Departamento de Computación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Tesis de Licenciatura

On-line Multi-threaded Scheduling

por
Marcelo Mydlarz
L.U.: 290/93

Director
Dr. Esteban Feuerstein

Julio de 1998

Contents

Preface

This work was originally designed to be a framework for the On-line Multi-threaded Scheduling problem, which is an extension of the classic job scheduling problem. It was intended to comprise a taxonomy for the different scenarios of the problem -each one modeling a group of real life contexts.

Along the development of this work, this taxonomy has been subject to change, sometimes minor alterations, but other times big transformations, which reflected the increase in the acquired knowledge of the problem. As a result, relations among the different scenarios have come up.

In addition to this, we have proceeded our study in the scenarios of the Multi-threaded Scheduling problem tackling competitiveness. Both upper and lower bounds were conceived.

The main contributions of this work consist in:

- The definition of a framework for the on-line multi-threaded scheduling problem. The classification presented here, gives rise to many different scenarios, leaving a wide field for future research.
- The relations found among the different features of the scenarios. They enable us to analyze seemingly different scenarios as if they were the same one.
- The competitiveness results achieved for some important scenarios of our problem.

This work is divided in five chapters. The first one is introductory and presents some basic concepts about on-line algorithms and competitive analysis, scheduling, multi-threaded contexts, multi-threaded scheduling and on-line multi-threaded scheduling. This chapter ends with a context definition for on-line multi-threaded scheduling and general conventions that will be used through the following chapters.

In the second chapter the taxonomy is presented. The main feature that characterizes an on-line multi-threaded scheduling context is the *on-line paradigm*, where time plays an outstanding role. Other characteristics that shape the context of the problem will be presented in the same chapter, as the *objective functions*, the *number of machines*, the *precedence restrictions*, the *machines speed*, whether *preempting* jobs is allowed, the use of *priorities* for the jobs, the possibility to *reject* jobs, the *time window constraints* of the jobs, the *finiteness or infiniteness* of the length of the threads and whether there may be or not pairs of jobs *conflicting* with each other (i.e., that cannot be executed at the same time).

In the third chapter the relations among different scenarios are considered. First we tackle the relations among the precedence constraints, and after that the precedence constraints are related to the on-line paradigms. In the last section of this chapter, we evaluate the convenience of waiting before executing a job while leaving a machine idle.

Chapter 4 is devoted to competitive analysis of some of the scenarios defined in Chapter 2. Specifically, two objective functions (*Makespan* and *Total Completion Time*) and the three main paradigms defined in Chapter 2 are considered and some interesting results are achieved.

In the last chapter we give an overview of this work and recall the main paths that became open for future research. We also present there a table summarizing the competitiveness results achieved in Chapter 4.

Acknowledgements

First of all, I would like to thank my advisor, Esteban Feuerstein, who has introduced me in the wonderful land of research. It has been a great pleasure working with him and I truly appreciate his confidence and permanent support.

I would like to thank the external reviewers Irene Loiseau and Leen Stougie for their comments and corrections which have been very useful in the preparation of the final version. I would specially like to thank Leen for his affability. During the few days he was in Argentina, I enjoyed discussing about different research topics with him and Esteban.

I owe a debt of gratitude to Alejandro Strejilevich de Loma for his corrections and comments about the thesis and for his permanent help in the use of LaTeX.

During all these years at the University of Buenos Aires, I have relied on a small group of wonderful friends. I would specially like to thank Ro for being by my side at each moment I needed her.

Finally, I reserve these last lines to my parents. I really appreciate their patience and understanding, and I owe them most of what I am. My deepest feeling of gratitude is for them.

Marcelo Mydlarz.

Chapter 1

Introduction

1.1 On-line algorithms and Competitiveness

The past years have seen an explosive growth in studies of on-line problems. Typically when we solve problems and design algorithms we assume that the entire input is available beforehand. However, this may not be the case as in the Investment Problem, the Ski Rental Problem, Paging, List Processing [14, 19] and many other situations presented in the literature and largely motivated by many real-life problems.

On-line algorithms attempt to model a real life situation, where complete data is not known *a priori* (when the execution of the algorithm begins). In such cases, the input is obtained incrementally, and the algorithm has to make decisions to respond to the input, considering that it will not be allowed to change the past actions -decided on-line- later. Thus, as decisions are made irrevocably, the best action taken now may prove to be not very good some time later.

The set of actions taken has an associated cost. This cost can be reflected by different *cost functions.*

Posing that each part of the solution is obtained without knowledge of future requests, the goal is to find a solution as "good" as possible.

Generally, the on-line algorithm is compared to an optimal off-line algorithm that knows the entire input in advance. The optimal off-line algorithm is defined, for each entire input, as an algorithm that minimizes the cost of processing that input (w.r.t. a given cost function). In this way, it is as if that off-line algorithm *knew the future.* Intuitively, since on-line algorithms lack future information, it is clear that they may perform much worse than their off-line counterparts.

In contrast it is not straightforward to define a measure of performance for on-line algorithms, since no matter what action an on-line algorithm takes in response to an initial set of requests, there may probably be a sequence of further requests that makes that first action be counterproductive w.r.t. a given objective (function).

The most widely accepted way of measuring the performance of on-line algorithms is *competitive analysis.* This field began to achieve a steady development since Sleator and Tarjan formally introduced it in [29]. As explained above, competitive analysis addresses the comparison between the performances of the on-line algorithm and the optimal off-line algorithm. The *competitiveness* of an on-line algorithm is the ratio of its performance to the performance of an optimal off-line algorithm.

We will formalize these notions using the notation of [22]. Let $C_A(\sigma)$ denote the cost incurred by an algorithm A in satisfying a request sequence σ. Algorithm B is c-competitive (has competitive ratio c) if there is a constant a such that, for every request sequence σ,

$$\text{for every algorithm } A, \quad C_B(\sigma) \leq c.C_A(\sigma) + a.$$

Equivalently, if *opt* is the optimal off-line algorithm,

$$C_B(\sigma) \leq c.C_{opt}(\sigma) + a.$$

An algorithm is *competitive* if it is c-competitive for some constant c. This constant c is called the *competitive factor.* An algorithm is *strongly competitive* if it achieves the smallest possible competitive factor.

It is a usual practice to treat on-line algorithms as a game between an on-line player and an *adversary.* Given the on-line algorithm, this adversary picks an input sequence to the problem so as to make the on-line algorithm look as foolish as possible. Formally speaking, the adversary tries to maximize the ratio between the costs of the on-line algorithm and optimal off-line algorithm.

Presenting the problem in this way enables us to obtain both lower and upper bounds. Each lower bound l indicates that no on-line algorithm (for the same problem) can have a better competitive factor than l, i.e., no on-line algorithm is better than l-competitive. Generally, to prove this lower bound, a set of input instances is provided, such that for every on-line algorithm, the ratio between this algorithm and the optimal off-line algorithm is at least l.

On the other hand, each upper bound u is generally achieved by providing an on-line algorithm that is u-competitive.

For an overview of the literature on on-line optimization we refer to [10, 9].

1.1.1 Adversaries fighting against Randomization

Randomization has gained attention considering that, to construct input sequences, adversaries predict the deterministic on-line algorithms' behavior and try to force them to perform as bad as possible with respect to their off-line counterparts. The intention is to weaken the adversary's capacity to predict.

The main characteristic of randomized algorithms is that they toss coins in the course of their execution. This fact complicates the adversary's objective, who may put stones in a path that the randomized on-line algorithm will not necessarily follow.

As the behavior of a randomized on-line algorithm B may vary even for the same input, then B is defined to be c-competitive if there is a constant a such that, for every request sequence σ,

$$\text{for every adversary } A, \quad E(C_B(\sigma) - c.C_A(\sigma)) \leq a.$$

In contrast with the deterministic case, for the randomized case different types of adversaries have been defined. For all of them, it is assumed that the adversary knows the code of the randomized on-line algorithm, but does not know the results of the random coin tosses. Here we present three different types of adversaries:

- **Oblivious Adversary**. This adversary is called *oblivious* reflecting the fact that he is oblivious to the random choices made by the on-line algorithm.

Thus, the adversary generates a sequence of requests without knowing any result of the coin tosses made by the randomized on-line algorithm.

- **Adaptive On-line Adversary**. The adversary chooses each request after having observed the *previous* choices of the on-line algorithm. For this version, the adversary decides what action to follow without knowing the future requests.
- **Adaptive Off-line Adversary**. As in the previous case, the adversary chooses each request after having observed the *previous* choices of the on-line algorithm. In contrast with the Adaptive On-line Adversary, the Adaptive Off-line Adversary decides what action to take at each step knowing the whole sequence of requests and the corresponding on-line algorithm behavior.

Summing up, the advantage of the *adaptive* adversaries over the *oblivious* adversary is that the first one decides what request to generate after knowing the past behavior of the on-line algorithm. In addition, the advantage of the *adaptive off-line* adversary over the *adaptive on-line* is that the first one knows all the on-line algorithm's actions before selecting any of its own.

In case these adversaries are opposed to a deterministic on-line algorithm, they are the same: there is no advantage to observing the on-line algorithm's behavior since the adversary can exactly predict its behavior.

If we denote C^{obl}, C^{aon}, C^{aof} and C^{det} the best possible competitive ratios considering oblivious adversaries, adaptive on-line adversaries, adaptive off-line adversaries and the deterministic case respectively, we have that

$$C^{obl} \leq C^{aon} \leq C^{aof} \leq C^{det}.$$

In [8] the following two theorems are proven.

Theorem 1.1.1.1 *If there is a c-competitive randomized algorithm against adaptive off-line adversaries, then there is a c-competitive deterministic algorithm; in other words, there is no advantage to randomizing when playing against adaptive off-line adversaries.*

Theorem 1.1.1.2 *If there is a c-competitive randomized algorithm against oblivious adversaries and a d-competitive randomized algorithm against adaptive on-line adversaries, then there is a cd-competitive deterministic algorithm.*

We will not cover randomized algorithms any further. References of this topic are [19, 25, 23].

1.2 Multi-threaded Scheduling

Scheduling theory is concerned with the optimal allocation of scarce resources to activities over time. “The practice of this field dates to the first time two humans contended for a shared resource and developed a plan to share it without bloodshed” [18]. Inspired by applications in practical computer systems, it developed into a theoretical area with many interesting results.

We can sketch the following basic situation. Given a sequence of jobs, our task is to process them on the available machines. Each job, characterized by its processing time (possibly among other features), has to be scheduled (for execution) on one of the machines. Other variants that may change the set of *feasible* schedules are also possible.

The goal of achieving a schedule as efficient as possible, may have different meanings, according to the problem. In other words, many different objective functions are possible. *Minimizing the total length of the schedule (Makespan)* as well as *minimizing the sum of the time each job takes to be completed (Total Completion Time)* are the most common.

Scheduling problems have also been treated from an on-line viewpoint: the input is not complete when the execution of the algorithm begins. Instead, both the jobs to be processed and the characteristics of each of them are made available during the decision process.

Among others, the following constraint must also hold: time slots of each pair of jobs executed in the same machine must not overlap each other. An up-to-date survey on on-line scheduling problems is given in [28].

As we will be concerned with competitive analysis, we will think of the time that the on-line algorithm takes to calculate the assignment of the jobs as negligible with respect to the execution time.

Later, along the description of the different on-line paradigms that can be considered, we will refer to the central role that *time* plays, which has been previously enphasized in [11].

Here, we are concerned with the threaded variant of this problem. In a multi-threaded problem, there is a certain number of threads and each one consists of an ordered sequence of requests. Only the first request of each (non-empty) thread is available to be served. An algorithm for this problem must decide which of these requests to serve at any moment. In the on-line version, only the first request of each (non-empty) thread is seen.

Multi-threaded on-line optimization problems have been proposed first in [10, 12] for the paging model. The situation is the following. There is a given number of independent processes that simultaneously present requirements of pages of secondary memory that must be brought into fast memory. At each moment the algorithm that manages fast memory can see only one request per process, precisely the first unserved request of the sequence of requests that the process presents. The process does not see a request until the previous one in the sequence has been served. The algorithm must decide at each moment, not only which request(s) to serve, but also which page(s) of fast memory to evict to make place for the incoming page(s).

Multi-threaded Scheduling scenario can be considered as a generalization of the scheduling problem, which in many cases may be seen as a single-threaded scheduling problem. In scheduling problems with multiple threads, the jobs are distributed in more than one thread. Jobs are scheduled one at a time and each job is only *seen* when the previous job in the same thread has been scheduled. The threaded nature of the problem implies that a job cannot be scheduled before the previous jobs in the same thread have been scheduled, both for the on-line and the off-line algorithms.

