completion time of task T_j. Then we can compute the flow time $F_j = C_j - r_j$, and the latency $L_j = max(C_j - d_j, 0)$ for each task T_j. These functions can be used in min-max and min-sum criteria. For example, some important min-max criteria are maximum completion time (also called makespan) $max_j\{C_j\}$ denoted by C_{max}, maximum latency $L_{max} = max_j\{max(C_j - d_j, 0)\}$, and maximum flow time $F_{max} = max_j\{C_j - r_j\}$. Important min-sum criteria are total completion time $\sum C_j$, total weighted completion time $\sum w_j C_j$, total latency $\sum L_j$, and total flow time $\sum w_j F_j$.

For example $P|size_j, r_j, prec|\sum w_j C_j$ is the problem of minimizing weighted completion time for rigid tasks with release times and arbitrary precedence graph on identical parallel processors, and $P|any, pmtn|L_{max}$ is the problem of minimizing the maximum latency for preemptive moldable tasks on identical parallel processors.

1.3 Outline of thesis

In this thesis we study multi-processor task scheduling problems, their complexity, and some algorithms proposed to solve them. In Chapter 2 we study preemptive multi-processor tasks. The following problems are considered.

- $P|var, r_j|C_{max}$: The problem of minimizing the maximum completion time for malleable tasks with release times when preemption is allowed.
- $P|var, r_j, d_j|L_{max}$: The problem of minimizing the maximum latency for malleable tasks with release times and deadlines when preemption is allowed.
- $P|var, r_j|F_{max}$: The problem of minimizing the maximum flow time for malleable tasks with release times when preemption is allowed.

Polynomial time algorithms are proposed to solve these problems using linear programming.

In Chapter 3 we study non-preemptive multi-processor tasks scheduling. We consider the following problems.

- $P|var, s_j, min_j|C_{max}$: We show that finding a feasible solution for $P|var, s_j, min_j|C_{max}$ is strongly NP-hard. We also show that the problem is NP-hard when the processing time function is linear. We propose an algorithm for $Pm|var, lin, s_j, min_j|C_{max}$ which needs $(1+\epsilon)$-speed augmentation and runs in time polynomial in n and $\frac{1}{\epsilon}$.

- $P|size_j, r_j|F_{max}$: We show that the problem is strongly NP-hard and propose a bicriteria $(1+\epsilon, O(1/\epsilon))$-approximation algorithm for the problem.

In Chapter 4 we consider online scheduling of multi-processor tasks with minimum degree of parallelism. We show that it is not possible for an online scheduler to find a feasible scheduling for all input instances of the problem. We propose a bicriteria $(O(\log m), 1)$-approximation algorithm for the problem.

Chapter 2

Preemptive Scheduling

In this chapter we study scheduling of preemtive multi-processor tasks on a set of identical processors. In the preemptive execution, processors are allowed to interrupt a task during its execution and resume the task later with no extra cost. We also assume that migration is allowed, which means a different set of processors can be assigned to a task after interruption.

In multi-processor task scheduling problems each task can be executed on several processors simultaneously. In the rigid multi-processor task model, the number of processors for each task is specified for each task. In the moldable and malleable models tasks can be executed on an arbitrary number of processors and the processing time of a task depends on the number of processors assigned to that task (sometimes called the degree of parallelism).

In this chapter we focus on preemptive scheduling of malleable multi-processor tasks and consider different optimality criteria. In Section 2.2 we consider the scheduling problem of minimizing the makespan when each task has a release time. In Section 2.3 the problem of minimizing maximum latency and in Section 2.4 the problem of minimizing maximum flow time is considered.

2.1 Related works

Preemptive non-malleable parallel scheduling problem $P|size_j, pmtn|C_{max}$ is shown to be NP-hard [22, 20]. However, the special case of $size_j \in \{1, k\}$ is polynomially solvable [11]. When the number of processors is constant, the problem can be solved in polynomial time using Linear Programming [11]. Janson and Porolab [38] improved this result and proposed a linear time algorithm for $Pm|size_j, pmtn|C_{max}$. They also proposed a pseudo-polynomial time algorithm for $P|size_j, pmtn|C_{max}$.

Regarding the complexity of preemptive malleable parallel tasks, Du and Leung [22] showed that $P|any, pmtn|C_{max}$ is strongly NP-hard and $Pm|any, pmtn|C_{max}$ is NP-hard. They also proposed a pseudo-polynomial time algorithm for $Pm|any, pmtn|C_{max}$, which implies that this problem cannot be strongly NP-hard.

For preemptive moldable parallel tasks, Janson and Porolab [38] showed that their Linear Program for non-malleable tasks can be modified to solve $Pm|var|C_{max}$ in linear time and $P|var|C_{max}$ in pseudo-polynomial time. Drozdowski [19] proposed an on-line polynomial time algorithm for $P|var, lin, \delta_j, r_j|C_{max}$.

In [12], the problem $Pm|size_j, pmtn|L_{max}$ is shown to be polynomially solvable when $size_j \in \{1, k\}$ using Linear Programming. Janson and Porolab [38] also generalized their Linear Program for minimizing maximum latency. They developed linear time algorithms for $Pm|size_j, pmtn|L_{max}$ and $Pm|var|L_{max}$ and pseudo-polynomial algorithms for the scheduling problems $P|size_j, pmtn|L_{max}$ and $P|var|L_{max}$. For moldable parallel tasks with linear speedup, Vizing [55, 54] proposed a polynomial time algorithm for $P|var, lin, \delta_j, r_j|L_{max}$ and a polynomial time algorithm for $R|var, lin, r_j|L_{max} = 0$.

2.2 Makespan

In this section we consider the preemptive scheduing problem denoted by $P|var, r_j|C_{max}$. In this problem a set $T = \{T_1, T_2, ..., T_n\}$ of tasks has to be processed by m identical processors. Let r_j denote the release time of task T_j, and let $p_j(l)$ be the processing time of task T_j when it is executed on l processors simultaneously. Each task can be interrupted at any time and restarted later without any cost. The goal is to find a schedule minimizing the maximum completion time C_{max}.

We demonstrate how this problem can be solved in polynomial time using Linear Programming. Our Linear Program is similar in spirit to the configuration-based Linear Program used in [38] to solve some scheduling problems like $P|size_j, pmtn|C_{max}$ and $P|var|C_{max}$. In our problem a configuration is an assignment of processors to tasks which can be denoted by a function $f : \{1, 2, ..., m\} \to \{0, 1, ..., n\}$ where 0 denotes a dummy task.

Theorem 2.2.1. *The scheduling problem $P|var, r_j|C_{max}$ with n tasks and m processors can be solved in time polynomial in n and m.*

We may assume that the tasks are ordered according to their release time, *i.e.,* $r_1 \leq ... \leq r_n$. We define I_i to be the i-th time interval, where $I_i = [r_i, r_{i+1}]$ for $i = 1, 2, ..., n-1$ and $I_n = [r_n, C_{max}]$. Note that during I_i, only tasks in $\{T_1, T_2, ..., T_i\}$ can be scheduled.

We also define a configuration to be an assignment $f : \{1, 2, ..., m\} \to \{0, 1, ..., n\}$ of processors to tasks, where 0 denotes a dummy task. Let F_i be the set of all configurations that assign processors to only tasks from the subset $\{T_1, T_2, ..., T_i\}$. Note that the configurations in F_i can be scheduled during the i-th interval I_i and a configuration $f \in F_i$ is also a member of F_j for every $j > i$. For every $f \in F_i$, let $x_{f,i}$ denote the length (in time) that configuration f is being executed during the i-th interval. Then we can formulate our problem as follows.

$$
\begin{aligned}
&\text{Minimize} && C_{max} && && (2.1)\\
&\text{Subject to} && \sum_{f\in F_n} x_{f,n} \le C_{max} - r_n, && &&\\
& && \sum_{f\in F_i} x_{f,i} \le r_{i+1} - r_i, && \forall i \in \{1,2,...,n-1\}, &\\
& && \sum_{i=1}^{n}\sum_{f\in F_i} \frac{1}{p_j(|f^{-1}(j)|)} x_{f,i} \ge 1, && \forall j \in \{1,2,...,n\}, &\\
& && x_{f,i} \ge 0, && \forall i \in \{1,2,...,n\}, \forall f \in F_i. &
\end{aligned}
$$

The first and the second constraints are to assure that the total execution time of configurations scheduled in each time interval does not exceed the duration of the interval. The third constraint guarantees that all tasks are fully processed. In this constraint, i iterates over the time intervals and f iterates over the configurations executable in each time interval. The last constraint guarantees that the duration of a configuration in a time interval is not a negative number.

LP (2.1) has $3n$ constraints and $\sum_{i=1}^{n}(i+1)^m$ variables. We use the dual of this LP to solve the problem as justified in [53]. The dual has the following form.

$$
\begin{aligned}
&\text{Maximize} && r_n y_n - \sum_{i=1}^{n-1}(r_{i+1}-r_i)y_i + \sum_{j=1}^{n} z_j && && (2.2)\\
&\text{Subject to} && y_n \le 1, && &&\\
& && \sum_{j=1}^{n} \frac{1}{p_j(|f^{-1}(j)|)} z_j - y_i \le 0, && \forall i \in \{1,2,...,n\}, f \in F_i, &\\
& && y_i \ge 0, z_j \ge 0, && \forall i \in \{1,2,...,n\}, \forall j \in \{1,2,...,n\}. &
\end{aligned}
$$

LP (2.2) has polynomially many variables, but the number of constraints is exponential. However, we can solve this LP in polynomial time if there is a polynomial time separation oracle. The separation problem is the following: Given a $2n$-vector $X = (y_1, ..., y_n, z_1, ..., z_n)$, decide whether X is a feasible solution, and if it is not, find a constraint that vector X does not satisfy.

Lemma 2.2.2. *The separation problem for LP (2.2) can be solved in time polynomial in n and m, where n is the number of tasks and m is the number of processors.*

Proof. Given a $2n$-vector $X = (y_1, ..., y_n, z_1, ..., z_n)$, we are to decide if the following constraints are satisfied.

$$
\sum_{j=1}^{n} \frac{1}{p_j(|f^{-1}(j)|)} z_j \le y_i, \qquad \forall i \in \{1,2,...,n\}, f \in F_i
$$

We can check if all constraints are satisfied, by checking these constraints:

$$\max_{f \in F_i} \sum_{j=1}^{n} \frac{1}{p_j(|f^{-1}(j)|)} z_j \leq y_i, \qquad \forall i \in \{1, 2, ..., n\}$$

Let $F(p, k)$ denote the maximum of $\sum_{j=1}^{n} \frac{1}{p_j(|f_{pk}^{-1}(j)|)} z_j$ over the mappings $f_{pk} : \{1, ..., p\} \to \{0, 1, ..., k\}$. Now the constraints can be written as follows.

$$F(m, i) \leq y_i, \qquad \forall i \in \{1, 2, ..., n\}$$

Now it suffices to compute $F(p, k)$. Let $f_{pk} : \{1, ..., p\} \to \{0, 1, ..., k\}$ be the mapping that maximizes $\sum_{j=1}^{n} \frac{1}{p_j(|f_{pk}^{-1}(j)|)} z_j$. Assume that f_{pk} maps h processors to task T_k and $p-h$ processors to the other tasks. We know that $0 \leq h \leq p$. So we can compute $F(p, k)$ using the following recursive relation.

$$F(p, k) = \max_{0 \leq h \leq p} \{F(p - h, k - 1) + \frac{z_k}{p_k(h)}\}, \qquad \forall p \in \{0, ..., m\}, k \in \{1, ..., n\}$$

The base case is when $k = 0$.

$$F(p, 0) = 0, \qquad \forall p \in \{0, ..., m\}$$

Using this recursive formulation, we can use dynamic programming to compute $F(p, k)$. Algorithm 1 shows how this dynamic programming can be implemented.

