Εθνικο Μετσοβιο Πολυτεχνειο
Τμημα Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ ΚΑΙ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Προσεγγιστικοί Αλγόριθμοι Δρομολόγησης MapReduce Εργασιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Βασίλειου-Ορέστη Κ.
Παπαδιγενόπουλου

Επιβλέπων: Δημήτρης Φωτάκης
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2015

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ ΚΑΙ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Προσεγγιστικοί Αλγόριθμοι Δρομολόγησης MapReduce Εργασιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Βασίλειου-Ορέστη Κ. Παπαδιγενόπουλου

Επιβλέπων: Δημήτρης Φωτάκης
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 9 Ιουλίου 2015.

....	....	....
Δημήτρης Φωτάκης	Γεώργιος Γκούμας	Ιωάννης Μήλης
Επίκουρος Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π	Καθηγητής Ο.Π.Α.

Αθήνα, Ιούλιος 2015

....
Βασίλειος-Ορέστης Κ. Παπαδιγενόπουλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή της εργασίας αυτής, κ. Δημήτρη Φωτάκη, για τη συνεχή καθοδήγηση, την υπομονή και τους προβληματισμούς που μου έθεσε σε όλη τη διάρκεια αυτής της χρονιάς. Οι συμβουλές του ήταν καθοριστικής σημασίας τόσο για την εκπόνηση της εργασίας αυτής όσο και για τη συνέχιση της ακαδημαϊκής μου πορείας.

Θα ήθελα να ευχαριστήσω ιδιαιτέρως τον κ. Γεώργιο Γκούμα για το ενδιαφέρον του και για την ευγενική παραχώρηση των μηχανημάτων του Εργαστηρίου Υπολογιστικών Συστημάτων για την εκτέλεση των πειραμάτων της εργασίας αυτής.

Θα ήθελα να ευχαριστήσω ακόμα τον κ. Ιωάννη Μήλη που διατέλεσε μέλος της τριμελούς επιτροπής μου.

Χρωστάω ένα μεγάλο ευχαριστώ στην οικογένεια και στους φίλους μου, οι οποίοι μετέτρεψαν όλα αυτά τα χρόνια διαβάσματος, εργασιών και εξετάσεων μερικά από τα καλύτερα χρόνια της μέχρι τώρα ζωής μου.

Τέλος, θα ήθελα να ευχαριστήσω τα αναγνωστήρια και τις ήσυχες καφετέριες του κέντρου για την συμβολή τους στη συγγραφή του κειμένου αυτού.

Περίληψη

Το MapReduce αποτελεί ένα προγραμματιστικό μοντέλο καθώς και μια σχετική υλοποίηση για την παράλληλη επεξεργασία μεγάλων ποσοτήτων δεδομένων σε συστάδες πολυπύρηνων υπολογιστικών συστημάτων. Στο MapReduce, κάθε εργασία περιλαμβάνει δυο σύνολα υποεργασιών, τις map και τις reduce. Ενώ οι υποεργασίες κάθε συνόλου μπορούν να εκτελεστούν παράλληλα, τα δύο αυτά σύνολα οφείλουν να εκτελεστούν ακολουθιακά. Με άλλα λόγια, κάθε reduce υποεργασία μπορεί να ξεκινήσει την εκτέλεσή της μόνο μετά την ολοκλήρωση των αντίστοιχων map υποεργασιών. Ο δρομολογητής MapReduce εργασιών παίζει ένα πολύ ουσιαστικό ρόλο στην απόδοση του όλου συστήματος. Στη διπλωματική αυτή, μελετάμε το πρόβλημα της δρομολόγησης MapReduce εργασιών από τη σκοπιά της θεωρίας δρομολόγησης και των προσεγγιστικών αλγορίθμων. Παρουσιάζουμε μερικά γνωστά αποτελέσματα σχετικά με τη δρομολόγηση εργασιών σε παράλληλες μηχανές καθώς και με τη δρομολόγηση MapReduce εργασιών. Η βασική συνεισφορά της εργασίας αυτής είναι η πειραματική αξιολόγηση ενός πολυωνυμικού χρόνου 54-προσεγγιστικού αλγορίθμου για το πρόβλημα δρομολόγησης MapReduce εργασιών σε ασυσχέτιστες μηχανές με στόχο την ελαχιστοποίηση του βεβαρημένου μέσου χρόνου ολοκλήρωσης. Στη μελέτη αυτή δείχνουμε ότι σε πειραματικές συνθήκες ο ¨εμπειρικός' λόγος προσέγγισης αποδεικνύεται πολύ καλύτερος από αυτόν που έχει δειχθεί θεωρητικά για διαφορετικές κατανομές χρόνων εκτέλεσης και αριθμούς εργασιών. Επιπλέον, προτείνουμε ένα γρήγορο και ¨άπληστο' αλγόριθμο για το ίδιο πρόβλημα ο οποίος αποδεικνύεται αρκετά ανταγωνιστικός για συγκεκριμένες εισόδους. Τέλος, περιλαμβάνουμε στα πειράματά μας την μοντελοποίηση της μεταφοράς δεδομένων από map σε reduce υποεργασίες μέσω της εισαγωγής των 'shuffle' υποεργασιών.