The on-line multi-threaded context implies that, apart from the normal decisions in single-threaded on-line optimization, the order in which the threads are explored must also be decided. At any given moment, at most one job of each thread (the one on top) is available to be scheduled (and, depending on the on-line paradigm, executed). More than that, the on-line algorithm can only see those jobs (apart from the jobs already processed).

At the time a given job j in a thread becomes available to be scheduled, the following two facts happen. First, all the jobs preceding j (in the same thread) have been scheduled. Second, all the jobs following j (in the same thread) are not ready to be scheduled.

In addition to this, as we will see in Chapter 3, the multi-threaded environment gives rise to new precedence relation types.

References on multi-threaded on-line optimization problems are [10, 11, 12, 26, 27, 2].

1.3 Context Definition and Conventions

The context where we will be working is the following. There will be a certain number of threads -always denoted by w- and a certain number of machines -always denoted by m- where the jobs are to be processed. The terms *running time* and *processing time* will be used indistinctively. Each thread will be modeled as an ordered sequence of jobs. Each job will be characterized by its running time. The results presented from now on apply to arbitrary real running times. Jobs will be referenced as integer numbers.

The scheduling algorithm is designed to produce a schedule, which means that each job is assigned to one or more machines and one or more time slots, according to the variant of the scheduling problem. Each machine is assigned to a single job at any time, and the processing of a job always takes at least its *running time*. We say that a job has been scheduled when the decision of the machine and time slots in which that job will be executed has been made. Given two jobs A and B in a given thread, if A precedes B in the thread then A must be scheduled before B.

A *(legal) schedule* is an assignment of the jobs of the threads where

- Each job is assigned to one of the machines and for a given time period (in some variants, a job may be assigned to more than one time period).
- Any pair of jobs from the same thread must be assigned in the order of precedence of the thread (i.e., in the order they are presented). This fact does not imply that a given pair of jobs belonging to the same thread must be executed in the order of precedence of the thread, not even if they are

scheduled to the same machine. In other words, a job A preceding another job B may be scheduled for execution at a later time period than B's time period.

- There is no pair of jobs such that both are to be executed by the same machine at the same time, not even for some fraction of their running time. There is no intersection between the time periods of two different jobs executed in the same machine. In other words, their running times may not overlap.

A schedule produced from a given set of threads is said to be *feasible* under a given paradigm if it can be obtained under that paradigm from the same set of threads and satisfies any other constraint given by the problem.

All these conventions apply both for off-line and on-line algorithms. In the on-line case, the first job of each thread has to be scheduled to some machine(s) and time slot(s) before the next jobs in the thread are seen, consistently with other restrictions given by the problem.

Chapter 2

Classification

This classification has been designed with a view to gaining some insight into the relations that arise between the different contexts where Single and Multi-threaded Scheduling make sense. These contexts are defined by many different parameters.

As it will be shown later, a slight variation in the setting of only one of these parameters may imply a significant change in the obtained results for the problem. For example, we have found that changing the number of machines from 1 to 2 in a given context brings about a complete turn over in the achieved competitiveness result: not only the competitive algorithm "discovered" for the single machine problem does not work for the two machines problem, but also we have proven that in this context (2 machines) no algorithm is competitive.

Both single and multi-threaded scheduling models are defined by many features. These include the *on-line paradigm*, the *objective function*, the *number of machines*, the *precedence restrictions*, the *machine speed*, the use of *preemption*, the use of *priorities*, the use of *rejection*, the *time window constraints*, the *finiteness* of the number of jobs and the *conflict restrictions*.

Among these characteristics, special attention is given to the on-line paradigms and the precedence constraints. We have defined three main on-line paradigms that differ from previously related work [28]. We have found a close relation between the on-line paradigms and the precedence constraints; this topic is covered in the next chapter.

The first main division of the scenarios we will focus on regards **Data Availability**. Any algorithm may be classified as

- Off-line
- On-line.

Our view will be directed to on-line algorithms. Its performance is measured by the competitive ratio (w.r.t. some objective function). Thus, we will also be interested in off-line algorithms, not from their algorithmic viewpoint, but for the optimal schedules they produce, so as to compare them with the worst schedules obtained on-line.

By the time the different ways of classifying the algorithms are presented, it will appear that although most of them may be applied to both off-line and on-line algorithms, a few refer only to on-line algorithms. For example, the number of processors may be unknown for an on-line algorithm, but not for an off-line algorithm.

It will also appear that not all the combinations of the possible values of the parameters of our problem are equally important; furthermore, some of them are trivial and others make little or no sense at all.

In Sgall's survey [28], on-line scheduling has been treated, bringing on a broad classification. We have taken the ideas emerging from that survey into account and extended the classification for the multi-threaded case.

The present work, born with the conceptual view of Sgall, have suffered several transformations, specially in what concerns the on-line paradigms and the precedence constraints.

The first main topic we will be concerned in is On-line Paradigms. Its importance lies in that it defines a big part of the scenario in which the problem takes place.

2.1 Different On-line Paradigms

Many different issues shape an on-line paradigm. Among them, there are two that stand out.

- The relation between the scheduling time and the execution time.
- Whether scheduling each job to the first available space (time slot) is mandatory (or not).

Both of them give evidence of the central role that time plays. This fact is a natural consequence of the main objectives, which are functions of time.

If *scheduling each job to the first available space is* **mandatory**, then -as one job is not scheduled before the preceding job in the thread had been scheduled- some kind of execution precedence of the jobs is ensured. For example, if two jobs arriving from the same thread are scheduled to the same machine, they must be executed in the order of appearance.

On the other hand, if *scheduling each job to the first available space is* **optional**, then jobs may be executed at any time (avoiding the overlap of two jobs in a single processor) and in any order.

The **scheduling time may be independent of the execution time** or **scheduling may be done immediately before executing** each job. In the first case, time *does not flow during the scheduling*, but it flows during the execution. Thus two phases can be distinguished: the scheduling phase, in which a schedule is obtained, and the execution phase, in which execution is done according to that schedule, obtained some time before. As the scheduling must finish before the execution starts, all the information about the jobs must have arrived during the scheduling and before the execution begins. Nevertheless, assignments' decision are made with incomplete information and irrevocably. Naturally, we will only be concerned with the scheduling phase. This case may be seen as a *transformation* from the w initial threads into m sequences, one for each machine. In this case, *scheduling each job to the first available space* may be either mandatory or optional.

In the case where each job is executed immediately after being scheduled, a job may be scheduled only when the machine receiving that job is free (idle). Here, time *flows during the scheduling* as the scheduling time and the execution time coincide. As a consequence, information about the jobs (or the jobs themselves) may arrive during the execution. Also, each job must be scheduled to an idle machine.

Here we present a brief overview of the paradigms that arise when these issues are considered. Later, we will study these paradigms in deep.

- *Scheduling Jobs One By One*: Under this paradigm, the running time of each job is known when the jobs becomes available. Jobs can be scheduled for execution at arbitrary time slots (i.e., they can be delayed). This implies that a job can start running later than the successive jobs in the thread; however, once we see the successive jobs we cannot change the assignment of the previous jobs.

  In this paradigm, *scheduling each job to the first available space is optional* and *the scheduling time is independent of the execution time.*

- *Scheduling Before Executing*: In this paradigm each job must be scheduled to the first available time slot of any machine. There is no need to wait to schedule a job if the previous jobs in the same thread have been scheduled. Jobs may be scheduled one after another without delay.

  In this paradigm, *scheduling each job to the first available space is mandatory* and *the scheduling time is independent of the execution time.*

- *Scheduling On Execution*: In this paradigm, each available job has to be scheduled for *immediate* execution, i.e., each job is scheduled to be executed *now.*

  In this paradigm, *scheduling each job to the first available space is mandatory* and *the scheduling time and the execution time coincide.*

  Two variants can be considered:

  - **with known running time**: As soon as a job is ready to be executed, its running time becomes available.
  - **without known running time**: The running time of each job is not known until it has finished.

In the following paragraphs, these three paradigms are covered with greater insight; in addition, a few paragraphs are devoted to an overview of paradigms where the notions of competitiveness is slightly altered.

2.1.1 Scheduling Jobs One By One

This paradigm was introduced in [28]. As soon as the job is presented (i.e., becomes the first one of the thread) we know (or we are able to calculate) its running time. We are allowed to assign the jobs to arbitrary time slots (i.e., they can be delayed), thus a job can start running later than the successive jobs in the thread; however, once we see the successive jobs we cannot change the assignment of the previous jobs.

Clearly, no precedence restriction is implied regarding the order of execution.

In this paradigm, the order in which jobs are presented plays no role in the scheduling (time plays no role) -except for the on-line algorithm knowledge of the previously scheduled jobs. This implies that *scheduling each job to the first available space is optional.*

It is clear, that there is no need to wait to schedule a job if the previous jobs in the same thread have been scheduled. Thus, *the scheduling time is independent of the execution time.*

Under this paradigm, the number of threads is unimportant for any off-line algorithm. An optimal schedule obtained off-line from a set of jobs is independent of the number of threads in which the jobs are distributed. In other words, if there are two instances of this problem, both of them composed of the same set of jobs and distributed within a different number of threads, then a schedule is optimal (off-line) for one of them if and only if that schedule is optimal for the other.

Considering the on-line point of view, the more the number of threads, the more the number of jobs an on-line algorithm can see at any time. This fact which seems to be an advantage, is not necessarily so. The *adversary* can use the first job of each thread to hide jobs so as to make the on-line algorithm perform bad with respect to its off-line counterpart. Besides a lower bound for the competitive ratio achieved with a given number of threads is also a lower bound for the competitive ratio considering a greater number of threads because some threads may be empty.

An interesting variant of this paradigm that we will not cover is scheduling each job for execution in a given order by a given machine, instead of assigning it to a given time slot. This variant may be useful if we know the number of jobs to process (or process the jobs by shifts, each one a constant number of jobs) and makes useless to wait (i.e. to use idle times).

2.1.2 Scheduling Before Executing

In this paradigm each job must be scheduled to the first available time slot of any machine.

There is no need to wait to schedule a job if the previous jobs in the same thread have been scheduled. We will see that this fact makes the difference with the following paradigm (*Scheduling On Execution*). *The scheduling time is independent of the execution time.* As one job can be scheduled after another (in the same machine) without delay, we can say that time does not flow. In other words, the flow of time occurs during the execution, not during the scheduling.

In this paradigm, even time does not flow, it impacts the execution: if two jobs are scheduled to the same machine, the one that has been scheduled before will be executed before. This implies that two jobs coming from the same thread and scheduled to the same machine will be executed in the order of appearance. This is the main difference with the previous paradigm. *Scheduling each job to the first available space is mandatory.*

Scheduling Before Executing paradigm can be seen as a transformation from w queues into m rows: *only* the *first* job of any queue can be picked and scheduled *only* to the *first* available space of some row. These restrictions were not implied by the previous paradigm.

As it will be shown later, this paradigm guarantees the *Extended Partial Order* constraint.

2.1.3 Scheduling On Execution

In this paradigm, each available job has to be scheduled for **immediate** execution, i.e., each job is scheduled to be executed **now**. At any time, there is at most one job of each thread ready to be executed.

Unlike the previous paradigm, the scheduling time is the same as the execution time. So, a job is ready to be scheduled if and only if a job is ready to be executed. Here, a job can only be scheduled to an idle machine. The role that time plays in this paradigm is critical.

A job may have to wait before being assigned for execution to a given machine, because that machine may be executing another job. The waiting job, can also delay the jobs that come behind it in the thread. Then, we can say that time flows. This is the main difference with the previous paradigm.

In this paradigm, *scheduling each job to the first available space is mandatory* and *scheduling is done immediately before executing each job*. For each job, the scheduling time coincides with the beginning of the execution.

This paradigm also implies that two jobs coming from the same thread and scheduled to the same machine will be executed in the order of appearance.

We can consider variants of this problem:

- **with knwon running time**: As soon as a job is ready to be executed, its running time becomes available.

  As it will be shown later, this paradigm implies the *Start-Start* constraint.

- **without known running time**: The running time of each job is not known until it has finished. The on-line algorithm only knows whether a job is still running or not. This paradigm will not be covered in this work.

  In [28] a similar paradigm has been treated; it differs in that all the jobs are available to be executed. Thus, it is different even for the single-threaded case -where only the first job (not scheduled yet) is available. A summary of results and references are given in [28].

Sometimes the algorithms that know the running time of a job as soon as it arrives are called clairvoyant, in contrast to non-clairvoyant algorithms that correspond to the previous paradigm of unknown running times [28].

2.1.4 Relaxed Notions of Competitiveness

To face the negative results that appear in many variants of the scheduling problem, it has been allowed to use additional information or slightly more resources to the on-line algorithm.

The following cases are summarized in [28].