Algorithm 1 Separation Oracle

```
1: for all p ← 0 to m do                          ▷ Initializing Base Case
2:     F(p, 0) ← 0
3: for k ← 1 to n do                              ▷ Dynamic Programming
4:     for p ← 0 to m do
5:         F(p, k) ← 0
6:         for h ← 0 to p do
7:             if F(p, k) < F(p − h, k − 1) + z_k / p_k(h) then
8:                 F(p, k) = F(p − h, k − 1) + z_k / p_k(h)
```

In this dynamic program, we can also store the configuration that maximizes $F(p, k)$ for each $0 \leq p \leq m$ and $0 \leq k \leq n$. The configuration that maximizes $F(m, i)$ identifies the constraint that is violated. The running time of the algorithm is $O(nm^2)$. Since it is possible to compute $F(p, k)$ in time polynomial in n and m, the separation problem can be solved in time polynomial in n and m. □

2.3 Latency

In the previous section, we considered the problem of minimizing the makespan of parallel preemptive tasks. In this section we consider minimizing the maximum latency of such tasks. We are given a set $T = \{T_1, T_2, ..., T_n\}$ of tasks. For each task T_j, let r_j be the

release time, d_j be the deadline, and let $p_j(l)$ denote the processing time of task T_j when it is executed on l processors simultaneously. The goal is to find a scheduling of these tasks on m identical processors minimizing the maximum latency, *i.e.,* $max_j\{max(0, C_j - d_j)\}$.

First we prove that given a maximum latency L, there is a polynomial time algorithm that decides if there exists a feasible scheduling with maximum latency at most L. Then we do a binary search over the possible maximum latency to find the scheduling which minimizes the maximum latency.

Lemma 2.3.1. *For the scheduling problem $P|var, r_j, d_j|L_{max}$ with n tasks and m processors, it is possible to decide in time polynomial in n and m, if there exists a scheduling with maximum latency at most L.*

We demonstrate how this can be done using Linear Programming. If there is a scheduling with maximum latency less than or equal to L, it means that we can schedule all the tasks such that each task T_j is scheduled in time interval $[r_j, d_j + L]$. Let $S = \{s_1, s_2, ..., s_{2n}\}$ be the set of all points that are endpoints of these time intervals. we may assume that these points are ordered, *i.e.,* $s_1 \leq ... \leq s_{2n}$. We define I_i to be the i-th time interval, where $I_i = [s_i, s_{i+1}]$ for $i = 1, 2, ..., 2n-1$. Note that during I_i, task T_j can be scheduled if $r_j \leq s_i$ and $d_j + L \geq s_{i+1}$. Let $T(i) = \{T_j | r_j \leq s_i, d_j + L \geq s_{i+1}\}$.

We define a configuration to be an assignment $f : \{1, 2, ..., m\} \to \{0, 1, ..., n\}$ of processors to tasks, where 0 denotes a dummy task. Let F_i be the set of all configurations that consists of only tasks from $T(i)$. Note that the configurations in F_i can be scheduled during the i-th interval I_i and a configuration $f \in F_i$ is also a member of F_j for every $j > i$. For every $f \in F_i$, let $x_{f,i}$ denote the length (in time) that configuration f is being executed during the i-th interval. Then we can formulate our problem as follows.

$$
\begin{aligned}
&\text{Minimize} && 0 && && (2.3)\\
&\text{Subject to} && \sum_{f \in F_i} x_{f,i} \leq s_{i+1} - s_i, && \forall i \in \{1, 2, ..., 2n-1\},\\
& && \sum_{i=1}^{2n-1} \sum_{f \in F_i} \frac{1}{p_j(|f^{-1}(j)|)} x_{f,i} \geq 1, && \forall j \in \{1, 2, ..., n\},\\
& && x_{f,i} \geq 0, && \forall i \in \{1, 2, ..., 2n-1\}, \forall f \in F_i.
\end{aligned}
$$

The first constraints guarantee that the total execution time of configurations scheduled in each time interval does not exceed the duration of that interval and the second ones guarantee that all tasks are fully processed.

LP (2.3) has exponentially many variables. So we use the dual of this LP, to solve it [53]. The dual LP has the following form.

$$\begin{aligned}
\text{Maximize} \quad & \sum_{i=1}^{2n-1}(s_i - s_{i+1})y_i + \sum_{j=1}^{n} z_j && (2.4)\\
\text{Subject to} \quad & \sum_{j=1}^{n} \frac{1}{p_j(|f^{-1}(j)|)} z_j - y_i \leq 0, && \forall i \in \{1,2,...,2n-1\}, f \in F_i,\\
& y_i \geq 0, z_j \geq 0, && \forall i \in \{1,2,...,2n-1\}, \forall j \in \{1,2,...,n\}.
\end{aligned}$$

Although the number of constraints in LP (2.4) is exponential, we can solve it in polynomial time if there is a polynomial time separation oracle. In the separation problem one has to decide whether a vector $X = (y_1, ..., y_{2n-1}, z_1, ..., z_n)$ is a feasible vector, and if it is not, find a constraint that vector X does not satisfy.

Lemma 2.3.2. *The separation problem for LP (2.4) can be solved in time polynomial in n and m, where n is the number of tasks and m is the number of processors.*

Proof. Given a vector $X = (y_1, ..., y_{2n-1}, z_1, ..., z_n)$, we are to decide if following constraints are satisfied.

$$\sum_{j=1}^{n} \frac{1}{p_j(|f^{-1}(j)|)} z_j \leq y_i, \qquad \forall i \in \{1,2,...,2n-1\}, f \in F_i$$

We can check if all the constraints are satisfied, by checking these constraints:

$$\max_{f \in F_i} \sum_{j=1}^{n} \frac{1}{p_j(|f^{-1}(j)|)} z_j \leq y_i, \qquad \forall i \in \{1,2,...,2n-1\}$$

Let $F(l,p,k)$ denote the maximum of $\sum_{j=1}^{n} \frac{1}{p_j(|f_{pk}^{-1}(j)|)} z_j$ over all configurations $f_{pk} : \{1,...,p\} \to \{0,1,...,k\}$ that do not map any processor to a task which is not a member of $T(l)$. Now the constraints can be written as follows.

$$F(i,m,n) \leq y_i, \qquad \forall i \in \{1,2,...,2n-1\}$$

Note that the number of constraints is $2n-1$ and now it suffices to compute $F(l,p,k)$. Let $f_{pk} : \{1,...,p\} \to \{0,1,...,k\}$ be the mapping that maximizes $\sum_{j=1}^{n} \frac{1}{p_j(|f_{pk}^{-1}(j)|)} z_j$ and does not map any processor to a task which is not a member of $T(l)$. Assume that f_{pk} maps h processors to task T_k and $p-h$ processors to the other tasks. If $T_k \notin T(l)$, f_{pk} should not map any processor to task T_k and $h = 0$. If $T_k \in T(l)$, f_{pk} can map any number of available processors to task T_k, so $0 \leq h \leq p$. So we can compute $F(l,p,k)$ using the following recursive relations.

$$F(l,p,k) = \begin{cases} \max_{0 \leq h \leq p}\{F(l,p-h,k-1) + \frac{z_k}{p_k(h)}\}, & T_k \in T(l),\\ F(l,p,k-1), & T_k \notin T(l), \end{cases}$$

for each $1 \leq l < 2n$, $0 \leq p \leq m$, and $1 \leq k \leq n$. The base case is when $k = 0$.

$$F(l,p,0) = 0, \qquad 1 \leq l < 2n, \quad 0 \leq p \leq m$$

Algorithm 2 Separation Oracle

for all $l \leftarrow 1$ to $2n-1$ **do** ▷ Initializing Base Case
  for all $p \leftarrow 0$ to m **do**
    $F(l,p,0) \leftarrow 0$
for all $l \leftarrow 1$ to $2n-1$ **do** ▷ Dynamic Programming
  for $k \leftarrow 1$ to n **do**
    for $p \leftarrow 0$ to m **do**
      if $T_k \notin T(l)$ **then**
        $F(l,p,k) = F(l,p,k-1)$
      else
        $F(l,p,k) \leftarrow 0$
        for $h \leftarrow 0$ to p **do**
          if $F(l,p,k) < F(l,p-h,k-1) + \frac{z_k}{p_k(h)}$ **then**
            $F(l,p,k) = F(l,p-h,k-1) + \frac{z_k}{p_k(h)}$

Using this recursive formulation, we can use dynamic programming to compute $F(l,p,k)$. Algorithm 2 shows how this dynaminac programming can be implemented.

The running time of the algorithm is $O(n^2m^2)$. Since it is possible to compute $F(l,p,k)$ in time polynomial in n and m, the separation problem can be solved in time polynomial in n and m. □

The separation problem and LP (2.4) can be solved in polynomial time. So it is possible to decide in polynomial time if there exists a scheduling with maximum latency at most L.

Theorem 2.3.3. *The scheduling problem $P|var, r_j, d_j|L_{max}$ with n tasks and m processors can be solved in polynomial time.*

Proof. Let P_{max} be the maximum processing time, *i.e.,* $max_{j,l}\{p_j(l)\}$. The maximum latency L_{max} is always less than nP_{max}. We presented an algorithm which decides if there exists a feasible scheduling with maximum latency at most L. So we can use a binary search algorithm to find the minimum maximum latency for which there exists a feasible scheduling. This can be done in $O(\log(nP_{max})poly(n,m))$, where $poly(n,m)$ is the time required for the algorithm which decides if a feasible scheduling can be found for a specific maximum latency. □

2.4 Flow time

The problem of minimizing maximum flow time $P|var, r_j|F_{max}$, can be considered as a special case of the scheduling problem denoted by $P|var, r_j, d_j|L_{max}$. In the previous section we demonstrated how we can solve this problem in polynomial time. The following is a consequence of Theorem 2.3.3.

Corollary 2.4.1. *The scheduling problem $P|var, r_j|F_{max}$ with n tasks and m processors can be solved in polynomial time.*

Proof. For each task T_j, we define d_j to be equal to r_j. So flow time for each task equals its latency. Theorem 2.3.3 says we can find a scheduling minimizing the maximum latency in polynomial time. So we can find a scheduling minimizing the maximum flow time in polynomial time. □

Chapter 3

Non-preemptive scheduling

In this chapter we consider some non-preemptive multiprocessor task scheduling problems. A multiprocessor task is a task that can be executed on several processors simultaneously and a non-preemptive task is a task that is not allowed to be interrupted during its execution.

The problems considered in this chapter are related to the starvation problem in scheduling tasks. Generally speaking, starvation is a problem in scheduling where a task execution is delayed for a long time. For example a *First In First Out* (FIFO) scheduler can have starvation problem in some environments, because a large task can "starve" a small task that arrives a bit later.

One solution to avoid the starvation problem in scheduling is to use objective functions like the maximum flow time. If we use the maximum flow time for objective function of the scheduling problem, the scheduler will try to minimize the maximum delay for execution of tasks and starvation does not occur.

Another solution is to ensure that each task is scheduled on at least some minimum number of processors at each time. We call this number the *minimum degree of parallelism* of the task. Since in this solution it is not necessary to change the objective function, then it can be used with any objective function. Using this method, it is also possible to enforce a priority order among tasks, by giving larger minimum degree of parallelism to the tasks with higher priority.

This solution was used by one of MapReduce implementations called *Hadoop Fair Scheduler* [58, 57, 56]. MapReduce is a framework typically used for processing parallelizable tasks across huge datasets using distributed computing on clusters of computers [17]. Early MapReduce implementations employed First In First Out (FIFO) which has problems with task starvation in most environments. *Hadoop Fair Scheduler* overcomes this issue, by ensuring that each task is allocated at least some minimum number of processors.

In Section 3.2 we consider the problem of minimizing makespan for malleable tasks with minimum degree of parallelism and in Section 3.3 we study the problem of minimizing the maximum flow time for rigid tasks.

3.1 Related works

For scheduling non-preemptive rigid tasks, the problems of minimizing the maximum latency and minimizing the maximum completion time are strongly NP-hard [22, 45]. Du and Leung [22] showed that $P5|size_j|C_{max}$ and Lee and Cai [45] showed that $P2|size_j|L_{max}$ are strongly NP-hard. For constant number of processors, Amoura *et al.* [2] proposed a PTAS for $Pm|size_j|C_{max}$. Johannes [40] presented a 2-approximation algorithm for $P|size_j, r_j|C_{max}$.

The problem of scheduling non-preemptive moldable tasks has been studied in several papers, *e.g.,* [22, 36, 37, 6]. Jansen [36] proposed an AFPTAS for $P|any|C_{max}$ and Jansen and Porkolab [37] proposed a PTAS for the special case of constant number of processors, *i.e.,* $Pm|any|C_{max}$.