Λέξεις-Κλειδιά: Χρονοδρομολόγηση και οργάνωση πόρων, Προσεγγιστικοί αλγόριθμοι, Αλγοριθμική ανάλυση, Map-Reduce

Abstract

MapReduce is a programming model and an associated implementation for the parallel processing of data sets in large clusters. In MapReduce, each job is associated with two sets of tasks, the map and the reduce. While the tasks of each set can be scheduled in parallel, the two sets must be scheduled sequentially i.e. the reduce after the map tasks. The centralized scheduler of MapReduce plays a critical role in the performance of the system. In this thesis, we study the problem of scheduling MapReduce tasks from the view of scheduling theory and approximation algorithms. We present several known results concerning the problem of scheduling parallel machines as well as the problem of scheduling MapReduce tasks itself. The main contribution of this thesis is the experimental evaluation of a polynomial time 54-approximation algorithm for the non-preemptive scheduling of MapReduce tasks on unrelated machines with the objective of minimizing the total weighted completion time. In this study, we show that the empirical approximation ratio of this algorithm is much better than the theoretically proven guarantee on various processing time distributions and number of jobs. We also propose a fast, greedy heuristic for the same problem that appears to be very competitive for certain inputs. Finally, we include in our experiments the modelling of data transmission between map and reduce tasks, via the introduction of "shuffle" tasks.

Keywords: Scheduling and resource allocation, Approximation algorithms, Algorithm analysis, Map-Reduce

Contents

List of Figures

Introduction

Massive Parallelism

According to the well-celebrated Moore's Law: "The number of transistors on integrated circuits doubles approximately every two years". So far so good but, what does this self-fulfilling prophecy practically mean? The more transistors one can "fit" on a constant size silicon, the more complex logic he can implement. In the terminology of computer architecture, complex logic can be translated into faster clock speeds, larger caches, better branch predictors, deeper instruction level pipelines and out-of-order execution units. The "best-effort" parallel execution of serial machine code, the so-called *Instruction Level Parallelism*, combined with faster clock speeds were some of the basic features that caused our experience as computer users to be improved year after year. The same piece of code used to perform faster and faster as the logic on integrated circuits was becoming more complex. Unfortunately, at the beginning of the 21st century as Herb Sutter stated "the free lunch is over". What happened is that logic became very complex and the number of transistors could not improve the serial execution with respect to the power consumption constraints. This turning point has been called the "ILP wall" and we hit it around 2004. In order to exceed this barrier, the wide use of multiprocessors was at that point a demand.