- **Using machines with speed** $1 + \epsilon$. Results for this variant have been obtained in [17, 24].
- **Giving aditional information to the on-line algorithm**. For example, jobs ordered by non-increasing running times [16], knowledge of the total running times of all the jobs or the existence of one buffer to temporarily store one job or the possibility to produce two solutions and choose the best of them afterwards [20] and knowledge of the optimum [4].

These issues go beyond the scope of this work.

2.2 Objective Functions

There are several useful objective functions that measure different aspects of the schedule. Many of them have been summarized in [28] for the conventional on-line scheduling problem, and some of them may be extended for the multi-threaded case in a natural way.

The most common of these functions is the **Makespan** and the objective in this case is to complete the processing of all the jobs as quickly as possible. More formally, the makespan function measures the time at which the last job among all the threads is completed. The goal is to minimize this function. While we must care about the machine where each job is scheduled, the order in which the jobs will be executed on the machines is unimportant.

Makespan may reflect the viewpoint of the owner of the machines: the lower the makespan is, the higher utilization the machines have. In case rejection of jobs is allowed, each rejected job is punished with a penalty that must also be minimized. Rejection may be useful when a job has long running time and small benefit (i.e., there is a small penalty for not scheduling it).

In the user of the machines point of view, specially in interactive environments, the time it takes to complete each individual job may be more important. Other objective functions are considered in this situation, like **total completion time** (also called *Latency*), **total flow time** (also called *total response time*) and **total waiting time**.

The completion time of a job is the time it takes the job to be completed. Thus, *the makespan equals the maximum completion time.* The *total completion time* is the sum of the completion times of the jobs. Unlike in the makespan problem, the order in which jobs are scheduled on the machines influences the *total completion time* cost function.

The completion time of a scheduled job j is denoted by C_j. The makespan of the jobs is denoted by $C_{max} = \max_j C_j$ and the total completion time by $\sum_j C_j$.

In the conventional scheduling environment, the flow time of a job is defined as the time the job is in the system, i.e., the completion time minus the time when it becomes first available. The waiting time is the flow time minus the running time of a job. In a multi-threaded environment a job may be considered *available* when its preceding job has been scheduled, has started its execution or has ended its execution, depending on the on-line paradigm. Flow time and waiting time do not seem to be very useful in a multi-threaded context.

If preemption is allowed, minimizing the number of preempted jobs may be also a goal, although not as important as the others.

Along this work, we will only deal with Makespan and Latency cost functions.

2.3 Number of Machines

Considering the number of machines is important when calculating the competitive ratio of an on-line algorithm.

- Fixed.

The number of machines is fixed and known before the scheduling begins. The following cases define borders, where the change from one case to another may mean the change from a problem for which there exists a competitive algorithm to a problem for which no competitive algorithm exists.

  - $m = 1$. There are cases where on-line algorithms are competitive if only one machine is available to execute the jobs.
  - $2 \leq m \leq w$. When there are more than one machine, the number of threads as well as the number of machines may influence the competitive ratio.
  - $2 \leq w \leq m$.

- Variable.

  The number of machines may vary from time to time. A possible scenario where the number of machines may be treated as variable is a multiprogrammed environment.

  Initially, only one computation (composed of w threads) may be running, so it may be using all the m available machines. A moment later, another computation may be launched, taking control of at least one machine, and leaving the old computation with fewer than m machines to go on its processing. Now, the old computation faces the fact that the number of machines available for it have decreased. More than that, we can think that the adversary not only decides how much and the characteristics of the jobs that will appear, but also the number of machines available at any given moment. This topic has been thoroughly analyzed in [3].

2.4 Precedence Restrictions

In many applications, there are precedence restrictions that any feasible schedule must satisfy. For example, "one job cannot start its execution before the job that precedes it in the same thread has started". We will browse through some constraints that make sense in real life contexts.

In general, to get a significative competitive ratio, we must compare the on-line algorithm to the off-line algorithm under the same constraint. We will follow that policy when dealing with competitive analysis.

As it will be shown, outputs feasible under some constraints are not feasible under other constraints. We will now define the precedence constraints we have found more useful.

1. *No Precedence Restrictions.*
2. Start-End *On Each Machine*

   For every pair of jobs A, B belonging to a given thread scheduled to be executed (and executed) on the same machine, A precedes B in the thread if and only if A finishes before B starts (or equivalently, A is executed before B).

   Start-End *On Each Machine* constraint may also be called **partial order**. The reason is that each schedule produce by this constraint can be seen as a transformations from the w initial threads into m sequences, one for each machine. Each of these sequences preserves the *partial order* induced by every thread.
3. *Extended Partial Order*

   The *Extended Partial Order* constraint says that there is no sequence of jobs $A_1, \ldots, A_k$ such that

   - $A_i \rightarrow A_{i+1} \forall i \in \{1, \ldots, k-1\}$
   - $A_k \rightarrow A_1$,

   where $X \rightarrow Y$ means that either *job X precedes job Y in the same thread* or *job X is scheduled to be executed before job Y in the same machine.*

   This constraint is not as natural as the others and it is not easy to see its usefulness. Nevertheless, it comes to fill in a gap that exists between the Start-End *On Each Machine* constraint (partial order) and the restrictions that are guaranteed under the on-line paradigm *Scheduling Before Executing.* It is useful then to see an example that shows up this fact.

   > A possible schedule for the threads A: A_1, A_2 and B: B_1, B_2 is the following:
   >
   > machine 1: B_2, A_1
   >
   > machine 2: A_2, B_1
   >
   > The problem with this schedule is that it cannot be obtained after scheduling the first job of any thread, which is the idea behind the

> *Scheduling On Execution* paradigm. Nevertheless, it satisfies the Start-End *On Each Machine* constraint.

This restriction models the *Scheduling Before Executing* paradigm.

As it will be proven later, a schedule that satisfies this constraint also satisfies Start-End *On Each Machine* constraint.

4. *Start-Start*:

   For every pair of jobs A, B belonging to a given thread, A preceding B in this thread, B cannot start before A has started.

   This restriction models the *Scheduling On Execution* paradigm.

   As it will be proven later, a schedule that satisfies this constraint also satisfies *Extended Partial Order* constraint.

5. *Start-End*:

   For every pair of jobs A, B belonging to a given thread, A preceding B in this thread, B cannot start before A has finished.

   As it will be proven later, a schedule that satisfies this constraint also satisfies *Start-Start* constraint.

6. *End-End*:

   For every pair of jobs A, B belonging to a given thread, A preceding B in this thread, B cannot finish before A has finished.

   An instance of this model may be the following:

   > Suppose we have a set of machines in a network and our goal is to process one or more sequence of packets; each packet contains a compressed executable file and a number indicating an order of execution. The file in each packet must be uncompressed and delivered to a host computer for execution. Notice that the processing of each packet ends with the delivery of the file contained in the packet.
   >
   > The following condition must also be satisfied. No packet may be delivered if any file associated to a lower execution number packet has not been sent. However, each packet may be unshrinked at any time before being delivered.
   >
   > Here, the *End-End* restriction is derived from the fact that job $i+1$ cannot end (i.e., packet $i+1$ cannot be delivered) before job i has ended.

In the next chapter, we will state and prove theorems about the relations among these constraints. We will point out now a few considerations about them.

- For the *Extended Partial Order* constraint, the Start-End *On Each Machine* constraint and the absence of constraints there is always a schedule that can be viewed as a transformation of the w threads into m sequences -one for each machine- where there are no idle times between any pair of jobs. This property can not be guaranteed for the other constraints defined above.

- We will present an example showing that a schedule that is optimal satisfying a given restriction may not be feasible if another constraint is imposed. Furthermore, there may be no feasible schedule satisfying this last restriction that has as good cost as the schedule that satisfies the first constraint.

  Consider the following threads (identifying the jobs with their running times).

  – A: 9, 7
  – B: 1, 3, 2

  An optimal schedule for Makespan cost function satisfying the *Start-Start* constraint (job_{i+1} may only begin if job_i has started) is:

  Machine p: 9, 2 Machine q: 7, 1, 3

  which has a cost of 11.

  This schedule violates *Start-End* constraint (job_{i+1} may only begin if job_i has finished), for which the optimal schedule is

  Machine p: 9, 7 Machine q: 1, 3, 2

  and has a cost of 16, that is much worse than the cost obtained for the other case.

- Other restrictions not covered by the preceding definitions can be modeled by a directed acyclic graph; each directed edge means that the job represented by the first of the nodes adjacent to the edge must be scheduled or executed before the other job represented by the second node. We will not cover this situation.

2.5 Machine Speed

The speeds of the machines is another parameter of the problem. The following models have appeared in [28].

- Equal Machines:

  Each machine has the same speed. Then, any given job has the same running time on each machine.

- Uniformly Related Machines:

  The i-th machine has speed $s_i > 0$; a job with running time t executed on the i-th machine, takes time t/s_i.

  Equal Machines can be view as a special case of this one.

- Unrelated Machines:

  In this case, we know for each job the relative speed of the machines. That is, for each pair of one job and one machine, we have a possibly different speed. As we only know the speeds, we cannot predict the running time of a job until we know where it will be executed.

  Uniformly Related Machines can be view as a special case of this one.

- Restricted Assignment:

  Here, each job can be executed only on a given subset of the machines;

  This case can be view as a special case of Unrelated Machines, where forbidden machines for a given job have speed 0 for that job.

We will only cover the *Equal Machines*' case. The other cases are summarized and referenced in [28].

2.6 Preemption

Frequently we deal with situations where the processing of a job may be suspended and resumed (or restarted) later. The strategy of allowing processes that are logically runnable to be temporarily suspended is called *preemptive scheduling*, and it contrasts with *run to completion* (also called nonpreemtive scheduling) method.

- Without Preemption:

  Every job that is assigned to a machine, runs to completion.

- With Preemption:

  The on-line case gives us two variants to consider.

  - Every suspended job must be restarted later.
  - Every job that gains control of any machine goes on with its execution at the same point it stopped the last time it was running.

    Under the *Scheduling Jobs One By One* and *Scheduling Before Executing* paradigms, each job may be assigned to one or more machine and time slots, and (in contrast with *Scheduling On Execution* paradigm) this assignment has to be determined completely as soon as the job is presented.

In the following chapters we will assume that jobs are not allowed to be preemted. A summary and references about preemption for the classic on-line scheduling problem is provided in [28].

2.7 Priority

One of the characteristics of a job, apart from its running time may be a priority, that is an identification (usually a number) indicating how important the fast execution of the job is. The possibilities are the following.

- Without Priorities:

  Each job is as important as any other.

- With Priorities:

  Scheduling higher priority (and available) jobs first may be mandatory or it may just improve the cost without being mandatory. The last case may be modeled as a schedule with weighted jobs and a suitable cost function.

This item is closely related to **Preemption**. In case of working under a model *with preemption*, a job may be preempted if another job with higher priority becomes available.

Every analysis we will make from now on will be on jobs without priorities.

2.8 Rejection

In some given contexts it may be possible to decide whether a job will be served or not. In case there exists the possibility to *reject* jobs, a certain penalty must be paid for this behavior -or similarly, if each processed job has an associated benefit, this benefit is missed.

- **Without Rejection**:

  In this case, rejection of jobs is not allowed. This may happen when the jobs must necessarily be executed, or when the cost associated to a rejection clearly surpasses any possible intend to compensate for it.

- **With Rejection**:

  Jobs may be rejected at a certain price. Apart from its running time, each job is characterized by a *penalty* that the algorithm pays in case this job is rejected.

  The model presented in [7] used the *makespan* (of the scheduled jobs) *plus* the *sum of the penalties* (of the rejected jobs) as the objective function to minimize.

  As when deciding where to execute each job, in the on-line version of this problem the decision of either to reject or to schedule a job is made before knowing anything about the next job. Even though, in [7] the previously rejected jobs were allowed to be scheduled later.

  This situation of scheduling previously rejected jobs may happen, for example, when an algorithm that decides always to do the best *now*, i.e. to do at any time what minimizes the present value of the cost function. In such a situation, that algorithm may decide to reject a job -because scheduling it would increase the makespan too much- and sometime later to schedule it to some machine -as this action would imply a decrease in the cost function.

Thus, the scenario proposed in [7] may be seen as a non-trivial generalization of the well-known *sky-rental problem* [19].

Along this work, we will not be concerned with any schedule that allows rejection.

2.9 Time Window Constraints

There may be time window constraints that restrict the time at which each job may start and/or end its execution. The *release time* is the earliest time at which a job may start its execution. Let us note there may be a job that is not ready at its release time if there is a number big enough of jobs that precede it in the thread. For an on-line algorithm, if the release time of a job is greater than the time the jobs that precedes it in a thread have been scheduled, the release time is also the time at which this job is presented.