For scheduling malleable tasks, Drozdowski and Kubiak [21] considered $P|var, lin, \delta_j|C_{max}$ in which tasks have maximum degree of parallelism and showed that this problem is polynomially solvable. Vizing [55] considered a similar scheduling problem with different objective function and showed that $P|var, lin, \delta_j|L_{max}$ is also polynomially solvable.

Regarding minimizing the maximum flow time, Bansal and Pruhs [7] considered the single processor scheduling problem and proposed a bicriteria $(1+\epsilon, O(1/\epsilon))$-approximation algorithm with speed augmentation. They showed that a scheduler that schedules the shortest task first finds a solution within a factor of $O(1/\epsilon)$ of the optimal solution using $(1+\epsilon)$-speed augmentation. Chekuri and Moseley [15] studied this problem for multiprocessor environments and proposed a bicriteria $(1+\epsilon, O(1/\epsilon))$-approximation algorithm for parallel processors. This problem has also been studied for unrelated processors and a bicriteria $(1+\epsilon, O(1/\epsilon))$-approximation algorithm is proposed for it [35, 3].

3.2 Minimum degree of parallelism

In this section we consider the problem of minimizing makespan for malleable tasks with minimum degree of parallelism, denoted by $P|var, s_j, min_j|C_{max}$. In this problem task T_j must be started at time s_j and it must be scheduled on at least min_j processors simultaneously. The number of processors assgined to task T_j can change during the execution of T_j but it always must be at least min_j. We denote the processing time function of task T_j on l processors by $p_j(l)$.

In Section 3.2.1, we study the complexity of problem $P|var, s_j, min_j|C_{max}$ and show it is strongly NP-hard. In Section 3.2.2 we consider the problem when the processing time functions are linear, *i.e.,* $p_j(l) = p_j(1)/l$, and we show that $P|var, lin, s_j, min_j|C_{max}$ is NP-hard. In Section 3.2.3 we propose a dynamic program for $Pm|var, lin, s_j, min_j|C_{max}$ which needs $(1+\epsilon)$-speed augmentation and runs in time polynomial in input size and $\frac{1}{\epsilon}$.

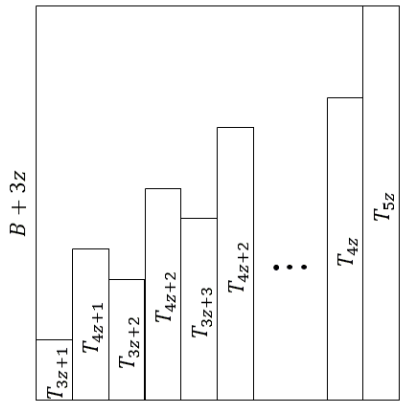

Figure 3.1: Scheduling of $R \cup S$

3.2.1 Complexity of $P|var, s_j, min_j|C_{max}$

In this section we show that $P|var, s_j, min_j|C_{max}$ is strongly NP-hard by reducing the 3-partition problem to it. The 3-partition problem is known to be strongly NP-hard [39].

3-partition problem: Given a list $A = (a_1, a_2, ..., a_{3z})$ of $3z$ integers such that $\sum_{i=1}^{3z} a_i = zB$ and $B/4 < a_i < B/2$ for each $1 \leq i \leq 3z$, can $J = \{1, 2, ..., 3z\}$ be partitioned into $J_1, J_2, ..., J_z$ such that $\sum_{j \in J_i} a_j = B$ and $|J_i| = 3$ for each $1 \leq i \leq z$?

Theorem 3.2.1. *Finding a feasible solution for the scheduling problem $P|var, s_j, min_j|C_{max}$ is strongly NP hard.*

Proof. We reduce the 3-partition problem to this problem. Given an instance I of the 3-partition problem, $A = \{a_1, a_2, ..., a_{3z}\}$, we construct an instance I' of $P|var, s_j, min_j|C_{max}$ as follows. Let $m = B + 3z$ and $T = Q \cup R \cup S$, where $Q = \{T_1, T_2, ..., T_{3z}\}$, $R = \{T_{3z+1}, T_{3z+2}, ..., T_{4z}\}$, and $S = \{T_{4z+1}, T_{4z+2}, ..., T_{5z}\}$.

$$s_j = \begin{cases} 0 & 1 \leq j \leq 3z \\ 2(j-3z) - 2 & 3z < j \leq 4z \\ 2(j-4z) - 1 & 4z < j \leq 5z \end{cases}$$

$$min_j = \begin{cases} 1 & 1 \leq j \leq 3z \\ 3(j-3z) & 3z < j \leq 4z \\ B + 3(j-4z) & 4z < j \leq 5z \end{cases}$$

$$p_j(l) = \begin{cases} 1 & 1 \leq j \leq 3z, l = a_j \\ \infty & 1 \leq j \leq 3z, l \neq a_j \\ 1 & 3z < j \leq 5z, 1 \leq l \leq m \end{cases}$$

The number of tasks is $5z$ and all of the computation for tasks' characteristics can be done in polynomial time. So this construction can be done in polynomial time. Now we prove that if instance I has a solution, then the constructed instance I' has a feasible scheduling, and vice versa.

First we show if instance I has a solution, there exists a feasible scheduling for instance I'. If I has a solution, we can partition $J = \{1, 2, ..., 3z\}$ into z disjoint sets $J_1, J_2, ..., J_z$, such that $\sum_{j \in J_i} a_j = B$ and $|J_i| = 3$ for $1 \leq i \leq z$. We construct a feasible scheduling for I' as follows. We assign min_j processors to $T_j \in R \cup S$ during $[s_j, s_j + 1]$. Then for each $j \in J_i$, we assign one processor to T_j during $[0, 2i-2]$ and a_j processors during $[2i-2, 2i-1]$. To show that this scheduling is a feasible scheduling, we should show that the number of required processors does not exceed the number of processors at any time.

Note that for each $j \in J_i$ task T_j will be finished at $t = 2i-1$, because we schedule T_j on a_j processors at $t = 2i-2$ and processing time of T_j on a_j processors is 1, *i.e.,* $p_j(a_j) = 1$ for $T_j \in Q$. So exactly 3 tasks finish at $t = 2i-1$ for every $1 \leq i \leq z$ and there are $3z - 3i$ tasks in Q to schedule during $[2i-1, 2i]$ and $3z-3i+3$ tasks during $[2i-2, 2i-1]$ for every $1 \leq i \leq z$.

During $[2i-1, 2i]$ we assign one processor to every task in Q. So we need $3z-3i$ processors for tasks in Q. We also need $B + 3i$ processors for T_{4z+i}. So the total number of required processors is $B + 3z$ which is equal to the number of processors. During $[2i-2, 2i-1]$ we have $3z - 3i + 3$ tasks in Q to schedule. We assign a_j processors to 3 of them and one processors to the other $3z - 3i$. We also assign min_{3z+i} processors to T_{3z+i}. So the total number of required processors are:

$$\begin{aligned}\sum_{j \in J_i} a_j + 3z - 3i + min_{3z+i} &= \sum_{j \in J_i} a_j + 3z - 3i + 3i \\ &= B + 3z - 3i + 3i \\ &= B + 3z.\end{aligned}$$

Now it suffices to show that if there exists a feasible scheduling for instance I', then instance I has a solution. Suppose there is a feasible scheduling for I'. Since for each $T_j \in R \cup S$, $p_j(l) = 1$ for all $1 \leq l \leq m$, there is no benefit in assigning more than min_j processors to these tasks and we can assume that the scheduling assigns min_j processor to these tasks. See Figure 3.1 for an illustration.

With similar reasoning, we can assume that the scheduling assigns either 1 or a_j processors to $T_j \in Q$ at any time. Moreover, the scheduling assigns a_j processors to $T_j \in Q$ for exactly one unit of time, because $p_j(l) = \infty$ for every $l \neq a_j$ and $p_j(a_j) = 1$.

Note that for $1 \leq i \leq z$, at time $t = 2i - 1$ the number of finished tasks in Q is exactly $3i$ for the following reason. It cannot be less than $3i$, because during $[2i-1, 2i]$ we assigned $B+3i$ processors to the tasks in $R \cup S$ and we have $3z-3i$ free processors available for tasks in Q. Each task in Q needs at least one processor, so at least $3i$ tasks should be finished at time $t = 2i - 1$.

It also cannot be more than $3i$, because we cannot finish more than 3 tasks in every 2 unit of time (because sum of every four members of A is more than B). So we can finish at

most $3i$ tasks by time $t = 2i - 1$.

So the solution assign a_j processors to T_j for exactly 3 tasks during $[2i-2, 2i-1]$ for each $1 \leq i \leq z$. For each task T_j scheduled on a_j processors during $[2i-2, 2i-1]$, we put j in J_i. This partitioning is a solution for instance I. □

Corollary 3.2.2. *$P|var, s_j, min_j|C_{max}$ is not approximable in polynomial time, unless P=NP.*

Proof. Any approximation algorithm for $P|var, s_j, min_j|C_{max}$ with approximation ratio α would find a feasible scheduling with makespan of at most α times the makespan of optimal scheduling. Since finding a feasible scheduling for this problem is strongly NP-hard, no polynomial time approximation algorithm exists for $P|var, s_j, min_j|C_{max}$ unless P=NP. □

3.2.2 Complexity of $P|var, lin, s_j, min_j|C_{max}$

In Section 3.2.1 we showed that $P|var, s_j, min_j|C_{max}$ is strongly NP-hard. In this section we consider the complexity of $P|var, lin, s_j, min_j|C_{max}$ which is a restricted version of $P|var, s_j, min_j|C_{max}$ in which the speed-up function is linear. We show that this problem is NP-hard by reducing the knapsack problem to it. The knapsack problem is well-known to be NP-hard [43, 51].

Knapsack Problem: Given a set U of n items, each with a weight w_u and a value v_u, a knapsack capacity limit W, and a desired total value V, is there a subset $U' \subset U$ such that $\sum_{u \in U'} w_u \leq W$ and $\sum_{u \in U'} v_u \geq V$?

Theorem 3.2.3. *Finding a feasible solution for $P|var, lin, s_j, min_j|C_{max}$ is NP-hard.*

Proof. We reduce the knapsack problem to this problem. Given an instance I of the knapsack problem, we construct an instance I' of the scheduling problem as follows. Let $m = W + \sum_{j=1}^{n} v_j$ and $T = \{T_1, T_2, ..., T_{n+1}\}$, where:

$$s_j = \begin{cases} 0 & 1 \leq j \leq n \\ 1 & j = n+1 \end{cases}$$

$$min_j = \begin{cases} v_j & 1 \leq j \leq n \\ W + V & j = n+1 \end{cases}$$

$$p_j = \begin{cases} v_j + w_j & 1 \leq j \leq n \\ W + V & j = n+1 \end{cases}$$

Note that the number of tasks is $n+1$ and this construction can be done in polynomial time. Now we show that if instance I has a feasible solution, then the constructed instance I' has a feasible scheduling, and vice versa.

First we assume that instance I' has a feasible scheduling and show that instance I has a solution. Assume that there exists a feasible scheduling for I' and let T' be the set of tasks completed before $t = 1$ in that feasible scheduling.

At $t = 1$, the number of processors required by all tasks is $\sum_{T_j \in T \setminus T'} min_j$ and the number of available processors is $W + \sum_{j=1}^{n} v_j$. In the feasible scheduling, the number of required processors should be not greater than the number of available processors. So:

$$\begin{aligned} W + \sum_{j=1}^{n} v_j &\geq \sum_{T_j \in T \setminus T'} min_j \\ &\geq \sum_{T_j \in T} min_j - \sum_{T_j \in T'} min_j \\ &\geq \sum_{j=1}^{n} v_j + V + W - \sum_{j:T_j \in T'} v_j \end{aligned}$$

Subtracting $W + \sum_{j=1}^{n} v_j - \sum_{j:T_j \in T'} v_j$ from both sides of the inequality yields:

$$\sum_{j:T_j \in T'} v_j \geq V \tag{3.1}$$

Now we compute the total number of processors assigned to the tasks during time interval $[0, 1]$. We know that all tasks in T' have been fully processed by time $t = 1$. So each task $T_j \in T'$ is scheduled on $v_j + w_j$ processors during time interval $[0, 1]$. Other tasks $T_j \in (T \setminus T') \setminus \{T_{n+1}\}$ should be scheduled on at least min_j processors during time interval $[0, 1]$. The total number of processors assigned to tasks should be not greater than the number of available processors. So:

$$\begin{aligned} W + \sum_{j=1}^{n} v_j &\geq \sum_{T_j \in T'} (v_j + w_j) + \sum_{T_j \in (T \setminus T') \setminus \{T_{n+1}\}} min_j \\ &\geq \sum_{T_j \in T'} v_j + \sum_{T_j \in T'} w_j + \sum_{T_j \in (T \setminus T') \setminus \{T_{n+1}\}} v_j \\ &\geq \sum_{T_j \in T'} w_j + \sum_{j=1}^{n} v_j \end{aligned}$$

Subtracting $\sum_{j=1}^{n} v_j$ from both sides of the inequality yields:

$$W \geq \sum_{T_j \in T'} w_j \tag{3.2}$$

Let U' be the set of items that their corresponding tasks are in T', formally $U' = \{u_j | T_j \in T'\}$. Equation (3.2) guarantees that the total weight of the items in U' is at most W and Equation (3.1) guarantees that their total value is at least V. So U' is a feasible solution for instance I.