Although multiprocessing systems were introduced into personal computers quite recently, the idea of massive parallelism as well as the use of multiprocessor and multicomputer systems is not novel. In fact, supercomputers have been used for decades mainly for scientific based applications. Some examples of these "number crunching" applications are engineering simulations, DNA analysis and high-frequency trading algorithms. The first multicore supercomputers were introduced around 1960 with less than ten processors while at the end of the 20th century this number exceeded thousand. Nowadays, according to top500.org, Tianhe-2, a supercomputer developed by China's National University of Defense Technology counts a total of 3,120,000 cores and a total memory of 1,375 TiB.

Due to the sudden growth in the market of parallel computers, the demand for developing fast and reliable parallel code by software companies is increasing day after day. However, the development of parallel code is a process far from trivial. In fact, the traditional way somebody develops an algorithm, study programming or even think of a solution is innately serial. Therefore, writing parallel code using the primitive tools provided by an operating system is a time-consuming and error-prone activity. To deal with these difficulties it was crucial to separate the transformation of an algorithm into lines of code and its parallelization. For this reason, various high-level libraries and assisting runtime systems have been created in order to simplify the parallel code production and release the programmer from the burden of orchestrating the coordination of multiple tasks. Typical examples of these systems are OpenMP, Java Threads, Cilk, Intel Thread Building Blocks and different versions of the Message Passing Interface (MPI) paradigm.

The MapReduce Paradigm

Due to the popularity of internet technology, the amount of data stored on the web is huge. For this reason, given the competition, companies that are active in the online market such as search engines, e-mail providers, social networking services etc. need to process everyday tons of raw data. However, given the amount of information, even the execution of a conceptually trivial functionality may demand hundreds or even thousands of processing units in order to complete in a reasonable amount of time. An important parallel programming model and an associated implementation that can be used to perform this kind of operations is MapReduce. The term MapReduce originally referred to the way Google [1] processed large data sets on its clusters. Nowadays, MapReduce is considered a parallel computation paradigm with various existing implementations such as *Google's MapReduce* [1], *Apache Hadoop* [2, 3], *Disco Project* [4] and *Infinispan* [5].

The MapReduce paradigm is based on a simple idea. The input data is considered as a stream of records consisting of *key-value pairs*. The execution of a MapReduce job is divided into two basic phases: the *map* and the *reduce* phase. In the map phase, a group of *map tasks* is executed on *map machines*. Every map task, takes as input a subset of the input's key-value pairs and execute on them a programmer defined *map function*, producing as an output new key-value pairs. After the execution of all map tasks and therefore the completion of the map phase, the reduce phase begins. In the same way, in the reduce phase, a set of *reduce tasks* is executed on *reduce processors*. These tasks take as input the key-value pairs generated in the map phase and apply to them a *reduce function*, under the constraint that all records with the same key will be processed together on the same reduce machine. During the execution of a MapReduce job in the aforementioned way, the map tasks and the reduce tasks are executed in parallel on the corresponding machines but no reduce task can start its execution unless all map tasks have finished their work. Between the map and reduce phases, the redistribution of the produced key-value pairs among processors takes place in order to satisfy the reduce key-constraints. This exchange of data is called *shuffle phase*.

The simplicity of the described MapReduce model makes it more than suitable for the parallel programming of various operations on large data sets. Typical examples are distributed sorting, construction of reversed web-link graphs, distributed pattern matching etc. While it is the programmer's responsibility to appropriately define the map and reduce functions, the role of MapReduce implementation is crucial for the parallelization and execution of those functions on raw data. Although the MapReduce paradigm is an inherently distributed protocol, the coordination of the whole execution is *centralized*. Characteristics of the MapReduce such as efficient parallelization, scheduling, fault tolerance, network bandwidth, data locality and machine availability are the responsibilities of a single node called the *master node*. These operations that the MapReduce implementation hides from the programmer are certainly the core of its power.