The *deadline* is the latest time at which a job may start (or end) its execution. Every job belonging to a feasible schedule finishes before its deadline. There may be jobs that cannot finish before its deadline for any feasible schedule, for example, if the number of jobs that precede it in the thread is big enough.

There may exist for each job, a release time, a deadline, both of them or none. As a result, the following cases may be studied.

- Interval Scheduling:

  Each job must be executed within a given interval of time. The following two cases may be considered.

  - The interval of each job starts when the first job among all the threads begins its execution.
  - The interval of each job starts when the job that precedes it in the same thread begins its execution.

- Only Release Time:

  Each job has a release time. This is a particular case of the previous one, where the deadline of each job is infinity.

- Only Deadline:

  Each job has a deadline. This is a particular case of Interval Scheduling, where the release time of each job is 0.

- No Time Window constraints:

  This is a particular case of Interval Scheduling, where the release time of each job is 0 and its deadline is infinity.

A summary and references about this topic for the classic on-line scheduling problem is provided in [28]. For the cases we will study no time window constraints will be assumed.

2.10 Finiteness

Two different approaches can be considered regarding the length of the threads. They deal with the *finiteness* of that length and are defined as follows.

- **Finite multiple threads**. In this case, algorithms are faced to a given number of threads, each one with a finite number of jobs. Each of these threads have to be served *completely* that is, the algorithm ends its task when the last job of the each thread has been processed.

- **Infinite multiple threads**. Under this model, each thread contains an *infinite* number of jobs and thus, an infinite scheduled is produced. In contrast with the *finite multiple threads* case, in this case we may have -according to one of the definitions- that at every point in the decision process an algorithm has to be competitive. In other words, the state of the system will be observed after a finite number of steps, comparing the costs incurred by different algorithms till that moment.

  Other definitions of an infinite model for the Multi-threaded Paging problem that can also be applied for the Multi-threaded Scheduling problem are defined in [13].

The conceptual difference between these models is that while in the first one every algorithm is forced to serve the same input (the same set of threads with the same jobs), although possibly in different orders, in the second model, at every step different algorithms may see different set of jobs. That is, in the *finite* model, there is a moment (at the end of the scheduling) when every given algorithm has the same view of the input, while in the *infinite* model this may not be the case.

References for this topic applied for the *Multi-threaded Paging* problem are [12, 13].

In [12] it is shown that for the paging problem, infinite multiple threads may yield significantly different competitiveness from the case with finite threads.

Along this work, we will only be concerned in the *finite* model.

2.11 Conflict Restriction

There may be a conflict within some pairs of jobs. In such a case, they must not be executing at the same time, although there is no need to execute one of them before the other. A case where this may happen is when two processes share a common memory space.

These are two possibilities:

- **No Conflict**: any pair of jobs may be executed at the same time (at different machines) if there are no precedence constraints that relate them.

- **With Conflicting jobs**: this case is explained above. An undirected graph is given, where each edge between two nodes indicates that the jobs associated to these nodes are in conflict. As a result, only an independent set of that graph is allowed to be executing at any time.

Conflicting jobs for the non-threaded scenario are treated in [28], where a few results and references are provided. We will not deal with conflicting jobs along this work.

2.12 Restricting the Model

We have defined three (3) main *on-line paradigms*, four (4) different *objective functions*, four (4) different cases regarding the *number of machines*, six (6) different *precedence constraints*, four (4) different cases regarding the *machines speed*, two (2) different cases regarding *preemption*, two (2) different cases regarding *priorities*, two (2) different cases regarding *rejection* of jobs, four (4) different cases regarding *time window constraints*, two (2) different cases regarding *finiteness* and two (2) different cases regarding *conflict restrictions.*

The combinations of these cases gives rise to $3{*}4{*}4{*}6{*}4{*}2{*}2{*}2{*}4{*}2{*}2 = 147456$ different scenarios.

Some of them are trivial or easy to analyze and others do not even make any sense. Nevertheless, it is obvious that *many books* would be needed to analyze all of them. For this reason, our model will be restricted for the rest of this work to the following framework.

We will be covering the three main paradigms without the assumption that any precedence constraint is present unless otherwise stated. We will assume that there are present a fixed number of machines. The objective functions we will work with are *Makespan* and *Total Completion time.* Only the *Equal machines* case will be covered, that is, all the machines will have the same speed. Jobs will *not be allowed to be preempted. No priorities* will be associated to the jobs. *Rejection* of jobs will *not* be permitted, i.e., all the jobs will have to be scheduled and executed. There will be *no time window constraints*, that is, the jobs will have neither release times nor deadlines. We will consider threads with *finite* number of jobs. Finally, there will be *no conflict* between any pair of jobs.

Chapter 3

Relations in the Classification

This chapter can be considered the core of this work because it comes to interconnect different aspects of the taxonomy as well as to state some facts about them.

We will focus on the relations that arise among different scenarios and the convenience of waiting before executing a job while leaving a machine idle.

The first section will be devoted to the relations among the precedence constraints. In the following section the precedence constraints will be related to the on-line paradigms. In the last section of this chapter, we will prove that in some contexts leaving some machine idle is useless.

3.1 Analyzing the Precedence Constraints

Intuitively, the presence of Start-End *On Each Machine* constraint does not avoid an anomaly related to some notion of cycles involving the jobs. This was what gave us the intuition to define graphs that model the *scheduling* and *execution* flow that arise in a given schedule.

We have found two of these graphs, one that is associated to the Start-End *On Each Machine* constraint and another one that is associated to the *Extended Partial Order* constraint.

First, we will define these graphs. Then we will provide two theorems that relate these graphs with Start-End *On Each Machine* and *Extended Partial Order* constraints. After that, we will state and prove relations that arise among some precedence constraints.

3.1.1 SEP Graphs

Here, we will define two kinds of directed graphs which we have called *SEP (Scheduling-Execution Precedence)* graphs, because they capture precedence relations that arise both during the scheduling and during the execution. These relations are different for each schedule and so are their associated graphs.

For both of these directed graphs, each node represents a job and the edges represent two different types of relations. The first type of directed edge represents the *thread flow*, that is, a directed edge between a pair of nodes means that both of them belong to a given thread and the first node precedes the second one in that thread.

The second type of directed edge represents the *execution flow*, that is, a directed edge between a pair of nodes means that both of them are executed on a given machine and the first node is executed before the second one in that machine.

The first type of graph, called *Particular SEP* graph considers only the relations corresponding to individual machines and -as it will be shown below- is related to the Start-End *On Each Machine* constraint. We can build one of these graphs for each machine and any given schedule.

The second type of graph, called *General SEP* graph considers the relations that arise not only within each machine alone but also among all the the machines together. As it will be shown, it is related to the *Extended Partial Order* constraint. There is only one of these graphs for each schedule.

- **Particular SEP Graph**: Given the schedule S, the Particular SEP Graph for machine $k, G_k(S) = (V_k^S, T_k^S \cup E_k^S)$ is a directed graph such that
  - $V_k^S = \{v_i : \text{in } S, \text{job } i \text{ is executed on machine } k\}$.
  - $(v_i, v_j) \in T_k^S$ if and only if

1. $\{v_i, v_j\} \subseteq V_k^S$
2. job i precedes job j in a given thread. This implies that job i will be scheduled before job j.

– $(v_i, v_j) \in E_k^S$ if and only if

1. $\{v_i, v_j\} \subseteq V_k^S$
2. job i is executed before job j on machine k for the schedule S.

Note that for schedule S, T_k^S represents the "thread flow" restricted to the jobs that are scheduled to machine k and E_k^S represents the "execution flow" of the jobs executed in machine k.

- **General SEP Graph**: A General SEP Graph $G(S) = (V, T \cup E^S)$ is a directed graph such that

  – $V = \{v_i : i \text{ is a job }\}$.

  – $(v_i, v_j) \in T$ if and only if

  1. $\{v_i, v_j\} \subseteq V$
  2. job i precedes job j in a given thread.

  – $E^S = \cup_{k=1}^{w} E_k^S$, where E_k^S is defined as in the Particular SEP graph.

  T represents the "threaded flow" and E^S represents the "execution flow" in the schedule S.

  Note that T includes not only the union of the T_K^S defined in the Particular SEP, but also the edges that represent the "thread flow" between jobs executed on different machines.

3.1.2 Relating SEP Graphs with Precedence Constraints

Here, we relate the SEP graphs with the precedence constraints. Firstly, we state and prove the relation between Start-End *On Each Machine* constraint and a set of *Particular SEP* graphs. Then, we associate the *Extended Partial Order* constraint with a *General SEP* graph.

Theorem 3.1.2.1 *A schedule S satisfies the Start-End* On Each Machine *constraint if and only if for every machine k the Particular SEP graph $G_k(S)$ has no cycles.*

Proof: It is enough to prove that for the schedule S, Start-End *On Each Machine* constraint is violated on machine k if and only if there is a cycle on the Particular SEP graph $G_k(S)$.

Firstly, suppose that for the schedule S, Start-End *On Each Machine* constraint is violated on machine k. Then, there are two jobs A and B, A precedes B in a given thread and A starts after B in machine k.

Then, in the Particular SEP graph $G_k(S) = (V_k^S, T_k^S \cup E_k^S)$, there is an edge from v_A to v_B in T_k^S (the "thread flow") and another edge from v_B to v_A in E_k^S (the "Machine execution flow" of machine k). It is clear now that there is a cycle on the Particular SEP Graph $G_k(S)$.

To prove the opposite implication, suppose that S satisfies the Start-End *On Each Machine* constraint. That is, given two jobs i and j that come from the same thread and are executed on the same machine, we have that if job i is scheduled before job j in S, then job i will be executed before job j in S.

This is equivalent to say that

$$(v_i, v_j) \in T_k^S \Rightarrow (v_i, v_j) \in E_k^S.$$

Now, suppose (by contradiction) that there is a cycle on the Particular SEP graph $G_k(S) = (V_k^S, T_k^S \cup E_k^S)$. Then, there exists a cycle where each of its edges belongs to E_k^S. This implies that any job associated to a node in this cycle must be executed before itself, which is a contradiction. □

Theorem 3.1.2.2 *A given schedule satisfies the* Extended Partial Order *constraint if and only if its General SEP graph has no cycles.*

Proof: It is a straightforward consequence from the definitions of the *Extended Partial Order* constraint and the General SEP graph. □

3.1.3 Relations among Precedence Constraints

To conclude this section, we state and prove some relations that connect the different precedence constraints. Firstly, we show that *Extended Partial Order* constraint is stronger than Start-End *On Each Machine* constraint, that is, that every schedule satisfying *Extended Partial Order* constraint also satisfies Start-End *On Each Machine* constraint.

Secondly, we show that *Start-Start* constraint is stronger than *Extended Partial Order* constraint and we will show an example of a family of schedules that satisfy *Extended Partial Order* constraint but do not satisfy *Start-Start* constraint.

After that, we show that *Start-End* constraint is stronger than *Start-Start* constraint.

Finally, we show that *Start-End* constraint is also stronger than *End-End* constraint.

Theorem 3.1.3.1 *Every schedule satisfying* Extended Partial Order *constraint also satisfies Start-End* On Each Machine *constraint.*

Proof: Given a schedule S, let $G_K(S)$ be a Particular SEP graph for some machine k and let $G(S)$ be a General SEP graph. Clearly (considering the Particular and General SEP graphs' definitions), if there is a cycle in $G_k(S)$ then there is a cycle in $G(S)$.

Therefore the claim follows from Theorem 3.1.2.1 and Theorem 3.1.2.2. □

Theorem 3.1.3.2 *Every schedule satisfying* Start-Start *constraint also satisfies* Extended Partial Order *constraint.*

Proof: Suppose that a given schedule S satisfies the *Start-Start* constraint and by contradiction, that S violates the *Extended Partial Order* constraint. Then, there is a cycle in the General SEP graph $G(S) = (V, T \cup E^S)$.

As *Start-Start* is satisfied, every edge $(v_i, v_j) \in T$ implies that job i must start no later than job j. Considering the definition of E^S, every edge $(v_i, v_j) \in E^S$ implies that job i is executed before job j in a given machine, and then, job i starts before job j.

As a result, each path of $G(S)$ from v_i to v_j with at least one edge in E^S means that job i must start before job j.

As by definition of T, there can not be cycles consisting only of edges of T, each cycle must have at least one edge in E^S. Then any node in a cycle is associated to a job that must start before itself, which is clearly a contradiction. □

The converse is not true. That is, a schedule that satisfies *Extended Partial Order* constraint may not satisfy *Start-Start* constraint. To show an example that supports this claim, we will use the following notation.

Let J be a job. Then $t(J)$ denotes the running time of J.

Here is the example:

In the case of a single thread $A : A_1, A_2, A_3, A_4$ such that

$$t(A_1) = 7,\ t(A_2) = 3,\ t(A_3) = 8,\ t(A_4) = 2,$$

the following schedule satisfies *Extended Partial Order* constraint:

machine 1: A_1A_2

machine 2: A_3A_4.