Now it suffices to show that if there is a solution for instance I, there is a feasible scheduling for instance I'. Let U' be the feasible set of items for instance I.

$$\sum_{u_j \in U'} w_j \leq W \tag{3.3}$$

$$\sum_{u_j \in U'} v_j \geq V \tag{3.4}$$

Let T' denote the set of tasks corresponding to the items in U'. We schedule every task in $T_j \in T'$ on $v_j + w_j$ processors and other tasks $T_j \in T \backslash T'$ on min_j processors. In this scheduling, we need $\sum_{T_j \in T \backslash T' \backslash \{T_{n+1}\}} v_j + \sum_{T_j \in T'} v_j + w_j$ processors at $t = 0$ and $\sum_{T_j \notin T'} min_j$ at $t = 1$. It is sufficient to show that these two numbers do not exceed the number of available processors, which is $W + \sum_{j=1}^{n} v_j$.

$$\begin{aligned}
\sum_{T_j \in T \backslash T' \backslash \{T_{n+1}\}} v_j + \sum_{T_j \in T'} v_j + w_j &= \sum_{T_j \in T'} w_j + \sum_{j=1}^{n} v_j \\
&= \sum_{u_j \in U'} w_j + \sum_{j=1}^{n} v_j \\
&\leq W + \sum_{j=1}^{n} v_j && \text{by Equation (3.3)} \\
\sum_{T_j \notin T'} min_j &= V + W + \sum_{j=1}^{n} v_j - \sum_{T_j \in T'} v_j \\
&= V + W + \sum_{j=1}^{n} v_j - \sum_{u_j \in U'} v_j \\
&\leq W + \sum_{j=1}^{n} v_j && \text{by Equation (3.4)}
\end{aligned}$$

□

Corollary 3.2.4. *$P|var, lin, s_j, min_j|C_{max}$ is not approximable in polynomial time, unless P=NP.*

Proof. Any approximation algorithm for $P|var, lin, s_j, min_j|C_{max}$ with approximation ratio $\alpha \geq 1$ should find a feasible scheduling with makespan of at most α times the makespan of optimal scheduling. Since finding a feasible scheduling for this problem is NP-hard, no polynomial time approximation algorithm exists for $P|var, lin, s_j, min_j|C_{max}$ unless P=NP. □

3.2.3 Dynamic programming algorithm for $Pm|var, lin, s_j, min_j|C_{max}$

In this section we propose an algorithm for $Pm|var, lin, s_j, min_j|C_{max}$ which finds a scheduling with minimum makespan that runs in time polynomial in input size and $\frac{1}{\epsilon}$ and requires $(1+\epsilon)$-speed augmentation for processors, which means that this scheduler needs processors which are $(1 + \epsilon)$ times faster than the processors than the optimal scheduler uses.

We divide each task into equal size blocks and require the scheduler to process an integer number of blocks of each task $[s_j, s_{j+1}]$. First we show that such a scheduling exists. Then we propose a dynamic program which finds such a scheduling in time polynomial in input size and $\frac{1}{\epsilon}$ and finally we explain how it can be used to develop an algorithm which finds a scheduling with minimum makespan using binary search.

Without loss of generality we may assume that ϵ is at most $\frac{1}{2}$. Because any feasible scheduling which uses $\frac{1}{2}$-speed augmentation can be considered as a feasible scheduling which uses ϵ-speed augmentation if $\epsilon > \frac{1}{2}$, so when $\epsilon > \frac{1}{2}$ we assume that $\epsilon = \frac{1}{2}$. For every $\epsilon \in [0, \frac{1}{2}]$ the following inequality holds.

$$(1+\epsilon)(1-\frac{\epsilon}{2}) \geq 1 \tag{3.5}$$

We may also assume that tasks are ordered such that $s_1 \leq s_2 \leq ... \leq s_n$. We define I_i to be the i-th time interval, where $I_i = [s_i, s_{i+1}]$ for $1 \leq i \leq n-1$ and $I_n = [s_n, C_{max}]$. For a feasible scheduling S, let S_{ji} denote the amount of processing dedicated to T_j during I_i.

For each task T_j, we define u_j in the following way:

$$u_j = \frac{\epsilon}{2n} p_j. \tag{3.6}$$

We also define $b_j = (1+\epsilon)u_j$ and $n_j = \lceil \frac{p_j}{b_j} \rceil$. We call b_j the block size and n_j the number of blocks of task T_j. The idea behind our algorithm is to require the scheduler to process an integer number of blocks of each task in each time interval. The following lemma shows that if there is a feasible scheduling for the problem, then there is a feasible scheduling which processes an integer number of blocks of tasks in every time interval and uses $(1+\epsilon)$-speed augmentation.

Lemma 3.2.5. *If there is a feasible scheduling S for $Pm|var, lin, s_j, min_j|C_{max}$, then there is a scheduling S' with $(1+\epsilon)$-speed augmentation such that $\frac{S'_{ji}}{(1+\epsilon)u_j}$ is integer for $1 \leq i \leq n$ and $1 \leq j \leq n$.*

Proof. We know that S is a feasible scheduling for $Pm|var, lin, s_j, min_j|C_{max}$. Then

$$\sum_{i=1}^{n} S_{ji} = p_j \qquad \forall 1 \leq j \leq n \tag{3.7}$$

$$\sum_{j=1}^{n} S_{ji} \leq m|I_i| \qquad \forall 1 \leq i \leq n \tag{3.8}$$

Equation (3.7) guarantees that each task is fully processed. Equation (3.8) guarantees that in each time interval the amount of processing required for tasks is not more than number of processors m multiplied by the length of the time interval $|I_i|$.

Now we construct scheduling S' in the following way.

$$S'_{ji} = (1+\epsilon)\lfloor \frac{S_{ji}}{u_j} \rfloor u_j \tag{3.9}$$

It can be seen that $\frac{S'_{ji}}{(1+\epsilon)u_j}$ is equal to $\lfloor \frac{S_{ji}}{u_j} \rfloor$ which is an integer. Now it suffices to show that S' is a feasible scheduling for the problem with $(1+\epsilon)$-speed augmentation. To show that S' is a feasible scheduling with $(1+\epsilon)$-speed augmentation, we need to show that in

each time interval the amount of processing required for tasks is at most $(1+\epsilon)m|I_i|$ and we also need to show each task is fully processed.

First we prove that the amount of processing required for the tasks in time interval I_i is at most $(1+\epsilon)m|I_i|$ for every $1 \leq i \leq n$.

$$\begin{aligned}
\sum_{j=1}^{n} S'_{ji} &= \sum_{j=1}^{n} (1+\epsilon) \lfloor \frac{S_{ji}}{u_j} \rfloor u_j && \text{by Equation (3.9)} \\
&\leq \sum_{j=1}^{n} (1+\epsilon) \frac{S_{ji}}{u_j} u_j \\
&= (1+\epsilon) \sum_{j=1}^{n} S_{ji} \\
&\leq (1+\epsilon) m|I_i| && \text{by Equation (3.8)}
\end{aligned}$$

Now we prove that that each task is fully processed by scheduling S'. It is sufficient to show that $\sum_{i=1}^{n} S'_{ji} \geq p_j$ for every $1 \leq j \leq n$.

$$\begin{aligned}
\sum_{i=1}^{n} S'_{ji} &= \sum_{i=1}^{n} (1+\epsilon) \lfloor \frac{S_{ji}}{u_j} \rfloor u_j && \text{by Equation (3.9)} \\
&\geq (1+\epsilon) \sum_{i=1}^{n} \left(\frac{S_{ji}}{u_j} - 1 \right) u_j \\
&= (1+\epsilon) \left(\sum_{i=1}^{n} S_{ji} - \sum_{i=1}^{n} u_j \right) \\
&= (1+\epsilon) \left(p_j - \sum_{i=1}^{n} u_j \right) && \text{by Equation (3.7)} \\
&= (1+\epsilon) \left(p_j - \frac{\epsilon}{2} p_j \right) && \text{by Equation (3.6)} \\
&= (1+\epsilon) \left(1 - \frac{\epsilon}{2} \right) p_j \\
&\geq p_j && \text{by Equation (3.5)}
\end{aligned}$$

So S' is a feasible scheduling with $(1+\epsilon)$-speed augmentation. □

Assuming there is a feasible scheduling for the problem, we proved that there exists a feasible scheduling which processes an integer number of blocks of each task in each time interval and uses $(1+\epsilon)$-speed augmentation. The following lemma explains how an algorithm can find such a scheduling.

Lemma 3.2.6. *For constant m, there exists an algorithm which finds a feasible (but not necessarily optimal) scheduling for $Pm|var, lin, s_j, min_j|C_{max}$ with $(1+\epsilon)$-speed augmentation and its running time is polynomial in input size and $\frac{1}{\epsilon}$.*

Proof. We require the scheduler to process an integer number of blocks of task T_j during time interval I_i for $1 \leq i \leq n$ and $1 \leq j \leq n$.

We define a configuration to be an n-vector $c = (c_1, c_2, ..., c_n)$, in which c_j is the remaining number of blocks of T_j. A configuration shows that at a specific time, which tasks should be processed and how many blocks of them are remaining. We use $T(c)$ to denote the set of tasks which need to be scheduled in configuration c, *i.e.,* $T(c) = \{T_j | c_j \neq 0\}$. We say configuration c is feasible if $\sum_{j:T_j \in T(c)} min_j \leq m$.

Let C be the set of all feasible configurations. Since at most m tasks can be scheduled at any time the number of non-zero elements of a feasible configuration is at most m. We also know that $c_j \leq n_j$ and $n_j \leq \frac{2n}{\epsilon}$ for each task T_j. So

$$\begin{aligned} |C| &\leq \binom{n}{m} \left(\frac{2n}{\epsilon}\right)^m \\ &= \frac{n!}{(n-m)!m!} 2^m n^m \left(\frac{1}{\epsilon}\right)^m \\ &\in O\left(n^{2m} \left(\frac{1}{\epsilon}\right)^m\right), \end{aligned}$$

where m is a constant.

Let $F(c, i)$ denote the feasibility of configuration c at time s_i, which means $F(c, i) = 1$ if there is a feasible scheduling in which at time s_i configuration c is scheduled, and $F(c, i) = 0$ otherwise.

We can use a dynamic program for computing $F(c, i)$. The base cases for this dynamic program are $F(c, i)$ where $i = 1$. The only feasible configuration at time s_1 is $c_1 = (n_1, 0, 0, ..., 0)$. Because at time s_1, T_1 is just released so its remaining number of blocks is n_1 and other tasks are not released.

For $i \geq 2$, to check if configuration c is feasible at time s_i, we check if there exists a feasible configuration c' at time s_{i-1}, such that we can reach from configuration c' at time s_{i-1} to configuration c at time s_i. So:

$$F(c, i) = \sum_{c' \in C} F(c', i-1) R(c', c, i), \qquad 2 \leq i \leq n, c \in C,$$

where $R(c', c, i) = 1$ if it is possible to reach from configuration c' at time s_{i-1} to configuration c at time s_i, and $R(c', c, i) = 0$ otherwise. We can compute $R(c', c, i)$ by computing the total processing required for reaching from configuration c' to configuration c and comparing it to $(1 + \epsilon)m|I_{i-1}|$.