The MapReduce Scheduling Problem

As we have mentioned before, the scheduling of MapReduce jobs is a centralized activity motivating the study of new scheduling problems. A MapReduce job, in terms of scheduling theory, consists of two sets of tasks, the map and the reduce ones. The tasks of each set can be executed in parallel on any available corresponding processor in an arbitrary order. However, in the MapReduce scheduling problem, there is one crucial constraint one must keep in mind. The execution of any reduce task of a job cannot start before the completion of all map tasks of the same job. This precedence rule, emerging directly from the nature of MapReduce, models the situation where key-value pairs are transmitted from map to reduce tasks. This general model combined with more specific details such as the type of machines or the exact metric to be optimized defines new scheduling problems. Given the fact that the performance of the centralized scheduler of a MapReduce system is crucial for the efficient exploitation of the inherent parallelism, recent studies deal with this issue from both a practical as well as a theoretical viewpoint.

From the practical point of view, there has been a great deal of empirical work [6–9] comparing the performance of various heuristics for the MapReduce scheduling problem. This work demonstrates the advantages or trade-offs of different scheduling policies under various objective functions. Typical examples are the work of Yoo and Sim [10]: "A Comparative Review for Job Scheduling for MapReduce" and the work of Wolf et al. "FLEX: a slot allocation scheduling optimizer for MapReduce workloads" [11]. In the latter, Wolf et al. formalize the problem of slot allocation by the Hadoop Fair scheduler and present various heuristic allocation schemes. In these papers, there exist no theoretically proven guarantees of the heuristics' performance in the language of scheduling theory.

There have been various theoretical approaches concerning MapReduce from its analysis as a computational model [12–14], studying its power and limitations, to the development of MapReduce algorithms for well-known problems [15–20]. As far as the MapReduce scheduling problem is concerned, Chang et al. [21] proposed a simple model, equivalent to the well-known *concurrent open shop* problem [22], where there are no dependencies between map and reduce tasks and the assignment of tasks to processors is given. In this direction, Chen et al. [23] generalized the last model by considering dependencies between tasks and presented a LP-based 8-approximation algorithm for this problem. In the same work, they involved in their model the existence of data shuffle by presenting a 58-approximation algorithm for this extension.

In their recent work, "On scheduling in map-reduce and flow-shops", Moseley et al [24] deal with the MapReduce scheduling problem following a different direction. They associated the MapReduce problem with the so-called *Flexible Flow Shop* problem [25, 26]. In the *FFS* scheduling problem, there is a set of jobs, each job has an arbitrary number of tasks and each task corresponds to a *stage*. Although, tasks can be scheduled on any available processor, they also have to be executed in their strict stage order. In other words, a task corresponding to the second stage cannot start before the execution of all tasks of the first stage of the same job. Given this new setting, Moseley et al. propose algorithms for different variants of the problem, considering offline and online scheduling cases as well as identical and unrelated processors. In all cases, the objective function of the scheduling problem is to minimize the average completion time of all jobs.

In their work "Scheduling MapReduce Jobs and Data Shuffle on Unrelated Processors", Fotakis et al. [27] generalized the last model by considering the objective of weighted average completion time. More specifically, in this model each job has multiple tasks for the map and reduce stage, the assignment of tasks to processors is flexible, there are precedence constraints between map and reduce tasks of the same job and the processing time of each task depends on the processor where it is assigned to in order to capture issues of data locality. Based on this model, they proposed a LP-based 54-approximation algorithm, which becomes an 81-approximation considering the significant cost of data communication, via the modeling of shuffle tasks.

Contribution of this Thesis

The main contribution of this thesis, is the experimental evaluation of the algorithm of Fotakis et al. and the estimation of its empirical approximation ratio under various types of input. More specifically, we show that for realistic inputs, the performance of this algorithm is much better than the theoretically proven 54-approximation. In fact, for every type of input the empirical approximation ratio lay within the interval $[1.5, 3.5]$.