This schedule violates *Start-Start* constraint.

An analogous example for w threads:

$$\begin{array}{rcc} W_1 : A_1 & \cdots & A_{2m} \\ W_2 : B_{2,1} & \cdots & B_{2,m} \\ & \cdots & \\ W_w : B_{w,1} & \cdots & B_{w,m} \end{array}$$

$$\begin{array}{rl} t(A_{2i+1}) = 2^i, & \quad t(A_{2i+2}) = 2^m - 2^i \\ t(B_{i,j}) = \epsilon & \quad \text{where } w\epsilon < 1 \end{array}$$

The following schedule satisfies *Extended Partial Order* constraint (and is optimal for the makespan).

machine 1: $A_1 A_2 B_{2,1} \cdots B_{w,1}$

...

machine m: $A_{2m-1} A_{2m} B_{2,m} \cdots B_{w,m}$

This schedule violates *Start-Start* constraint. Furthermore, there is no schedule satisfying *Start-Start* constraint with the same cost as the schedule that satisfies *Extended Partial Order* constraint.

We may think each schedule as a transformation from an array of sequences (the threads) into another array of sequences (the sequences for each machine). In that case, it is interesting to observe the following property.

There exists a schedule $S : a[w] \rightarrow a[m]$ (from w threads into m machines) that satisfies *Extended Partial Order* constraint if and only if there exist schedules $S_1 : a[w] \rightarrow a[1]$ (from w threads into 1 machine) and $S_2 : a[1] \rightarrow a[m]$ (from 1 thread into m machines) such that both S_1 and S_2 satisfy Start-End *On Each Machine* constraint and $S = S_2 o S_1$.

Note that a schedule $S : a[w] \rightarrow a[1]$ satisfies Start-End *On Each Machine* constraint if and only if S satisfies *Extended Partial Order* constraint and a schedule $S' : a[1] \rightarrow a[m]$ satisfies Start-End *On Each Machine* constraint if and only if S' satisfies *Extended Partial Order* constraint.

Theorem 3.1.3.3 *Every schedule satisfying* Start-End *constraint also satisfies* Start-Start *constraint.*

Proof: Follows directly from the definitions of *Start-End* and *Start-Start* constraints. □

Theorem 3.1.3.4 *Every schedule satisfying* Start-End *constraint also satisfies* End-End *constraint.*

Proof: Follows directly from the definitions of *Start-End* and *End-End* constraints. □

3.2 Relating On-line Paradigms with Precedence Constraints

In this section we relate on-line paradigms with precedence constraints. Firstly, we note that *Scheduling Jobs One By One* paradigm does not force any precedence constraint.

Secondly, we state and prove that any schedule can be obtained under the *Scheduling Before Executing* paradigm if and only if it does not violate *Extended Partial Order* constraint.

After that, we prove that any schedule can be obtained under the *Scheduling On Execution* paradigm if and only if it does not violate *Start-Start* constraint.

Finally, the fact that every schedule obtained under the *Scheduling On Execution* paradigm can be obtained under the *Scheduling Before Executing* paradigm is deduced as a corollary.

Proposition 3.2.0.5 Scheduling Jobs One By One *paradigm does not imply any precedence constraint, i.e., every schedule is feasible under the* Scheduling Jobs One By One *paradigm.*

Theorem 3.2.0.6 *Schedules that can be obtained under the* Scheduling Before Executing *paradigm are exactly those that do not violate the* Extended Partial Order *constraint.*

Proof: Firstly, we will prove that every schedule obtained under the *Scheduling Before Executing* paradigm does not violate *Extended Partial Order* constraint. After that, we will prove that every schedule satisfying *Extended Partial Order* constraint can be obtained under the *Scheduling Before Executing* paradigm.

Let us recall the definition of *Extended Partial Order* constraint.

There is no sequence of jobs $A_1, \dots, A_k$ such that

- $A_i \rightarrow A_{i+1} \forall i \in \{1, \dots, k-1\}$
- $A_k \rightarrow A_1$,

where $X \rightarrow Y$ means that *job X precedes job Y in the same thread* or *job X is scheduled to be executed before job Y in the same machine.*

We can infer that under the *Scheduling Before Executing* paradigm, if job X precedes job Y in a given thread then X is scheduled before Y. Therefore, under the *Scheduling Before Executing* paradigm, $X \rightarrow Y$ implies that X is scheduled before Y and, by transitivity, $X_1 \rightarrow \dots \rightarrow X_j$ implies that X_1 is scheduled before X_j.

Now it is clear that a cycle of "$\rightarrow$" would imply that each node belonging to it has been scheduled before itself, which is a contradiction.

To prove that every schedule satisfying *Extended Partial Order* constraint can be obtained under the *Scheduling Before Executing* paradigm, we will use Theorem 3.1.2.1.

Let S_1 be a schedule satisfying *Extended Partial Order* constraint, and let $G(S_1) = (V, T \cup E^{S_1})$ be the General SEP graph of S_1. By Theorem 3.1.2.1, $G(S_1)$ is a DAG (directed acyclic graph). To simplify the proof, we will think that S_1 does not wait, although a similar proof may be applied if this is not the case.

We will prove that the following procedure, that can be applied under the *Scheduling Before Executing* paradigm, produces a schedule S_2 that is equal to S_1.

```
$S_2 \leftarrow \emptyset$
While $V$ is not empty
    Pick $v_i \in V$ such that $d_{out}(v_i) = 0$
    $V \leftarrow V \setminus \{v_i\}$
    assign job $i$ to the first available space of $S_2$ in the same
        machine as in $S_1$
        (same time slot as in $S_1$)
```

As each job is scheduled to the same machine in S_1 as in S_2 and the *thread flow* is the same for S_1 and S_2, we must prove that the *execution flow* is also the same for both of them to prove that $S_1 = S_2$. That is, we must prove that any given pair of jobs executed in the same machine (both in S_1 and S_2) have the same execution order in S_1 than in S_2.

Suppose, by contradiction, that in S_1 job i is scheduled before job j and in S_2 job j is scheduled before job i. By construction, when j was scheduled in S_2, v_j has no *outgoing* edges in $G(S_1)$; Thus, considering that j and i are scheduled to the same machine and that j was scheduled when there were no edges from v_j to v_i in $G(S_1)$, we can infer that j is scheduled before i in S_1, which is a contradiction.

To conclude, we must prove that this procedure may be applied under the *Scheduling Before Executing* paradigm. To show that, it suffices to prove that at the moment a job is scheduled, either this job is the first one of a thread, or all the preceding jobs in that thread have already been scheduled.

Suppose, by contradiction, that job i precedes job j in the same thread and job j is scheduled before job i in S_2. As job i precedes job j in the same thread, then there is an edge from v_i to v_j in $G(S_1)$. Thus, by construction, job i cannot be scheduled before j in S_2, which is a contradiction. $\square$

Theorem 3.2.0.7 *Schedules that can be obtained under the* Scheduling On Execution *paradigm are exactly those that do not violate the* Start-Start *constraint.*

Proof: First we will prove that every schedule obtained under the *Scheduling On Execution* paradigm satisfies *Start-Start* constraint.

Let A and B be two jobs in a schedule obtained under the *Scheduling On Execution* paradigm such that A precedes B in a given thread. As in *Scheduling On Execution* paradigm A must start its execution at the same moment in which it is scheduled (and A must be scheduled before B), then A must start its execution before B.

Now, we will prove that every schedule that satisfies *Start-Start* constraint can be obtained under the *Scheduling On Execution* paradigm.

Let S be a schedule satisfying *Start-Start* constraint. It suffices to prove that every job can be executed under the *Scheduling On Execution* paradigm at the same time it is scheduled for execution in S. Suppose this is false.

Let i be the first job in S that is scheduled for execution at a time that cannot be executed under the *Scheduling On Execution* paradigm. The only reason for which i may not be available for execution under the *Scheduling On Execution* paradigm is that a job j preceding it in the same thread has been scheduled for execution at a later time.

To prove that this is a contradiction, it is enough to notice that i cannot be scheduled for execution at a later time than j because it would violate the *Start-Start* constraint. □

Corollary 3.2.0.8 *Every schedule that may be obtained under the* Scheduling On Execution *paradigm may also be obtained under the* Scheduling Before Executing *paradigm.*

Proof: It is a consequence of Theorem 3.2.0.6, Theorem 3.2.0.7 and Theorem 3.1.3.2. □

3.3 What about waiting?

In this section we tackle the following question: *how much worth is it to wait?* In other words, can we obtain any benefit if we let one or more machines to have some idle time before executing a job? Is there any advantage in postponing a job's execution?

This question will be considered under the three main paradigms we are studying, without any precedence constraints and for *Makespan* and *Total Completion Time* cost functions.

Firstly, we will focus on optimal schedules obtained off-line.

Definition 3.3.1 *We will consider that machine p is waiting at time t in a given schedule if at time t, p is not executing any job and there is a job ready and waiting to be scheduled for (immediate) execution. In this case p will be called* waiting machine.

Definition 3.3.2 *A schedule that is produced in the preceding situation will be called* waiting schedule*.*

Theorem 3.3.0.9 *For the* Total Completion time *cost function, any optimal schedule obtained under the* Scheduling On Execution *paradigm is not a waiting schedule.*

Proof: Suppose there exists an optimal schedule S such that on its execution there exists a machine p that starts waiting at time t for δ units of time. Let A be the job scheduled to machine p at time $t+\delta$.

- If A is ready at time t, then let S' be the schedule obtained by scheduling A to machine p at time t and all the other jobs at the same time as in schedule S. It is clear that S' is feasible under the *Scheduling On Execution* paradigm and its cost is smaller than the cost of S, which is a contradiction as S was optimal.
- If A is not ready at time t, then there exists a job B, preceding A in the same thread, which will start being executed before A, at time $t+\epsilon$ $(0<\epsilon<\delta)$. Suppose that B is scheduled to machine q. Now let S' be a schedule equal to S except that the jobs scheduled to start after time t on machine q (including job B) are swapped with the jobs scheduled to start after time t on machine p (including job A). It is clear that S' is feasible under the *Scheduling On Execution* paradigm and its cost is the same as the cost of S. Let S'' be a schedule equal to S' except that job B is scheduled to begin at time t (instead of time $t+\epsilon$). As in the previous case, S'' is feasible under the *Scheduling On Execution* paradigm and has smaller cost than the one of S, which is a contradiction.

□

Note: The extension of Theorem 3.3.0.9 for the case where *release times* are used does not hold. The following example supports this claim.

Thread A: two jobs, A_1, A_2 with release times 0 and 2 respectively, both of length 1.

Thread B: One job B_1 with release time 0 and length 8.

With only one machine, the optimal schedule is $A_1A_2B_1$ with cost $1+3+11=15$ where job A_2 waits one unit of time before starting its execution. This schedule is better than any schedule that does not wait.

Theorem 3.3.0.10 *Under the* Scheduling On Execution *paradigm and for the* Makespan *cost function there exists an optimal schedule in which no machine is waiting at any time.*

Proof: Let I : *schedules* $\rightarrow$ *time* be a function, $I(S)=t$, where t is the time at which the first waiting machine(s) starts waiting in S. That is, in S no machine starts waiting before time t. For schedules such that no machine waits, we define $I(S)=+\infty$.

Let $\mathcal{S}=\{\mathcal{S} : \mathcal{S}$ is an optimal schedule and for all feasible schedule S', $I(S) \geq I(S')\}$. That is, $\mathcal{S}$ is the set of optimal schedules that "wait as late as possible".

Let $\mathcal{S}'=\{\mathcal{S} : \mathcal{S} \in \mathcal{S}$ such that the number of machines that start to wait at time $I(S)$ is minimum $\}$.

Let S_1 be a schedule such that $S_1 \in \mathcal{S}'$. To prove that S_1 "does not wait", suppose by contradiction that S_1 has a machine p that starts waiting at time t for δ units of time. Let A be the job scheduled to machine p at time $t+\delta$.

- If A is ready at time t, then let S_2 be the schedule obtained by scheduling A to machine p at time t and all the other jobs at the same time as in schedule S_1. It is clear that $S_1 \in \mathcal{S}$ and $S_2 \in \mathcal{S}$ and the number of machines that begin to wait at time $I(S_1)$ is greater than the number of machines that begin to wait at time $I(S_2)$. This fact contradicts the assumption that $S_1 \in \mathcal{S}'$.
- If A is not ready at time t, then there exists a job B, preceding A in the same thread, which will start to be executed before A, at time $t+\epsilon$ ($0 \leq \epsilon < \delta$). Suppose that B is scheduled to machine q. Now let S_2 be a schedule equal to S except that the jobs scheduled to start after time t on machine q (including job B) are swapped with the jobs scheduled to start after time t on machine

p (including job A). It is clear that $S_2 \in \mathcal{S}'$. Note that we are now in the preceding case.