Algorithm 3 shows how this dynamic programming algorithm can be implemented. Function *IsReachable*(c', c, i) checks if it is possible to reach from configuration c' at time s_{i-1} to configuration c at time s_i, by comparing $(1 + \epsilon)m|I_{i-1}|$ to the total processing required for reaching from c' to c. The running time of functions *IsFeasible*(c, i) and *IsReachable*(c', c, i) is of $O(n)$. Our dynamic program calls these functions $|C|^2 n$ times, where C is the set of feasible configurations. We know that $|C| \in O\left(n^{2m} \left(\frac{1}{\epsilon}\right)^m\right)$. So the running time of the algorithm is in $O\left(n^{4m+2} \left(\frac{1}{\epsilon}\right)^{2m}\right)$. □

Algorithm 3 Dynamic Program

for all $c \in C$ **do** ▷ Initializing Base Case
    $F(c, 1) \leftarrow 0$
$c_1 \leftarrow (n_1, 0, 0, ..., 0)$
$F(c_1, 1) \leftarrow 1$
for $i \leftarrow 2$ to n **do** ▷ Dynamic Program
    for all $c \in C$ **do**
        $F(c, i) \leftarrow 0$
        if IsFeasible$(c, i) =$ False **then**
            Continue
        for all $c' \in C$ **do**
            if $F(c', i-1) = 1$ and IsReachable(c', c, i) **then**
                $F(c, i) \leftarrow 1$
function IsFeasible(c, i) ▷ Checks if c is feasible
    if $c_i \neq n_i$ **then**
        return False
    for all $j \leftarrow i + 1$ to n **do**
        if $c_j \neq 0$ **then**
            return False
    $Sum \leftarrow 0$
    for all $j \leftarrow 1$ to n **do**
        if $c_j \neq 0$ **then**
            $Sum \leftarrow Sum + min_j$
    if $Sum > m$ **then**
        return False
    return True
function IsReachable(c', c, i) ▷ Checks if c is reachable from c'
    $TotalProcess \leftarrow 0$
    for all $j \leftarrow 1$ to $i - 1$ **do**
        $Process \leftarrow (c_j - c'_j)b_j$
        if $c_j \neq 0$ **then**
            $Process \leftarrow max\{Process, min_j|I_{i-1}|\}$
        else
            $Process \leftarrow max\{Process, 0\}$
        $TotalProcess \leftarrow TotalProcess + Process$
    if $TotalProcess > (1 + \epsilon)m|I_{i-1}|$ **then**
        return False
    return True

Now we use this algorithm to develop an algorithm which finds a scheduling with minimum makespan using a binary search method.

Theorem 3.2.7. *For constant m, there is an algorithm which finds a feasible scheduling with minimum makespan for $Pm|var, lin, s_j, min_j|C_{max}$ with $(1+\epsilon)$-speed augmentation in time polynomial in input size and $\frac{1}{\epsilon}$.*

Proof. Let p_{max} be the maximum processing time, *i.e.,* $max_j\{p_j\}$. We know that makespan is at least s_n and at most $s_n + p_{max}$. We use binary search to find the minimum time we can finish all the tasks.

We add another task T_{n+1} to the set of tasks, where $min_{n+1} = m$, $p_{n+1} = m$, and $s_{n+1} = s_n + X$ for some positive X. Since this task needs all of processors, other tasks should be finished by starting time of this task. If the algorithm described in Lemma 3.2.6 cannot find a feasible scheduling for $T \cup \{T_{n+1}\}$, it means that there is no feasible scheduling for T with makespan $s_n + X$.

Assuming that there is a feasible scheduling for T, we know that the minimum makespan is at least s_n and at most $s_n + p_{max}$. So we know the minimum X for which there is feasible scheduling is at least 0 and at most p_{max}. We can use a binary search to find the minimum X. Algorithm 4 shows how we can implement the binary search method for this problem.

Algorithm 4 Binary Search

```
1: min_{n+1} ← m
2: p_{n+1} ← m
3: MinX = BinarySearch(0, p_max + 1)
4: if MinX = p_max + 1 then
5:     No Feasible Solution
6: else
7:     Makespan = s_n + MinX
8: function BinarySearch(f, l)
9:     if f + 1 = l then
10:        return l
11:    x ← (f+l)/2
12:    s_{n+1} ← s_n + x
13:    if DynamicProgram(T ∪ T_{n+1}) then
14:        return BianrySearch(f, x)
15:    else
16:        return BianrySearch(x, l)
```

Notice that if function *BinarySearch* does not find any feasible scheduling for a positive X, it returns $p_{max} + 1$. In this case there is no feasible scheduling for T. The algorithm executes the dynamic program $O(\log p_{max})$ times. So the running time of the algorithm is in $O\left(n^{4m+2}\left(\frac{1}{\epsilon}\right)^{2m}\log p_{max}\right)$. □

3.3 Flow time

In this section we consider the problem of minimizing the maximum flow time for rigid parallel tasks, *i.e.,* $P|size_j, r_j|F_{max}$. In this problem, each task T_j should be scheduled after its release time r_j on exactly $size_j$ processors.

In Section 3.3.1 we consider the complexity of $P|size_j, r_j|F_{max}$ and we show that this problem is strongly NP-hard. In Section 3.3.2 we propose a bicriteria $(1+\epsilon, O(1/\epsilon))$-approximation algorithm for the problem, which finds a solution within a factor of $1+\epsilon$ of the optimal solution using $O(1/\epsilon)$-speed augmentation.

3.3.1 Complexity of $P|size_j, r_j|F_{max}$

We show that $P|size_j, r_j|F_{max}$ is strongly NP-hard by reducing the Bin packing problem to it. The Bin packing problem is known to be strongly NP-hard [26].

Given a list of objects and their weights, and a collection of bins of fixed size, find the smallest number of bins so that all of the objects are assigned to a bin.

Bin packing problem: Given a list $A = \{a_1, a_2, ..., a_n\}$ of integer numbers, and an integer number V, find the minimum integer B and a B-partition $S_1 \cup ... \cup S_B$ of $\{1, 2, ..., n\}$ such that $\sum_{j \in S_i} a_j \leq V$ for all $1 \leq i \leq B$.

Theorem 3.3.1. *The scheduling problem $P|size_j, r_j|F_{max}$ is strongly NP-hard.*

Proof. We reduce the bin packing problem to this problem. Given an instance I of the bin packing problem, we construct an instance I' of the scheduling problem as follows. Let $m = V$ and $T = \{T_1, T_2, ..., T_n\}$, where $r_j = 0$, $size_j = a_j$, and $p_j = 1$. We should show that for every solution for I with B bins, there exists a scheduling for I' with maximum flow time of B, and vice versa.

First we show that if instance I has a solution with B bins, there exists a feasible scheduling for I' with maximum flow time of B. Let $S_1 \cup ... \cup S_B$ be the B-partition for I. If $j \in S_i$, then we schedule task T_j in time interval $[i-1, i]$. Since $\sum_{j \in S_i} a_j \leq V$, at any time the number of processors assigned to the tasks is at most V and this scheduling is a feasible scheduling. We scheduled all the tasks during time interval $[0, B]$, so the maximum flow time is equal to B.

Now we assume there exists a feasible scheduling for I' with maximum flow time of B and show that there is a solution for I with B bins. If T_j is scheduled during time interval $[i-1, i]$, we put j in S_i. The total number of required processors for the tasks scheduled during $[i-1, i]$ is equal to $\sum_{j \in S_i} a_j$ and it cannot be more than the number of processors V, so the total size of the items in S_i does not exceed V and it is a feasible packing for I. We know that all tasks are scheduled during $[0, B]$, so the number of bins is equal to B. □

Corollary 3.3.2. *The problems $P|size_j, r_j|C_{max}$ and $P|size_j, r_j|L_{max}$ are strongly NP-hard.*

Proof. Notice that in the constructed instance of the scheduling problem, F_{max} equals C_{max} and L_{max}. □

3.3.2 Bicriteria $(1+\epsilon, O(1/\epsilon))$-approximation for $P|size_j, r_j|F_{max}$

In this section we propose an approximation algorithm for $P|size_j, r_j|F_{max}$which finds a solution within a factor of $1+\epsilon$ of the optimal solution using $O(1/\epsilon)$-speed augmentation and runs in time polynomial in input size and $1/\epsilon$.

Our algorithm is based on an LP-relaxation where preemption of tasks is allowed. Using the solution of the LP, we create a preemptive scheduling for an instance of the scheduling on heterogeneous platforms (SPP) and use the rounding method proposed by Bougeret *et al.* [14] to obtain a non-preemptive scheduling for SPP. Finally we create a non-preemptive solution for our problem using the non-preemptive scheduling of SPP.

Scheduling on heterogeneous platforms (SPP): Given a set of tasks $T_1, ..., T_n$ that have to be scheduled on a set of platforms $P_1, ..., P_N$, find a scheduling in which the maximum completion time of the tasks is minimum. A platform P_l contains a set of m_l processors. A task T_j is described by a pair (p_j, q_j) where p_j is its processing time and q_j is its degree of parallelism on all platforms.

First we show that the preemptive version of $P|size_j, r_j|F_{max}$ can be solved polynomially.

Lemma 3.3.3. *Problem $P|size_j, r_j, pmtn|F_{max}$ with n tasks and m processors can be solved in time polynomial in n and m.*

Proof. Note that this problem is a special case of $P|var, r_j|F_{max}$ in which the processing time function of task T_j has the following form.

$$p_j(l) = \begin{cases} p_j & l = size_j \\ \infty & otherwise \end{cases}$$

where l is the number of processors assigned to task T_j. In Section 2.4 we showed that $P|var, r_j|F_{max}$ can be solved in time polynomial in n and m, so $P|size_j, r_j, pmtn|F_{max}$ can be solved in polynomial time. □

Let S denote the preemptive scheduling produced by Lemma 3.3.3. We divide S into $R = \frac{2C_{max}}{\epsilon F_{max}}$ intervals of length $L = \frac{\epsilon F_{max}}{2}$. Let I_i denote the i-th interval, where $I_i = [(i-1)L, iL]$. Now we create an instance of SPP with R platforms where each platform has m processors. The set of tasks for SPP is the same set of tasks we have in our problem. We create a preemptive scheduling for SPP by scheduling time interval I_i on platform P_i. Since $|I_i| = L$, the constructed preemptive scheduling of SPP is of length L.

Let $p_{max} = max_{1 \leq j \leq n}\{p_j\}$. We use the following result proved by Bougeret *et al.* [14].

Theorem 3.3.4. *For every $\epsilon' \in (0, 1]$ there exists an approximation algorithm with running time polynomial in input size and $\frac{1}{\epsilon'}$ that for every preemptive schedule of length L of SPP produces a non-preemptive schedule of length at most $(1+\epsilon')L + O(\frac{1}{\epsilon'^2})p_{max}$, such that the non-preemptive schedule does not schedule task T_j on platform P_i if the preemtive schedule does not schedule task T_j on platform P_i.*

Let L' denote the length of the non-preemptive schedule produced with $\epsilon' = 1$.

$$\begin{aligned} L' &\leq (1+\epsilon')L + O(\frac{1}{\epsilon'^2})p_{max} \\ &\leq 2L + c\ p_{max}, \end{aligned}$$

where c is a constant. Since $L = \frac{\epsilon F_{max}}{2}$ and $p_{max} < F_{max}$, we know that $p_{max} < \frac{2L}{\epsilon}$. So

$$\begin{aligned} L' &\leq 2L + c\ \frac{2L}{\epsilon} \\ &\leq (2 + \frac{2c}{\epsilon})L \\ &\leq \frac{c'}{\epsilon}L \end{aligned}$$

where c' is a constant. Now we use this non-preemptive schedule of SPP to create a non-preemptive solution for our problem with $\frac{c'}{\epsilon}$-speed augmentation. The length of the non-preemptive scheduling of SPP is at most $\frac{c'}{\epsilon}L$. With $\frac{c'}{\epsilon}$-speed augmentation we can schedule tasks on platform P_i in time interval $[iL, (i+1)L]$.

Let S' denote this non-preemptive schedule. To show that S' is a feasible schedule, we have to show that S' does not schedule a task before its release time. Assume that S' schedules T_j in time interval $[iL, (i+1)L]$, then task T_j is scheduled on platform P_i. Which means that preemptive schedule S schedules task T_j in time interval $[(i-1)L, iL]$. So $r_j < iL$ and task T_j is scheduled after its release time by S'.

Now we show that the maximum flow time of S' is within a factor of $1+\epsilon$ of the optimal solution. Let C_j and C'_j denote the completion time of T_j in S and S', respectively. If T_j is scheduled during time interval $[iL, (i+1)L]$ in S', it is scheduled during time interval $[(i-1)L, iL]$ in S. So $C'_j \leq C_j + 2L$. Let F_{max} and F'_{max} denote the maximum flow time of S and S' respectively.

$$\begin{aligned} F'_{max} &= max_{1 \leq j \leq n}\{C'_j - r_j\} \\ &\leq max_{1 \leq j \leq n}\{C_j + 2L - r_j\} \\ &= 2L + max_{1 \leq j \leq n}\{C_j - r_j\} \\ &= 2\frac{\epsilon F_{max}}{2} + F_{max} \\ &= (1+\epsilon)F_{max}. \end{aligned}$$

Since any non-preemptive solution is also a preemptive solution for the problem, F_{max} is a lower bound for the optimal solution and F'_{max} is within a factor of $1+\epsilon$ of the optimal solution.

The running time of finding the optimal preemptive schedule is polynomial in n and m. The running time of constructing the non-preemptive schedule is polynomial in n, m, and R, where $R = \frac{2}{\epsilon}\frac{C_{max}}{F_{max}}$. To show that the algorithm is polynomial in input size and $1/\epsilon$, it suffices to show that $\frac{C_{max}}{F_{max}}$ is polynomially bounded by n.

Assume the tasks are sorted by non-decreasing order of their release time, *i.e.,* $r_1 \leq r_2 \leq ... \leq r_n$. We may assume that $r_1 = 0$. We may also assume that $r_{i+1} < r_i + i\ p_{max}$, because if $r_{i+1} \geq r_i + i\ p_{max}$, then we have enough time to schedule the first i tasks before T_{i+1} arrives and we can divide the tasks into two groups $\{T_1, ..., T_i\}$ and $\{T_{i+1}, ..., T_n\}$, and solve the problem for them separately. With similar reasoning, we know that $C_{max} < r_n + n\ p_{max}$. So

$$\begin{aligned} C_{max} &\leq p_{max} + 2\ p_{max} + ... + n\ p_{max} \\ &= O(n^2).p_{max} \\ &= O(n^2).F_{max} \end{aligned}$$

So $\frac{C_{max}}{F_{max}}$ is in $O(n^2)$ and the algorithm is polynomial in input size and $1/\epsilon$.

Chapter 4

Online Scheduling

One of the disadvantages of offline scheduling problems is their assumption on the availability of tasks' characteristics. In reality we usually cannot predict the future and the data on the future tasks are unknown. For example, the process scheduler of an operating system receives tasks that arrive over time, and generally must schedule the tasks without any knowledge of the future tasks.

In online scheduling problems, the scheduler does not have access to the entire input instance initially and the scheduler will have access to tasks' characteristics when the tasks arrive. The scheduler does not know about the future tasks and may make its decisions online based on the currently available information.

Online scheduling problems can be divided into two groups based on the way the information is released: *online-time* and *online-list*. In *online-time* scheduling problems, each task has a release time and the scheduler is unaware of the existence of a task and its characteristics before its release time. In contrast, in *online-list* scheduling problems, the scheduler must schedule tasks from some queue or list one by one. The characteristics of a task becomes known after scheduling the previous tasks on the list.

In *online-time* scheduling problems, once a task is released, the scheduler learns about its characteristics. In reality it is sometimes difficult to gather information about the processing time of tasks. For example the process scheduler of an operating system does not know the processing time of a task even after its release time. This lack of information about the processing time of tasks is called nonclairvoyance. In a nonclairvoyant online scheduling problem, the scheduler does not know tasks' processing times even after their release times. In contrast, in a clairvoyant online scheduling problem the scheduler knows p_j at release time of T_j. We denote nonclairvoyant scheduling problem by *online-time-nclv*.

For some online scheduling problems finding optimum online scheduling is possible. For example $1|\textit{online-time}, r_j, \textit{pmtn}|F_{max}$, which is the problem of minimizing the maximum flow time for online preemtive tasks on one processor, can be solved optimally by an online algorithm [49]. But for most online scheduling problems, because of the lack of information

about the future tasks, finding the optimal schedule for all input instances is impossible. For example no online scheduler can solve $P2|\textit{online-time}, r_j, pmtn|L_{max}$ optimally for all input instances [18].

For the scheduling problems for which an optimum online scheduler cannot be constructed, we use competitive analysis to evaluate the worst-case performance of the online scheduler. Competitive ratio is used to show the worst-case performance of an online algorithm similarly to the approximation ratio of an offline algorithm. More precisely, a *c-competitive* online algorithm finds a solution online which is at most c times worse than the optimum solution for any input instance.

For some online problems, a satisfactory competitive ratio cannot be achieved because of the nature of the problem. It can even be impossible for an online algorithm to find a feasible solution for some problems. To overcome this obstacle Kalyanasundaram and Pruhs [41] introduced *resource augmentation* for online scheduling problems. In resource augmentation techniques the online scheduler is allowed to use more resources than the optimal offline algorithm to which it is compared, e.g., more processors, faster processors, to achieve a better competitive ratio.

Let $A(m, I)$ denote the scheduling that online scheduling algorithm A outputs for input instance I on m processors. We say online scheduling algorithm A is (v, c)-competitive if it constructs a scheduling using vm processors which is at most c times worse than the optimum scheduling using m processors, *i.e.,*A is (v, c)-competitive if

$$f(A(vm, I)) \leq c \,.\, f(OPT(m, I)) \qquad \forall \text{ input instance } I$$

where f is the objective function of the problem.

In this chapter we consider the online problem $P|\textit{online-time}, var, lin, s_j, min_j|C_{max}$, which is the online scheduling problem of minimizing the maximum completion time for tasks with minimum degree of parallelism and linear speed-up. In Section 4.2 we show that no online algorithm can find any feasible schedule for all input instances of this problem. In Section 4.3 we show that an online algorithm for this problem needs at least $\frac{9-\sqrt{17}}{4}$ processor augmentation. In Section 4.4 we propose a $(O(\log m), 1)$-competitive algorithm for the problem.

4.1 Related works

In the *online-list* scheduling problems, tasks are ordered in a list and the scheduler should schedule the tasks one by one according to the list. Each task should be scheduled before the next tasks are seen and the scheduler cannot change the scheduling of the previous tasks.

Problem $P|\textit{online-list}|C_{max}$ is one of the most intensively studied *online-list* problems. Graham [30] showed that *LIST* algorithm which always assigns the next task to the least

loaded processor is a $(2-\frac{1}{m})$-competitive algorithm for the problem. Galambos and Woeginger [25] proposed a $(2-\frac{1}{m}-\epsilon_m)$-competitive algorithm, where $\epsilon_m > 0$, but ϵ_m tends to 0 as m goes to infinity. This result was improved by a series of papers [8, 42, 1] and finally online algorithm with competitive ratio 1.9201 was proposed by Fleischer and Wahl [24].

Regarding the lower bound for the competitive ratio of an online algorithm for problem $P|\textit{online-list}|C_{max}$, Faigle *et al.* [23] showed that 1.707 is a lower bound for $m \geq 4$. This bound was improved to 1.837 for $m \geq 3454$ by Bartal *et al.* [9], to 1.852 for $m \geq 80$ by Albers [1], and to 1.85358 for $m \geq 80$ by Gormley *et al.* [29]. Finally Rudin and Chandrasekaran [50] proved that no online algorithm for the problem can be better than 1.88-competitive.

Azar *et al.* [5] considered a restricted assignment variant of $P|\textit{online-list}|C_{max}$ in which each task cannot be scheduled on some of the processors. They showed that *LIST* algorithm is $(\lceil \log_2 m \rceil + 1)$-competitive for this variant of the problem and showed that no online scheduler can be better than $(\lceil \log_2 m \rceil)$-competitive.

Aspnes *et al.* [4] proposed an 8-competitive algorithm for $Q|\textit{online-list}|C_{max}$. Blum *et al.* [13] improved this result and proposed a 5.828-competitive algorithm. Aspnes *et al.* [4] also proposed a $O(\log m)$-competitive algorithm for unrelated processors $R|\textit{online-list}|C_{max}$.

In the *online-time* scheduling problems, tasks are released over time and the scheduler is not aware of the existence of the tasks before their release times. After a task is released the scheduler has access to its characteristics.

Problem $P|\textit{online-time}, r_j, \textit{pmtn}|C_{max}$ can be solved optimally [33]. For uniform processors $Q|\textit{online-time}, r_j, \textit{pmtn}|C_{max}$ there is a $(1+\epsilon)$-competitive online algorithm [44]. For the non-preemptive case $P|\textit{online-time}, r_j|C_{max}$ there is a 1.5-competitive algorithm and it is shown that no online algorithm can be better than 1.3473-competitive [16].

For minimizing total completion time, *SRPT* algorithm which schedules the task with shortest remaining processing time first, constructs the optimum scheduling for problem $1|\textit{online-time}, r_j, \textit{pmtn}|\sum_j C_j$. SRPT algorithm is also a 2-competitive algorithm for problem $P|\textit{online-time}, r_j, \textit{pmtn}|\sum_j C_j$ [48]. For the non-preemptive scheduling, there is a 2-competitive algorithm and it is shown that no online algorithm can be better than 2-competitive [34]. For minimizing total weighted completion time a $(1+\sqrt{2})$-competitive algorithm is proposed by Goemans [28].

Minimizing total flow time is harder than minimizing total completion time. For problem $1|\textit{online-time}, r_j|\sum_j F_j$ no online algorithm can be better than $(n-1)$-competitive and even for the offline case there is no algorithm with approximation ratio better than $(\sqrt{n})$ [52]. Leonardi [46] proposed an online algorithm with competitive ratio $\Theta(\log \frac{n}{m})$ for $P|\textit{online-time}, r_j, \textit{pmtn}|\sum_j F_j$. If we allow the algorithm to use processors with speed-up factor $1+\epsilon$, there is a $(1+\frac{1}{\epsilon})$-competitive algorithm for $P|\textit{online-time}, r_j, \textit{pmtn}|\sum_j F_j$ [41].

Another objective function which is studied in the context of file transfer over a network is the maximum stretch [27]. The stretch for a task is the ratio of between the execution time of the task to the processing time of the task (*i.e.*, $\frac{C_j - r_j}{p_j}$). Bender *et al.* [10] studied the maximum stretch as a performance measure for online scheduling problems. The average stretch also has been used to evaluate the scheduling algorithms. Muthukrishnan *et al.* [47] analyzed SRPT (Shortest remaining processing time) algorithm and showed that this algorithm is a constant factor competitive algorithm for minimizing average stretch on uniprocessor and multiprocessor systems.

4.2 Scheduling of $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$

In this section we show that problem $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$ cannot be solved by an online algorithm. We create an input instance with two scenarios and demonstrate that no online scheduler can find feasible schedules for both scenarios.

Theorem 4.2.1. *No online algorithm can find feasible scheduling for all input instances of problem* $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$.

Proof. We denote task T_j by a 3-vector (s_j, min_j, p_j), where s_j is the start time, min_j is the minimum degree of parallelism, and p_j is the processing time of T_j. Let $m = 4$ and suppose there are 4 tasks.

$$T_1 = (0, 1, 4)$$
$$T_2 = (0, 2, 9)$$
$$T_3 = (2, 2, 2)$$
$$T_3' = (3, 3, 3)$$

Tasks T_1 and T_2 are known at time $t = 0$, and T_3 and T_3' may arrive in different scenarios. Suppose the online algorithm does not schedule task T_1 on two processors during time interval $[0, 2]$, then if task T_3 arrives at time $t = 2$, there is not enough available processors to schedule task T_3. However if the scheduler had scheduled T_1 on two processors during $[0, 2]$, then a feasible schedule could have been constructed as in Figure 4.1a.

Now assume that the algorithm schedules T_1 on two processors during time interval $[0, 2]$, then task T_2 cannot be finished at time $t = 3$ and if task T_3' arrives at time $t = 3$, it cannot be scheduled. Had the algorithm scheduled T_1 on one processor during $[0, 2]$, task T_2 could be finished at $t = 3$ and T_3' could be scheduled as in Figure 4.1b.