Let S_3 be a schedule equal to S_2 except that job B is scheduled to begin at time t (instead of time $t+\epsilon$). As in the previous case, $S_3 \in \mathcal{S}$ and the number of machines that begin to wait at time $I(S_2)$ is greater than the number of machines that begin to wait at time $I(S_3)$. This fact contradicts the assumption that $S_2 \in \mathcal{S}'$.

□

Note: Theorems 3.3.0.9 and 3.3.0.10 say that under the *Scheduling On Execution* paradigm there is always an optimal schedule with respect to the Makespan and Total Completion Time cost functions such that it is not a waiting schedule. The intuition behind this fact is that as the release time of each job is at the time the preceding job is scheduled (starting its execution), there is no gain of information while waiting -no new job will get ready- and so, waiting will not improve the performance of a schedule.

The preceding theorems regards *Scheduling On Execution* paradigm. The case of *Scheduling Jobs One By One* paradigm is similar if we define "waiting" as "not scheduled to the first available time slot (of some machine)" and is stated in the following two proposition which will not be proven. For the case of *Scheduling Before Executing* paradigm this kind of analysis makes no sense.

Proposition 3.3.0.11 *Any optimal schedule with respect to the* Total Completion time *cost function obtained under the* Scheduling Jobs One By One *paradigm is not a waiting schedule.*

Proposition 3.3.0.12 *There exists an optimal schedule with respect to the* Makespan *cost function obtained under the* Scheduling Jobs One By One *paradigm in which no machine is waiting at any time.*

The preceding propositions are particular cases of scheduling problems in general, where if no release times are considered and no precedence constraints are present the same results also hold.

In Theorems 3.3.0.9 and 3.3.0.10 and the preceding propositions it is stated that both for the *Makespan* and *Total Completion Time* cost functions, waiting does not improve the cost of a schedule. In other words, in those cases there exists an optimal schedule that executes one job after another without leaving any machine idle before that machine ends the processing of the sequence of jobs it has been assigned.

Although we have treated the case where no precedence constraints are present, by the theorems of Chapter 3, these theorems and propositions are also valid for the case where *Extended Partial Order* constraint or *Start-Start* constraint is present. For the other precedence constraints some further analysis is required.

Until now, we have been talking about *optimal* schedules. In the cases we have treated, the conclusion was that we can find an optimal schedule that does not wait. Do on-line algorithms hold similar properties? Can we say that, for an on-line algorithm waiting does not improve performance?

For each of the preceding cases ... is there an on-line algorithm that achieves the best competitive ratio producing schedules that do not wait? In other words, is there an on-line algorithm that achieves a better competitive ratio than any other on-line algorithm that produces schedules that wait?

Theorem 3.3.0.13 *Under the* Scheduling On Execution *paradigm and with respect to both* Makespan *and* Total Completion Time *cost functions, for every on-line algorithm that waits there is another on-line algorithm that does not wait and has not worse competitive ratio.*

The proof of this theorem will not be provided because it is very similar to the one provided for optimal schedules. It may be done simulating the flow of the time, considering that no new information (no new jobs) will be available if time passes. In such a case, an algorithm that does not wait can make similar decisions than another one that waits, just simulating the flow of the time.

Theorem 3.3.0.14 *Under the* Scheduling Jobs One By One *paradigm and with respect to* Makespan *cost function, for every on-line algorithm that waits there is another on-line algorithm that does not wait and has not worse competitive ratio.*

Again, the proof of this theorem will not be provided because it is similar to the preceding ones. Even though, it is worthwhile noticing that, as it will be proven in the next chapter, if we consider Makespan cost function, algorithm working under the *Scheduling Jobs One By One* and *Scheduling Before Executing* paradigms have the same competitive ratio. Thus, the result of the theorem can easily be deduced from this fact.

Even we have not reached any proof, we believe this result does not apply if we consider Total Completion Time cost function. We base this intuition in the fact that if waiting is not allowed, it can easily be proved that no competitive algorithm can be achieved, while if waiting is allowed, the analysis is complex.

Chapter 4

Competitiveness Results

In this chapter, we present the competitiveness results related to some of the contexts that have been explained above. For all of these contexts, we will assume that the schedules are not required to satisfy any precedence constraint, unless otherwise stated.

We will analyze the different contexts considering two objective functions. In the first section, minimizing the *Makespan* will be our goal. In the second one, we will focus on the *Total Completion Time* cost function.

At last, we make a short analysis considering some cases were *Start-End* constraint is applied.

First of all, we will state two lemmas and a proposition that were pointed out in [11] and are valid for both the *Makespan* and the *Total Completion Time* objective functions.

Lemma 4.0.0.15 *Any algorithm for the on-line single-threaded scheduling problem under the* Scheduling On Execution *paradigm is* 1*-competitive.*

Proof: At any point in time there is only one job available for scheduling. Therefore an algorithm that schedules each job as soon as a machine becomes available, yields the optimal solution. □

Lemma 4.0.0.16 *Any algorithm for the on-line single-threaded single machine scheduling problem under the* Scheduling Before Executing *paradigm is* 1*-competitive.*

Proof: There is only one feasible solution for such a problem. □

Proposition 4.0.0.17 *Lower bounds on the competitiveness of on-line algorithms for single-threaded problems are also valid for the multi-threaded versions.*

Proof: This is easily seen from the fact that in a w-threaded problem it is always possible to have $w - 1$ threads empty or, if this is not appreciated, filled with negligible jobs. □

4.1 Makespan

The *Makespan* objective function measures the time at which the last job among all the threads is completed. This problem has an advantage over its counterpart with the *Total Completion Time* objective, because the order in which jobs are executed on the machines does not influence the *Makespan.*

Proposition 4.1.0.18 *There exists a c-competitive algorithm under the* Scheduling Jobs One By One *paradigm with respect to the makespan cost function if and only if there exists a c-competitive algorithm under the* Scheduling Before Executing *paradigm with respect to the same cost function.*

Proof: To prove this proposition, it is enough to present the following two facts. First, that optimal (off-line) schedules obtained under each of these paradigms have the same makespan cost. This is so because a schedule obtained under *Scheduling Jobs One By One* may be reordered into another schedule such that the last schedule can also be obtained under *Scheduling Before Executing* and each job is assigned to the same machine as in the first schedule (probably in a different order).

Second, that a schedule with the same makespan as the one obtained by an on-line algorithm under *Scheduling Jobs One By One* may be obtained under

Scheduling Before Executing. To show this is true, it suffices to note that each job can be assigned to the same machine as to the one used by an algorithm processing under *Scheduling Jobs One By One*, but to the first available time slot, with the same makespan cost. $\square$

The following theorem holds for the single-threaded case and appeared first in [11].

Theorem 4.1.0.19 *Given any algorithm A for the single-threaded makespan problem having competitive ratio ρ, there exists an algorithm A_w for the multi-threaded version also with competitive ratio ρ.*

With the goal of minimizing the Makespan objective function Graham in 1966 [15] conceived a simple deterministic greedy algorithm, called nowadays List Scheduling, for the non-threaded scheduling problem.

As pointed out in [28], the algorithm works for *Scheduling Jobs One By One* and *Scheduling On Execution* paradigms both with known running times (called in [28] *Jobs arrive over time*) and unknown running times, for the non-threaded version. In addition, it also works for the *Scheduling Before Executing* paradigm introduced in this work.

Theorem 4.1.0.20 *[15, 28] (for the non-threaded version) List Scheduling Algorithm (Fig. 4.1) is $(2 - \frac{1}{m})$-competitive with respect to the Makespan objective function.*

In this algorithm, both for *Scheduling Jobs One By One* and *Scheduling Before Executing* paradigms, we can rephrase “scheduling to an idle machine” as “scheduling to the least loaded machine”.

List Scheduling for a non-threaded input has proven to be an optimal deterministic algorithm for the cases $m = 2$ and $m = 3$. For $m \geq 4$, better competitive deterministic algorithms have been given, although they have not proven to be optimal. For arbitrary m, a better algorithm was first developed in [6] with competitive ratio 1.9230. Its analysis is relatively complicated. Lower bounds for this problem -1.8520 for large m- were presented in [1, 5].

```
Process the jobs in any feasible order

Whenever some machine is idle, we schedule on it the first
         available job (i.e., it is not scheduled yet and the
         current time is greater than its release time).
```

Figure 4.1: List Scheduling Algorithm

Better competitive ratios have been obtained using randomized algorithms. For both deterministic and randomized algorithms, references and summarized results are given in [28].

The same algorithm also works for the multi-threaded version.

Theorem 4.1.0.21 *[11] List Scheduling is a $(2 - \frac{1}{m})$-competitive algorithm for the multi-threaded makespan problem under any of the three main paradigms.*

Proof: It is known that List Scheduling is $(2 - \frac{1}{m})$-competitive for any on-line makespan problem, independently of the order in which the jobs are considered for scheduling on the machines, as long as no idle time on any of the machines occurs, i.e., as long as no machine is not busy while there are still jobs available for processing. Clearly, List Scheduling combined with a round robin routine for dealing with the threads gives a schedule that satisfies this property. $\square$

It is worthwhile noticing that under the *Scheduling On Execution* paradigm, this algorithm proves to be optimal and almost optimal for the cases $w \geq m$ and $w < m$ respectively, as shown by the following two theorems.

Theorem 4.1.0.22 *[11] Under the* Scheduling On Execution *paradigm, any on-line algorithm for the multi-threaded makespan problem is at least $(2 - \frac{1}{m})$-competitive for $w \geq m$. This means that for this problem, List Scheduling Algorithm is strongly competitive.*

Proof: Consider the following adversarial strategy. There are m threads given. Each of the threads begins with $m - 1$ jobs with processing time 1, which we call small jobs hitherto. One of the threads has additionally a big job with processing time m.

Given any on-line algorithm one of the m last small jobs in the threads will be scheduled last and cannot be started before time $m(m-1)/m = m-1$. The adversary "hides" the big job behind this job in the same thread. Thus, also the big job cannot be started by the on-line algorithm before time $m-1$, and the resulting makespan is at least $2m-1$.

In the optimal solution the $m-1$ small jobs in the thread containing the big job are scheduled first (at time 0) on the first $m-1$ machines and the big job is started also at time 0 on the m-the machine. After that the remaining $(m-1)^2$ small jobs are equally divided over the first $m-1$ machines, yielding a makespan of m. $\square$

Theorem 4.1.0.23 *Under the* Scheduling On Execution *paradigm, any on-line algorithm for the multi-threaded makespan problem is at least* $(2-\frac{1}{m}-\frac{1}{w})$*-competitive for* $w < m$.

The proof for this theorem is similar to the preceding one.

Theorem 4.1.0.24 *Under the* Scheduling Jobs One By One *and* Scheduling Before Executing *paradigms, the competitive ratio's lower and upper bounds for the non-threaded makespan problem also hold for the multi-threaded one.*

Proof: Considering that the non-threaded and single-threaded problems under the *Scheduling Jobs One By One* and *Scheduling Before Executing* paradigms are the same, the result can be inferred from Theorem 4.1.0.19. $\square$

4.2 Total Completion Time

The *Total Completion Time* is the sum of the completion times of the jobs. Unlike in the Makespan problem, for the *Total Completion Time* the on-line algorithm has to worry about the order in which jobs are executed on the machines.

All the results presented in this section will correspond to *Total Completion Time* objective.

To minimize this objective function off-line, it is easy to see that the optimum is obtained scheduling first the jobs with small running times when possible.

For the *Scheduling Before Executing* and *Scheduling On Execution* paradigms we have reached important results. We have found a strongly competitive algorithm under *Scheduling Before Executing* and *Scheduling On Execution* paradigms for the single machine case. Also for the same paradigms, we have found that no competitive algorithm can be obtained while processing in more than one machine.

Firstly, we will analyze the single machine case.

4.2.1 Scheduling Before Executing and Scheduling On Execution for the single machine case

We will present first the lower bound result for our present problem and after that we will show an on-line algorithm that achives the same value as an upper bound.

Lower bounds

The following theorem and its proof were presented in [11] regarding a lower bound to our problem under the *Scheduling Before Executing* paradigm and the single machine case. We have found that this result also applies for the *Scheduling On Execution* paradigm (also for the Total Completion Time objective function and the single machine case), with the same proof. The lower bound for these problems equals the number of threads.

Theorem 4.2.1.1 *Any on-line algorithm that is ρ-competitive under* Scheduling Before Executing *and* Scheduling On Execution *paradigms, for the single machine w-thread problem must have $\rho \geq w$.*

Proof: The proof for the single-threaded case ($w = 1$) is trivial. We will assume $w \geq 2$.

To prove the theorem we establish an adversary input sequence that will force any algorithm to be at best w-competitive. The w threads have each of them a job with processing time 1 at the first position. Then only one of the threads has after this job a sequence of n jobs with processing time ϵ each. The adversary will “hide” this sequence of n jobs in the thread that will be explored last by the on-line algorithm.

For this sequence the optimal schedule is obviously to start processing the first job of the thread that contains the n small jobs. Then these n jobs with processing time ϵ are processed and afterwards the remaining $w-1$ jobs leading to a sum of completion times

$$\begin{aligned}
z^{OPT} &= 1+n+\frac{1}{2}n(n+1)\epsilon+(1+n\epsilon)(w-1)+\frac{1}{2}(w-1)w \\
&= 1+n+\frac{1}{2}n^2\epsilon+\frac{1}{2}n\epsilon+w-1+n(w-1)\epsilon+\frac{1}{2}w^2-\frac{1}{2}w \\
&= n+\frac{1}{2}w+\frac{1}{2}w^2-\frac{1}{2}n\epsilon+\frac{1}{2}n^2\epsilon+\frac{1}{2}nw\epsilon.
\end{aligned}$$

The sum of completion times of the on-line algorithm is given by

$$\begin{aligned}
z^{OL} &= \frac{1}{2}w(w+1)+nw+\frac{1}{2}n(n+1)\epsilon \\
&= nw+\frac{1}{2}w+\frac{1}{2}w^2+\frac{1}{2}n\epsilon+\frac{1}{2}n^2\epsilon.
\end{aligned}$$

Therefore, the ratio between the two tends to w if $\epsilon = o(n^{-3})$ and n tends to infinity. □

Upper bounds

The first attempt to develop a competitive algorithm for this problem was the following.

ShortestFirst Algorithm: “Among the ready unserved jobs of each thread, pick the shortest one and schedule it”.

The following instance shows that *ShortestFirst algorithm* is not c-competitive for any constant c.

There are two non-empty threads, one with n_1 jobs of length $1-\epsilon$ and the other non-empty thread starts with a job of length 1 and after it, n_2 jobs of length ϵ ($\epsilon < 1$).

The cost of ShortestFirst algorithm is

$$\begin{aligned}\sum_{i=1}^{n_1} i(1-\epsilon) + n_1(1-\epsilon) + 1 + \sum_{i=1}^{n_2}(n_1(1-\epsilon)+1+i\epsilon) &\geq \\ \sum_{i=1}^{n_2} n_1(1-\epsilon) &= \\ n_2n_1(1-\epsilon)\end{aligned}$$

The optimal way (off-line) of assigning them is: process the job of length 1, then the n_2 jobs of length ϵ and finally the n_1 jobs of length $1-\epsilon$.

Then, the optimal cost is

$$\begin{aligned}1 + \sum_{i=1}^{n_2}(1+i\epsilon) + \sum_{i=1}^{n_1}(1+n_2\epsilon+i(1-\epsilon)) &= \\ 1 + n_2 + \frac{n_2+1}{2}n_2\epsilon + n_1(1+n_2\epsilon) + \frac{n_1+1}{2}n_1(1-\epsilon) &\leq \\ 1 + n_2 + \frac{n_2+n_2}{2}n_2\epsilon + n_1(1+n_2\epsilon) + \frac{n_1+n_1}{2}n_1(1-\epsilon) &= \\ 1 + n_2 + n_2^2\epsilon + n_1 + n_1n_2\epsilon + n_1^2(1-\epsilon)\end{aligned}$$

Therefore, any c-competitive algorithm must verify

$$c \geq \frac{n_2n_1(1-\epsilon)}{1+n_2+n_2^2\epsilon+n_1+n_1n_2\epsilon+n_1^2(1-\epsilon)}$$

We can think that $\epsilon = n^{-2}$, $n_1 = n$ and $n_2 = n^2$, in which case

$$c \geq \frac{n^2n(1-n^{-2})}{1+n^2+n^4n^{-2}+n+nn^2n^{-2}+n^2(1-n^{-2})} = \frac{n^2-1}{3n+2}$$

This term cannot be bounded by any constant.

```
H[i]:  Head of the $i$-th thread (first job of this thread now)
Sum[i]:  Total processing time of the $i$-th thread up to now

for $i : 1 \ldots w$
       Sum[i] $\leftarrow$ 0

while there exists an unserved job do
       Choose $i : i = min_j\{$Sum[j] + RunningTime(H[j]):  there is an
              available job in the $j$-th thread ready to be served$\}$
       process H[i]
       Sum[i] $\leftarrow$ Sum[i] + RunningTime(H[i])
```

Figure 4.2: BalanceProcessing Algorithm

BalanceProcessing Algorithm

We present now *BalanceProcessing Algorithm* (Fig. 4.2), and we will show that it is strongly competitive (i.e., achieves the same competitive factor as the lower bound).

The intuitive idea behind *BalanceProcessing Algorithm* is as follows. At any given moment t and for each thread the algorithm maintains the processing time it requires to serve all the jobs of that thread that have been scheduled up to t. We have to minimize all along the processing the maximal processing time among all the threads, so as to keep the partial processing times of the threads *balanced.* In other words, BalanceProcessing Algorithm chooses the thread that would have the smallest cumulative processing time after serving its present available job. A similar idea for the *k-server problem* appeared in [21].

To prove that *BalanceProcessing Algorithm* is w-competitive we will first introduce some notation, and then the theorem will be proven using the lemma below.

Notation

$S_k[i]$: total processing time of the i-th thread after the first k jobs among all the threads have been processed.

j_i: number of jobs of the i-th thread

w_{ik}: k-th job of the i-th thread

$H_k[i]$: first job of the i-th thread not processed after the first k jobs among all the threads have been processed.

p_n: n-th job processed.

$C^A(p_k)$: total time used by the first k jobs processed.

Lemma 4.2.1.2 $p_n = H_{n-1}[i] \Rightarrow S_n[i] \geq S_n[j] \forall j \in \{1, \ldots, w\} \forall n \geq 0$. *That is, if the last processed job belongs to the i-th thread, then the total processing time of the i-th thread is not below the total processing time of any other thread.*

Proof: The proof will be done using induction on the number of processed jobs (n).

The basic case, where $n = 1$, is simple: $S_1[i] = H_0[i] \geq 0 = S_1[j] \forall j \neq i$.

Suppose now that $n \geq 2$. Let $p_{n+1} = H_n[i]$.

We will prove that $S_{n+1}[i] \geq S_{n+1}[j] \forall j$. Let $p_n = H_{n-1}[k]$

- if $k = i$, that is, if the n-th and $(n+1)$-th processed jobs belong to the same thread:

  $$S_{n+1}[i] = S_{n+1}[k] \geq S_n[k] \geq_{\text{(induction hypothesis)}} S_n[j] = S_{n+1}[j] \forall j \neq k$$

- if $k \neq i$, that is, if the n-th and $(n+1)$-th processed jobs do not belong to the same thread:

  By construction,

  $$S_{n-1}[k] + H_{n-1}[k] \leq S_{n-1}[j] + H_{n-1}[j] \forall j \tag{4.1}$$

  By (4.1),

  $$S_{n-1}[k] + H_{n-1}[k] \leq S_{n-1}[i] + H_{n-1}[i] \tag{4.2}$$

By the induction hypothesis,

$$S_n[k] \geq S_n[j] \forall j \tag{4.3}$$

$$\begin{aligned} S_{n+1}[i] &= S_n[i] + H_n[i] = \\ & S_{n-1}[i] + H_{n-1}[i] \geq \text{ by (4.2)} \\ & S_{n-1}[k] + H_{n-1}[k] = S_n[k] \geq \text{ owing to (4.3)} \\ & S_n[j] = S_{n+1}[j] \forall j \neq k. \end{aligned}$$

To complete the proof, it is enough to show that $S_{n+1}[i] \geq S_{n+1}[k]$:

$$\begin{aligned} S_{n+1}[k] = S_n[k] &= S_{n-1}[k] + H_{n-1}[k] \leq \text{(by construction)} \\ &\leq S_{n-1}[i] + H_{n-1}[i] = S_n[i] + H_n[i] = S_{n+1}[i] \end{aligned}$$

□

Theorem 4.2.1.3 *BalanceProcessing algorithm is w-competitive under* Scheduling Before Executing *and* Scheduling On Execution *paradigms, for the single machine w-thread problem with respect to the Total Completion Time objective.*

Proof: Let $w_{ij} = p_n$. Then

$$C^{OL}(w_{ij}) = \sum_{k=1}^{w} S_n[k] \leq \sum_{k=1}^{w} S_n[i]$$

owing to Lemma 4.2.1.2. Besides, $S_n[i] \leq C^{OPT}(w_{ij})$, because the optimal algorithm must process at least $w_{i1}, \ldots, w_{i(j-1)}$ before processing w_{ij}. So,

$$C^{OL}(w_{ij}) \leq \sum_{k=1}^{w} S_n[i] = wS_n[i] \leq wC^{OPT}(w_{ij})$$

Then, the Total Completion Time cost of the on-line algorithm is

$$C^{OL} = \sum_{i=1}^{w} \sum_{j=1}^{j_i} C^{OL}(w_{ij}) \leq \sum_{i=1}^{w} \sum_{j=1}^{j_i} wC^{OPT}(w_{ij}) = wC^{OPT}$$

□

4.2.2 Scheduling Before Executing, $m \geq 2$

We will show now that if the jobs are processed in more than one machine ($m \geq 2$), then there is no competitive algorithm under both *Scheduling Before Executing* or *Scheduling On Execution* paradigms. We begin analyzing the first of those paradigms.

Theorem 4.2.2.1 *No competitive on-line algorithm can be obtained under* Scheduling Before Executing *paradigm, with respect to the Total Completion Time cost function and for $m \geq 2$.*

Proof: For simplicity, let us study first the case $m = 2$. An instance where no competitive on-line algorithm can be found is the following: $w - 1$ empty threads and one thread beginning with a job of length h^2 and h jobs of length 1.

- Suppose that the on-line algorithm processes all the jobs on a unique given machine. Its cost would be

$$h^2 + \sum_{i=1}^{h}(h^2 + i) = (h+1)(h^2 + h/2)$$

  The optimal algorithm would process the jobs on two machines having a cost of

$$h^2 + \sum_{i=1}^{h} i = h^2 + h(h+1)/2$$

  So the on-line algorithm would be at least c-competitive, where

$$c = \frac{(h+1)(h^2 + h/2)}{h^2 + h(h+1)/2} = 1 + \frac{2h^2}{3h+1}$$

  which is bounded by no constant.

- If the on-line algorithm processes any job in the second machine, the thread goes on with n jobs of length ϵ. In this case, the n jobs of length ϵ are scheduled behind one or more jobs. Thus, the on-line algorithm has cost not better than

$$h^2 + \sum_{i=1}^{h} i + n(1+\epsilon) = h^2 + h(h+1)/2 + n + n\epsilon = h(3h+1)/2 + n + n\epsilon$$

The optimal schedule is not worse than a schedule that processes in one machine the first $h+1$ jobs and the n remaining jobs in the other. Its cost is

$$h^2 + \sum_{i=1}^{h}(h^2+i) + \sum_{i=1}^{n} i\epsilon = (h+1)(h^2+h/2) + \frac{n(n+1)}{2}\epsilon$$

So the on-line algorithm will be at least c-competitive, where

$$c = \frac{h(3h+1)/2 + n + n\epsilon}{(h+1)(h^2+h/2) + \frac{n(n+1)}{2}\epsilon}$$

which tends to infinity when $\epsilon = o(n^{-2})$ and n tends to infinity.

For the case $m \geq 2$ we generalize the instance above. The new instance is built taking initially the same instance as in the case $m = 2$, that is, a job of length h^2 and then h jobs of length 1. As shown above, any on-line algorithm must use at least 2 machines to try to be c-competitive for some constant c. It uses the machines 1 and 2 (wlog).

The thread goes on with k groups of jobs (k will be defined later). The i-th group will be composed of n^i jobs, each one of length n^{-2i}. Other threads are empty. Group $i+1$ follows group i. Each group i assigns at least n^i/m jobs to one machine. Each of these machines (receiving at least n^i/m jobs from group i) will be called "*loaded*" machine. Machines 1 and 2 will be called loaded, too. Group k (the last one) will be the first to load (to assign at least n^k/m jobs to) an already loaded machine. Note that as each assigned group loads at least one machine, loading all the machines (without re-loading any machine twice) will take at most $m-2$ groups. Then $k \leq m-1$.