Whatever decision the algorithm makes during time interval $[0, 2]$, there is a scenario for which there is not a feasible schedule. So there is no online algorithm to a find feasible scheduling for all input instances of $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$. □

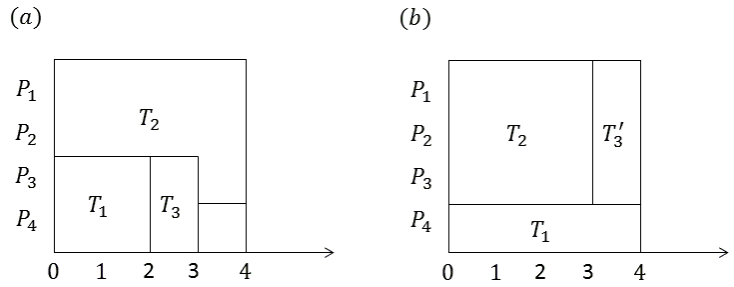

Figure 4.1: Online Scheduling

4.3 Lower-bound for processor augmentation

In Section 4.2 we showed that an online algorithm cannot find a feasible schedule for all input instances of problem $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$, So it is reasonable to use resource augmentation techniques and allow the online algorithm to use extra processors.

In this section we show that the processor augmentation should be at least $\frac{9-\sqrt{17}}{4}$ for an online algorithm to find a feasible solution for problem $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$.

Theorem 4.3.1. *For $v < \frac{9-\sqrt{17}}{4}$, there is no (v, c)-competitive online algorithm for problem $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$ for any $c \geq 1$.*

Proof. Assume that for some $v < \frac{9-\sqrt{17}}{4}$ and some $c \geq 1$, there exists a (v, c)-competitive algorithm for the problem. We create an input instance with two scenarios and show that the algorithm cannot find a feasible schedule for both scenarios.

We choose m such that $m > 3/(\frac{9-\sqrt{17}}{4} - v)$. So

$$\begin{aligned} \frac{9-\sqrt{17}}{4}m &= vm + (\frac{9-\sqrt{17}}{4} - v)m \\ &\geq vm + 3 \end{aligned} \tag{4.1}$$

By Equation (4.1), we know that the difference between $\frac{9-\sqrt{17}}{4}m$ and vm is at least 3, so there exist at least two integer numbers between $\frac{9-\sqrt{17}}{4}m$ and vm and we can find a positive integer a such that

$$vm < m + a - 1 < m + a < \frac{9-\sqrt{17}}{4}m.$$

We give the algorithm $a - 1$ extra processors. Note that

$$a < (\frac{5-\sqrt{17}}{4})m. \tag{4.2}$$

We denote task T_j by a 3-vector (s_j, min_j, p_j). Suppose there are 4 tasks.

$$
\begin{aligned}
T_1 &= (0, a, p_1) \\
T_2 &= (0, 2a, p_2) \\
T_3 &= (\frac{p_1}{m-2a}, m-2a, p_3) \\
T_3' &= (\frac{p_2}{m-a}, m-a, p_3')
\end{aligned}
$$

Lemma 4.3.2. *We can choose p_1 and p_2 such that:*

$$\frac{m-1}{m-2a}p_1 < p_2 < \frac{m-a}{2a-1}p_1 \tag{4.3}$$

Proof. First we show that $\frac{m-1}{m-2a} < \frac{m-a}{2a-1}$.

$$
\begin{aligned}
\frac{m-1}{m-2a} &= \left(\frac{m-1}{m-2a}\right)\left(\frac{m-a}{m-a}\right)\left(\frac{2a-1}{2a-1}\right) \\
&= \frac{m-a}{2a-1} \cdot \frac{(2a-1)(m-1)}{(m-a)(m-2a)} \\
&< \frac{m-a}{2a-1} \cdot \frac{2am}{(m-a)(m-2a)} \\
&\leq \frac{m-a}{2a-1} \cdot \frac{\frac{5-\sqrt{17}}{2}}{(1-\frac{5-\sqrt{17}}{4})(1-\frac{5-\sqrt{17}}{2})} && \text{By Equation (4.2)} \\
&\leq \frac{m-a}{2a-1}
\end{aligned}
$$

Now we choose p_1 such that $p1 > 2/(\frac{m-a}{2a-1} - \frac{m-1}{m-2a})$, then $\frac{m-1}{m-2a}p_1 + 2 < \frac{m-a}{2a-1}p_1$. Which means there exist an integer between $\frac{m-1}{m-2a}p_1$ and $\frac{m-a}{2a-1}p_1$. we choose p_2 to be that integer. □

Tasks T_1 and T_2 are known at time $t = 0$, and T_3 and T_3' may arrive in different scenarios. Note that $s_3 < s_3'$ because

$$
\begin{aligned}
s_3 &= \frac{p_1}{m-2a} \\
&= (\frac{1}{m-1})(\frac{m-1}{m-2a})\ p_1 \\
&\leq (\frac{1}{m-1})\ p_2 && \text{by Equation (4.3)} \\
&\leq \frac{p_2}{m-a} \\
&= s_3'
\end{aligned}
$$

First we show that both scenarios $\{T_1, T_2, T_3\}$ and $\{T_1, T_2, T_3'\}$ can be scheduled on m processors. If the scheduler schedules T_1 on $1-2a$ processors and T_2 on $2a$ processors during time interval $[0, \frac{p_1}{m-2a}]$, task T_1 will be finished at $t = \frac{p_1}{m-2a}$ and if task T_3 arrives, it can be scheduled. So $\{T_1, T_2, T_3\}$ can be scheduled on m processors.

For the second scenario, if the scheduler schedules T_1 on a processors and T_2 on $1-a$ processors during time interval $[0, \frac{p_2}{m-a}]$, task T_2 will be finished at $t = \frac{p_2}{m-a}$ and if task T_3' arrives, it can be schedules. So $\{T_1, T_2, T_3'\}$ also can be scheduled on m processors.

Now we show that the online scheduler cannot schedule tasks in both scenarios on $m + a - 1$ processors. If task T_3 arrives at time $t = \frac{p_1}{m-2a}$, at least one of the tasks T_1 and T_2 should be finished, otherwise, there is not enough available processors to schedule task T_3. During time interval $[0, \frac{p_1}{m-2a}]$, task T_2 can be scheduled on at most $m-1$ processor. So task T_2 cannot be finished before $\frac{p_1}{m-2a}$, because

$$\frac{p_1}{m-2a}(m-1) < p_2 \qquad \text{by Equation (4.3)}$$

Which means task T_1 should be finished by time $t = \frac{p_1}{m-2a}$.

On the other hand, if task T_3' arrives, task T_2 should be finished, because $min_2 + min_3' > m + a - 1$ and they cannot be scheduled simultaneously. We showed that task T_1 should be finished by time $t = \frac{p_1}{m-2a}$. So by time $t = \frac{p_2}{m-a}$ both T_1 and T_2 should be finished. To show that both of them cannot be finished by time $t = \frac{p_2}{m-a}$, it suffices to show $p_1 + p_2 > \frac{p_2}{m-a}(m + a - 1)$.

$$\begin{aligned} p_1 + p_2 &\geq \frac{2a-1}{m-a} p_2 + p_2 && \text{by Equation (4.3)} \\ &= \frac{2a-1+m-a}{m-a}\ p_2 \\ &= \frac{p_2}{m-a}(m+a-1) \end{aligned}$$

So the online algorithm cannot schedule both $\{T_1, T_2, T_3\}$ and $\{T_1, T_2, T_3'\}$ on $m + a - 1$ processors, which means there is no (v, c)-competitive algorithm for the problem for any $v < \frac{9-\sqrt{17}}{4}$. □

4.4 A $(O(\log m), 1)$-competitive algorithm

In this section we propose a $(O(\log m), 1)$-competitive online algorithm for the scheduling problem $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$. First we divide the tasks into $\lfloor \log_2(m) \rfloor + 2$ groups. Then we show that we can schedule each group online on $2m$ processors.

We divide the tasks based on their minimum degree of parallelism min_j. We partition T into $\lfloor \log_2(m) \rfloor + 2$ subsets in the following way.

$$T(i) = \begin{cases} \{T_j | min_j = 0\} & i = 0 \\ \{T_j | 2^{i-1} \leq min_j < 2^i\} & 1 \leq i \leq \lfloor \log_2(m) \rfloor + 1 \end{cases}$$

Note that if there is a feasible scheduling for T on m processors, then there is a feasible scheduling for each subset $T(i)$ on m processors.

Lemma 4.4.1. *There exists a feasible scheduling for $T(i)$ on $2m$ processors which assigns at least 2^i processors to each task T_j during its execution.*

Proof. Let S be a feasible scheduling for $T(i)$ on m processors. We create another scheduling S' in the following way. At any time t, for every task $T_j \in T(i)$, if S assigns at least 2^i processor to T_j, S' assigns the same number of processors to T_j. Otherwise, S' assigns 2^i processors to T_j. Let $P(S, T_j, t)$ denote the number of processors schedule S assigns to task T_j at time t. Then

$$\begin{aligned}\sum_{T_j \in T(i)} P(S', T_j, t) &\leq \sum_{T_j \in T(i)} 2P(S, T_j, t) \\ &= 2 \sum_{T_j \in T(i)} P(S, T_j, t) \\ &\leq 2m\end{aligned}$$

So S' does not use more than $2m$ processors at any time and it is a feasible schedule on $2m$ processors. □

Now that we showed that a scheduling for $T(i)$ on $2m$ processors exists which assigns at least 2^i processors to each task $T_j \in T(i)$ during $[s_j, C_j]$, it suffices to develop an online algorithm to find such a scheduling. Note that $min_j < 2^i$ for all $T_j \in T(i)$, so we can assume that the minimum degree of parallelism for all tasks in $T(i)$ is 2^i now. The following lemma shows that when the minimum degree of parallelism of all tasks are equal, SRPT (shortest remaining processing time) algorithm is optimal.

Lemma 4.4.2. *SRPT algorithm is optimal for* $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$, *when* $min_i = c$ *for* $1 \leq i \leq n$.

Proof. SRPT algorithm assigns c processors to each task and assigns the remaining free processors to the task with the shortest remaining processing time. Assume that SRPT is not optimal and let S be the optimal scheduling. Let t be the smallest number such that at time t scheduling S does not assign the free processors to the the task with the shortest remaining processing time. let T_i be the task with the shortest remaining processing time and T_j be the task to which scheduling S assigns the free processors. Also let C_i and C_j denote the completion time of T_i and T_j, respectively.

Assume that $C_i \leq C_j$. If we assign the free processors at time t to T_i, task T_i will finish at time C_i', such that $C_i' < C_i$. Then we can use the processors that were assigned to T_i during time interval $[C_i', C_i]$ for processing T_j and the completion time of T_j does not change. So if we assign the free processors at time t to T_i, completion time of T_i decreases and completion time of T_j does not change.

Now assume $C_i > C_j$. Since the remaining processing time of T_i is less than T_j, if we use the processors assigned to T_j during $[t, C_j]$ to process T_i, task T_i will finish at time C_i', such that $C_i' < C_j$. Then we can use the processors that were assigned to T_i during $[t, C_i]$ and the processors that were assigned to T_j during $[C_i', C_j]$ to process T_j and T_j will finish at

time C_i, because we do not assign any processor previously dedicated to T_i and T_j to other tasks and we still should be able to finish T_i and T_j using the same processors during time interval $[t, C_i]$. So the new completion time of T_i is less than C_j and the new completion time of T_j is equal to C_i.

In both cases, we can assign the free processors to T_i and the scheduling stays optimal. So SRPT algorithm is optimal for problem $P|\mathit{online\text{-}time}, var, lin, s_j, min_j|C_{max}$, when $min_i = c$ for $1 \leq i \leq n$. □

Algorithm 5 shows how this online algorithm can be implemented. The first loop partitions T into $\lfloor \log_2(m) \rfloor + 2$ subsets based on their minimum degree of parallelism min_j. The second loop schedules each subset on $2m$ processors based on SRPT algorithm. For each subset $T(i)$, it assigns 2^i processors to each task in $T(i)$, then it assigns the remaining $2m - 2^i|T(i)|$ processors to the task with the smallest remaining processing time in $T(i)$.

Algorithm 5 $(O(\log m), 1)$-competitive online algorithm

```
1: for T_j ∈ T do
2:     if min_j = 0 then
3:         T(0) ← T(0) ∪ T_j
4:     else
5:         T(⌊log_2(min_j)⌋ + 1) ← T(⌊log_2(min_j)⌋ + 1) ∪ T_j
6: for i ← 0 to ⌊log_2(m)⌋ + 1 do
7:     freeProc ← 2m
8:     for T_j ∈ T(i) do
9:         assign 2^i processors to T_j
10:        freeProc ← freeProc − 2^i
11:    assign freeProc processors to the task with minimum remaining processing time in
       T(i)
```

The running time of the first loop is in $O(|T|)$ and the running time of the second loop is in $O(\sum_{i=0}^{\lfloor \log_2(m) \rfloor + 1} |T(i)|)$. So the running time of Algorithm 5 is in $O(|T|)$. When a new task arrives or a task finishes, Algorithm 5 should be executed to reschedule the tasks on the processors. So it is executed $2|T|$ times and the running time of the online algorithm is in $O(|T|^2)$.

Chapter 5

Conclusion

We conclude by summarizing the results of this thesis and discussing future work directions for the problems considered.

5.1 Summary

In this thesis, we studied hardness of approximation and polynomial-time algorithms for several scheduling problems of multi-processor tasks on parallel identical processors.

In Chapter 2 we considered some preemptive scheduling problems with different objective functions and solved them in time polynomial in input size using Linear Programming. Three preemptive scheduling problems with different objective function were considered:

- $P|var, r_j|C_{max}$: The problem of minimizing the maximum completion time for preemptive malleable tasks.
- $P|var, r_j, d_j|L_{max}$: The problem of minimizing the maximum latency for preemptive malleable tasks.
- $P|var, r_j|F_{max}$: The problem of minimizing the maximum flow time for preemptive malleable tasks.

We assumed that the tasks have release times and are malleable with arbitrary speed-up function. Since the speed-up function is arbitrary, it can also be used to solve scheduling of rigid tasks.

In Chapter 3 we studied some non-preemptive scheduling problems, which are related to the starvation problem in scheduling tasks. Two solutions for starvation problem were studied. One solution is to use objective functions like the maximum flow time. The other solution is to ensure that each task is scheduled on at least some minimum number of processors at each time. We considered complexity and proposed approximation algorithms for two non-preemptive scheduling problems:

- $P|var, s_j, min_j|C_{max}$: The problem of minimizing the maximum completion time of non-preemptive malleable tasks with minimum degree of parallelism.

- $P|size_j, r_j|F_{max}$: The problem of minimizing the maximum flow time of non-preemptive rigid tasks.

For the first problem, we showed that finding a feasible solution for $P|var, s_j, min_j|C_{max}$ is strongly NP-hard. We also showed that the problem is NP-hard when the processing time function is work preserving. We proposed an algorithm for $Pm|var, lin, s_j, min_j|C_{max}$ which needs $(1+\epsilon)$-speed augmentation and runs in time polynomial in n and $\frac{1}{\epsilon}$.

For the second problem, we showed that the problem is strongly NP-hard and propose a bicriteria $(1+\epsilon, O(1/\epsilon))$-approximation algorithm for the problem.

Finally in Chapter 4 we considered online scheduling of malleable tasks with minimum degree of parallelism, *i.e.,* $P|online\text{-}time, var, lin, s_j, min_j|C_{max}$. We showed that speed augmentation is necessary and no online algorithm can schedule the tasks feasibly without speed augmentation. We also found a lower bound for speed augmentation and proposed an online algorithm with $O(\log(n))$-speed augmentation which finds an optimal scheduling.

5.2 Direction for Future Work

An interesting open problem is to find a Linear Program for preemptive scheduling of moldable tasks. While the Linear Programs in Chapter 2 can be used for preemptive scheduling of rigid and malleable tasks, they cannot be used for moldable tasks. Since some approximation algorithms for non-preemtive problems are based on an LP-relaxation where the preemption is allowed, finding an LP-relaxation for preemptive moldable tasks can also be helpful for solving some non-preemptive problems.

Regarding scheduling non-preemptive tasks with minimum degree of parallelism, we presented complexity results for arbitrary number of processors. We also presented an algorithm to solve the problem when the number of processors is constant. It remains open if the problem is NP-hard when the number of processors is constant.

For solving $P|size_j, r_j|F_{max}$ we used a rounding method proposed by Bougeret *et al.* [14] for scheduling on heterogeneous platforms (SPP). It is possible that other rounding techniques used for solving SPP can be used to solve $P|size_j, r_j|F_{max}$ more efficiently.

Another open problem is whether it is possible for an online algorithm with constant speed augmentation to solve scheduling of malleable tasks with minimum degree of parallelism. We found a constant lower bound for speed augmentation and proposed an algorithm with $O(\log(n))$ speed augmentation. But it is still open if there is an online algorithm with constant speed augmentation or not.

Bibliography

[1] S. Albers. Better bounds for online scheduling. *SIAM Journal on Computing*, 29(2):459–473, 1999.

[2] A. Amoura, E. Bampis, C. Kenyon, and Y. Manoussakis. Scheduling independent multiprocessor tasks. In *AlgorithmsESA'97*, pages 1–12. Springer, 1997.

[3] S. Anand, N. Garg, and A. Kumar. Resource augmentation for weighted flow-time explained by dual fitting. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1228–1241. SIAM, 2012.

[4] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 623–631. ACM, 1993.

[5] Y. Azar, J.S. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 203–210. Society for Industrial and Applied Mathematics, 1992.

[6] K.P.B.P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. *Urbana*, 51:61801, 1990.

[7] N. Bansal and K. Pruhs. Server scheduling in the weighted p norm. *LATIN 2004: Theoretical informatics*, pages 434–443, 2004.

[8] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 51–58. ACM, 1992.

[9] Y. Bartal, H. Karloff, and Y. Rabani. A better lower bound for on-line scheduling. *Information Processing Letters*, 50(3):113–116, 1994.

[10] M.A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 270–279. Society for Industrial and Applied Mathematics, 1998.

[11] J. Blazewicz, M. Drabowski, and J. Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *Computers, IEEE Transactions on*, 100(5):389–393, 1986.

[12] J. Blazewicz, M. Drozdowski, D. De Werra, and J. Weglarz. Deadline scheduling of multiprocessor tasks. *Discrete applied mathematics*, 65(1):81–95, 1996.

[13] A. Blum, S. Chawla, and A. Kalai. Static optimality and dynamic search-optimality in lists and trees. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1–8. Society for Industrial and Applied Mathematics, 2002.

[14] M. Bougeret, P.F. Dutot, K. Jansen, C. Robenek, and D. Trystram. Scheduling jobs on heterogeneous platforms. In *Computing and Combinatorics*, pages 271–283. Springer, 2011.

[15] C. Chekuri and B. Moseley. Online scheduling to minimize the maximum delay factor. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1116–1125. Society for Industrial and Applied Mathematics, 2009.

[16] B. Chen and A. Vestjens. Scheduling on identical machines: How good is lpt in an on-line setting? *Operations Research Letters*, 21(4):165–169, 1997.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] M.L. Dertouzos and A.K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *Software Engineering, IEEE Transactions on*, 15(12):1497–1506, 1989.

[19] M. Drozdowski. Real-time scheduling of linear speedup parallel tasks. *Information processing letters*, 57(1):35–40, 1996.

[20] M. Drozdowski. Scheduling multiprocessor tasksan overview. *European Journal of Operational Research*, 94(2):215–230, 1996.

[21] M. Drozdowski and W. Kubiak. Scheduling parallel tasks withsequential heads and tails. *Annals of Operations Research*, 90:221–246, 1999.

[22] J. Du and J.Y.T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.

[23] U. Faigle, W. Kern, and G. Turán. *On the performance of on-line algorithms for partition problems*. University of Twente, 1989.

[24] R. Fleischer and M. Wahl. Online scheduling revisited. *Algorithms-ESA 2000*, pages 202–210, 2000.

[25] G. Galambos and G.J. Woeginger. An on-line scheduling heuristic with better worst-case ratio than graham's list scheduling. *SIAM Journal on Computing*, 22(2):349–355, 1993.

[26] M.R. Garey and D.S. Johnson. "strong"np-completeness results: Motivation, examples, and implications. *Journal of the ACM (JACM)*, 25(3):499–508, 1978.

[27] A. Goel, M.R. Henzinger, S. Plotkin, and E. Tardos. Scheduling data transfers in a network and the set scheduling problem. *Journal of Algorithms*, 48(2):314–332, 2003.

[28] M.X. Goemans. Improved approximation algorthims for scheduling with release dates. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 591–598. Society for Industrial and Applied Mathematics, 1997.

[29] T. Gormley, N. Reingold, E. Torng, and J. Westbrook. Generating adversaries for request-answer games. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 564–565. Society for Industrial and Applied Mathematics, 2000.

[30] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[31] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.R. Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey, annals discrete mathematics 5. 1979.

[32] M. Grotschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[33] K.S. Hong and J.Y.T. Leung. On-line scheduling of real-time tasks. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 244–250. IEEE, 1988.

[34] J. Hoogeveen and A. Vestjens. Optimal on-line algorithms for single-machine scheduling. *Integer Programming and Combinatorial Optimization*, pages 404–414, 1996.

[35] S. Im and B. Moseley. An online scalable algorithm for minimizing lk-norms of weighted flow time on unrelated machines. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 95–108. SIAM, 2011.

[36] K. Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39(1):59–81, 2004.

[37] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. *ALGORITHMICA-NEW YORK-*, 32(3):507–520, 2002.

[38] K. Jansen and L. Porkolab. Computing optimal preemptive schedules for parallel tasks: linear programming approaches. *Mathematical programming*, 95(3):617–630, 2003.

[39] M.T.C.S. JIS. Computers and intractability a guide to the theory of np completeness. 1979.

[40] B. Johannes. Scheduling parallel jobs to minimize the makespan. *Journal of Scheduling*, 9(5):433–452, 2006.

[41] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM (JACM)*, 47(4):617–643, 2000.

[42] D.R. Karger, S.J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 132–140. Society for Industrial and Applied Mathematics, 1994.

[43] R.M. Karp. Reducibility among combinatorial problems. complexity of computer computations,(re miller and jm thatcher, eds.), 85–103, 1972.

[44] J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G.R. Kan. *Preemptive scheduling of uniform machines subject to release dates*. Stichting Math. Centrum, 1982.

[45] C.Y. Lee and X. Cai. Scheduling one and two-processor tasks on two parallel processors. *IIE transactions*, 31(5):445–455, 1999.

[46] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 110–119. ACM, 1997.

[47] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J.E. Gehrke. Online scheduling to minimize average stretch. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 433–443. IEEE, 1999.

[48] C. Phillips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. *Mathematical Programming*, 82(1):199–223, 1998.

[49] K. Pruhs and E. Torng. Online scheduling. 2007.

[50] J.F. Rudin III and R. Chandrasekaran. Improved bounds for the online scheduling problem. *SIAM Journal on Computing*, 32(3):717–735, 2003.

[51] J. Savage. The complexity of computing, 1976.

[52] L. Stougie and A. Vestjens. Randomized algorithms for on-line scheduling problems: how low can't you go? *Operations Research Letters*, 30(2):89–96, 2002.

[53] V. Vazirani. Introduction to lp duality. In *Approximation Algorithms*, pages 93–107. springer, 2004.

[54] V.G. Vizing. About schedules observing deadlines. *Kibernetika,(1)*, pages 128–135, 1981.

[55] V.G. Vizing. Minimization of the maximum delay in servicing systems with interruption. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 22(3):717–722, 1982.

[56] J. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.L. Wu, and R. Vernica. Circumflex: a scheduling optimizer for mapreduce workloads with shared scans. *ACM SIGOPS Operating Systems Review*, 46(1):26–32, 2012.

[57] J. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.L. Wu, and R. Vernica. On the optimization of schedules for mapreduce workloads in the presence of shared scans. *The VLDB Journal*, pages 1–21, 2012.

[58] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.L. Wu, and A. Balmin. Flex: A slot allocation scheduling optimizer for mapreduce workloads. *Middleware 2010*, pages 1–20, 2010.