The cost of assigning the $\frac{n^t}{m}$ jobs (at least) of group t that load one machine will be at least

$$\sum_{i=1}^{n^h/m} in^{-2h}.$$

As group k will load at least one machine already loaded, the cost of scheduling the $\frac{n^k}{m}$ jobs (at least) of group k that load one machine already loaded will be at least

$$\sum_{i=1}^{n^k/m} (\frac{n^{k-1}}{m}n^{2-2k} + in^{-2k}).$$

As a result, any on-line algorithm will have a cost of at least

$$\begin{aligned}
& h^2+\frac{h}{m}+\sum_{i=1}^{n/m} in^{-2}+\sum_{i=1}^{n^2/m} in^{-4}+ \\
& \cdots+\sum_{i=1}^{n^{k-1}/m} in^{2-2k}+\sum_{i=1}^{n^k/m}\left(\frac{n^{k-1}}{m}n^{2-2k}+in^{-2k}\right)= \\
= \; & h^2+\frac{h}{m}+\frac{n}{m}\frac{\frac{n}{m}+1}{2}n^{-2}+\frac{n^2}{m}\frac{\frac{n^2}{m}+1}{2}n^{-4}+ \\
& \cdots+\frac{n^{k-1}}{m}\frac{\frac{n^{k-1}}{m}+1}{2}n^{2-2k}+\frac{n^k}{m}\frac{n^{k-1}}{m}n^{2-2k}+\frac{n^k}{m}\frac{\frac{n^k}{m}+1}{2}n^{-2k}\geq \\
\geq \; & h^2+\frac{h}{m}+\frac{n}{m}\frac{\frac{n}{m}}{2}n^{-2}+\frac{n^2}{m}\frac{\frac{n^2}{m}}{2}n^{-4}+\cdots+\frac{n^{k-1}}{m}\frac{\frac{n^{k-1}}{m}}{2}n^{2-2k}+\frac{n}{m^2}+\frac{n^k}{m}\frac{\frac{n^k}{m}}{2}n^{-2k}\geq \\
\geq \; & \frac{1}{2m^2}+\frac{1}{2m^2}+\cdots+\frac{1}{2m^2}+\frac{n}{m^2}+\frac{1}{2m^2}= \\
= \; & \frac{k}{2m^2}+\frac{n}{m^2}= \\
= \; & \frac{k+2n}{2m^2}.
\end{aligned}$$

A better way (off-line) of assigning those jobs is assigning the first $h+1$ jobs (the first of length h^2 and the subsequent h of length 1) to machine 1, and each of the groups in an empty machine. This is possible because $k \leq m-1$. Its cost -which is obviously greater than or equal to the optimal cost- is

$$\begin{aligned}
& h^2+\sum_{i=1}^{h}(h^2+i)+\sum_{i=1}^{n} in^{-2}+\cdots+\sum_{i=1}^{n^{k-1}} in^{2-2k}+\sum_{i=1}^{n^k} in^{-2k}= \\
& h^2+h^3+\frac{h(h+1)}{2}+\frac{n+1}{2n}+\cdots+\frac{n^{k-1}+1}{2n^{k-1}}+\frac{n^k+1}{2n^k}\leq \\
& 4h^3+\frac{2n}{2n}+\cdots+\frac{2n^{k-1}}{2n^{k-1}}+\frac{2n^k}{2n^k}=4h^3+k-1.
\end{aligned}$$

Thus, a c-competitive algorithm will have

$$c \geq \frac{\frac{k+2n}{2m^2}}{4h^3+k-1}=\frac{k+2n}{2m^2(4h^3+k-1)}$$

which tends to infinity when n tends to infinity. □

4.2.3 Scheduling On Execution, $m \geq 2$

The following proposition is a consequence of Lemma 4.0.0.15.

Proposition 4.2.3.1 *Under* Scheduling On Execution *paradigm, for Total Completion Time cost function, $w = 1$ and $m \geq 2$, an on-line algorithm that schedules each job to the first machine that gets idle (at the time the machine gets idle) is* 1*-competitive.*

Theorem 4.2.3.2 *No competitive on-line algorithm can be obtained the under* Scheduling On Execution *paradigm, with respect to the Total Completion Time cost function for $w \geq 2$ and $m \geq 2$.*

Proof: To prove this result, we will show an instance that makes any on-line algorithm fail to be competitive.

Let the instance be composed of 2 non-empty threads and $w-2$ empty threads. Each of the non-empty threads starts with a job of length 1. The first thread to be served has no more jobs. The second thread to be served goes on with $m-2$ jobs of length 1 and nm jobs of length ϵ.

After serving the job of the first non-empty thread in one machine, an on-line algorithm cannot do better than serving the first the $m-1$ jobs of the other non-empty thread in the remaining $m-1$ machines. Afterwards, the best the on-line algorithm can do is distribute the nm jobs of length ϵ evenly in the m machines, in which case n of them are scheduled to each machine.

In this way, the cost of the on-line algorithm is:

$$m + m\sum_{i=1}^{n}(1+i\epsilon) = m(n+1) + m\frac{n(n+1)}{2}\epsilon$$

On the other hand, the optimal algorithm (off-line), serves first the thread that contains the jobs of length ϵ and then the other non-empty thread. Its cost (assuming ϵ small enough) is:

$$(m-1) + \sum_{i=1}^{nm} i\epsilon + (nm\epsilon + 1) = m + \frac{nm(nm+1)}{2}\epsilon + nm\epsilon = m + \frac{nm(nm+3)}{2}\epsilon$$

So, a lower bound for the competitiveness of an on-line algorithm is:

$$\frac{m(n+1) + m\frac{n(n+1)}{2}\epsilon}{m + \frac{nm(nm+3)}{2}\epsilon} = \frac{2(n+1) + n(n+1)\epsilon}{2 + n(nm+3)\epsilon}$$

which tends to infinity when $\epsilon = o(n^{-2})$ and n tends to infinity. $\square$

4.3 Start-End

All the preceding theorems of this chapter assume that the schedules obtained do not need to satisfy any precedence constraint. Some of these results can be "extended" without much work for the cases were *Extended Partial Order* or *Start-Start* constraints are present.

If we restrict the problem with the *Start-End* constraint, a completely new scenario arises. The following theorem regards the case in which we have at least one machine for each thread.

Theorem 4.3.0.3 *Under any of the three main paradigms constrained with* Start-End*, if $w \leq m$ then an algorithm that for each thread chooses one machine and assigns its jobs to that machine at the first time they can be assigned is* 1*-competitive with respect to Makespan and Total Completion Time cost functions.*

Proof: It suffices to note that the off-line algorithm cannot process any of the jobs at an earlier time than that used by the presented on-line algorithm. $\square$

The same problem for the case where the number of machines is less than the number of threads is more complex. It is left as an open problem.

Chapter 5

Conclusions and Open Problems

In this work, we have first presented a classification for the different scenarios of the *On-line Multi-threaded Scheduling* problem. The different parameters that characterize the model have been described. For each combination of the different values for those parameters a possible new scenario arises.

Even though we have analyzed the most appealing scenarios of the classification, there are many contexts we have not, leaving a wide field for future research.

We have analyzed the precedence constraints and found close relations among them and between them and the on-line paradigms. Then, we turned towards the possibility of waiting and concluded that waiting does not present any benefit in the contexts studied.

Our main contributions beyond the classification are the results of competitiveness achieved with respect to the Total Completion Time cost function under *Scheduling Before Executing* and *Scheduling On Execution* paradigms. We have found a strongly competitive algorithm for the single machine case and that no competitive algorithm can be obtained if more than one thread have to be processed in more than one machine.

A summary of the results regarding the competitive analysis done with respect to Makespan and Total Completion cost functions is presented in Tables 5.1 and 5.3 respectively. For the Makespan cost function, results related to *Scheduling Jobs One By One* and *Scheduling Before Executing* paradigms are presented together,

Table 5.1: Makespan

	SJOBO and SBE		**SOE**	
	l.b.	u.b.	l.b.	u.b.
$w \geq m$			$2 - 1/m$	$2 - 1/m$
$w < m$	1.8520	1.9230	$2 - 1/m - 1/w$	$2 - 1/m$

Table 5.2: SJOBO: *Scheduling Jobs One By One*; SBE: *Scheduling Before Executing*; SOE: *Scheduling On Execution.*

Table 5.3: Total Completion Time

	SJOBO		**SBE**		**SOE**	
	l.b.	u.b.	l.b.	u.b.	l.b.	u.b.
any $w, m = 1$	?	?	w	w	w	w
$w = 1$ and $m \geq 2$	?	?	∞	$-$	$-$	1
$w \geq 2$ and $m \geq 2$	?	?	∞	$-$	∞	$-$

Table 5.4: SJOBO: *Scheduling Jobs One By One*; SBE: *Scheduling Before Executing*; SOE: *Scheduling On Execution.*

consequently with Proposition 4.1.0.18.

From these tables arise the main open problems. Firstly, it would be interesting to reduce the gap between lower bounds for our problem under *Scheduling Jobs One By One* and *Scheduling Before Executing* paradigms with respect to the Makespan objective. This is an ancient problem, as it is the same problem to the non-threaded one.

Other obviously emerging open problems regards the cases under *Scheduling Jobs One By One* paradigm with respect to the Total Completion cost function for which we have no results at all.

Bibliography

[1] Susanne Albers. Better bounds for online scheduling. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 130–139, El Paso, Texas, 4–6 May 1997.

[2] H. Alborzi, E. Torng, P. Uthaisombut, and S. Wagner. The k-client problem. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 73–82, New Orleans, Louisiana, 5–7 January 1997.

[3] N.S. Arora, R.D. Blumofe, and C.G. Plaxton. Thread scheduling for multiprogrammed multiprocessors, 1998. Manuscript.

[4] Y. Azar and O. Regev. On-line bin-streching, 1997. Manuscript.

[5] Y. Bartal, H. Karloff, and Y. Rabani. A new lower bound for m-machine scheduling, 1994. Inf. Process Lett., 50:113-116.

[6] Yair Bartal, Amos Fiat, Howard Karloff, and Rakesh Vohra. New algorithms for an ancient scheduling problem. *Journal of Computer and System Sciences*, 51(3):359–366, December 1995.

[7] Yair Bartal, Stefano Leonardi, Alberto Marchetti-Spaccamela, Jiří Sgall, and Leen Stougie. Multiprocessor scheduling with rejection. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 95–103, New York/Philadelphia, 28–30*January 1996. ACM/SIAM.

[8] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Widgerson. On the power of randomization in online algorithms. Technical Report TR-90-023, ICSI, June 1990.

[9] Allan Borodin and Ran El-Yaniv. On randomization in online computation (extended abstract). In *Proceedings, Twelfth Annual IEEE Conference on*

Computational Complexity, pages 226–238, Ulm, Germany, 24–27 June 1997. IEEE Computer Society Press.

[10] E. Feuerstein. *On-line Paging of Structured Data and Multi-threaded Paging.* PhD thesis, Università degli Studi di Roma "La Sapienza", 1995.

[11] E. Feuerstein, M. Mydlarz, and L. Stougie. Multi-threaded scheduling, 1998. Manuscript.

[12] E. Feuerstein and A. Strejilevich de Loma. On multi-threaded paging. In *Proc. Seventh Annual International Symposium on Algorithms and Computation (ISAAC'96)*, volume 1178 of *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag, 1996.

[13] E. Feuerstein and A. Strejilevich de Loma. Different competitive measures for infinite multi-threaded paging. In *Proc. Workshop on On-line Algorithms (OLA'98)*, 1998.

[14] Michel X. Goemans. On-line algorithms, 1994. 18.415/6854 Advanced Algorithms.

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

[16] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(3):416–429, March 1969.

[17] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 214–223, Los Alamitos, October 1995. IEEE Computer Society Press.

[18] David Karger, Cliff Stein, and Joel Wein. *Scheduling Algorithms.* CRC Handbook on Algorithms. 1997.

[19] R. Karp. On-line algorithms versus off-line algorithms: how much is it worth to know the future? Technical Report TR-92-044, ICSI, July 1992.

[20] H. Kellerer, V. Kotov, M.G. Speranza, and Z. Tuza. Semi on-line algorithms for the partition problem, 1996. Oper. Res. Lett.

[21] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for on-line problems. In *Proc. of 20th ACM Symposium on Theory of Computing*, pages 322–333, 1988.

[22] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

[23] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[24] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 140–149, El Paso, Texas, 4–6 May 1997.

[25] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. *IBM Journal of Research and Development*, 38(6):683–707, November 1994.

[26] Dario G. Robak. Nuevas cotas para el problema de paginacion on-line de multiples procesos. Tesis de licenciatura, Universidad de Buenos Aires, Departamento de Computación, December 1997.

[27] S. S. Seiden. Randomized online multi-threaded paging. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, Jul 1998. To appear.

[28] J. Sgall. On-line scheduling, 1997. Manuscript.

[29] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of ACM*, 28:202–208, 1985.