In order to study the performance of the algorithm on normal inputs, we implement the algorithm and compare the objective values of the produced schedules with a LP-based lower bound to the optimal schedule. We test the performance of this algorithm for two different processing time distributions. In the first case, the processing times of tasks on machines are uniformly distributed, while in the second case, there is a strong correlation among the processing time of a task, the job that this task belongs to and the machine it runs on. More specifically, in the second case, the average processing time of the tasks of a job depends on the job, while at the same time some processors has more capabilities than others for all tasks. Moreover, we experiment on the influence of the existence of the shuffle phase and how the algorithm performs in this case. This is achieved by the introduction to our experiments of the shuffle tasks, in order to model the data transmission time from a map to a reduce processor.

In addition to this, we propose a fast, greedy heuristic for the MapReduce problem, released from the overhead of LP solving. We test the performance of this heuristic in comparison with the algorithm of Fotakis et al. on the same inputs. Based on the experimental results, we try to explain intuitively the reasons behind the performance of the two algorithms under various processing time distributions and number of jobs.

To the side of this experimental evaluation, we present in this thesis a complete analysis of the algorithm of Fotakis et al. as well as other known results for the MapReduce problem. Lastly, we present various well-known results and algorithms from the theory of *parallel machine scheduling* and *shop scheduling.*

Outline of this Reading

In **Chapter 1**, we present some fundamental definitions and terminology concerning scheduling problems. In **Chapter 2**, we present some classic results of the parallel scheduling literature with varying types of machines and metrics to be optimized. In **Chapter 3**, examine the application of precedence constraints on our problems and present some algorithms for "job shop" scheduling. In **Chapter 4**, we survey some known results concerning the exact problem of scheduling MapReduce Jobs while in

Chapter 5 we present a constant approximation algorithm for so-far most general version of the same problem. Lastly, in **Chapter 6** we present our experimental evaluation of this latter algorithm comparing it with a lower bound on the same problem as well as with a fast heuristic.

Chapter 1

Preliminaries

In this chapter we examine some basic definitions and notation concerning the theory of scheduling parallel machines. We discuss a general scheduling problem formulation, different types of machines and various metrics that may be optimized in a scheduling problem. Moreover, we consider problems with precedence constraints and we present a standard notation for describing many scheduling problems. We conclude this chapter with a reference to some complexity issues, an introduction to the notion and use of approximation analysis and a clarification of the difference between online and offline algorithms.

1.1 Definitions

How can we formally define a scheduling problem? To begin, although the relevant literature is vast, we will try to present a general model for the majority of problems. Imagine we have a set $\mathcal{M}$ of *machines* or *processors* and a set $\mathcal{J}$ of *jobs* that need to be processed. In this text, we are going to use the terms "machines" and "processors" interchangeably. Of course, the terms "machines" and "jobs" depend on the problem's context and may refer to any type of available resources and tasks to be completed. To the rest of this text, unless otherwise noted, let $m = |\mathcal{M}|$ and $n = |\mathcal{J}|$, the number of machines and jobs respectively.

Each job is associated with a specific *processing time* which, in the general case, depends on the machine which it is assigned on. We denote $p_{i,j}$ the processing time of a job j when it is executed by the machine i. In other words, we can say that each job has a vector of processing times consisting of one element for each machine it can be scheduled on. It is clear that the processing time generally refers to the time domain. Throughout this text we measure the magnitude of processing time by *units of time*, while in the relevant literature it can be found as cycles, seconds etc.

Many scheduling problems include the notion of *release date* or *release time* of a job. We say that a job j has a release time r_j if it is not available for scheduling before time r_j. Likewise, we can define a *due date* or *deadline* d_j for a job j to denote that j has to be completed before time d_j. The key difference between the notions of due date and deadline is that in the former case, a job is allowed to be completed after time d_j with a specific penalty included, while in the latter case a job has to be completed before d_j in order for a schedule to be valid.

Another crucial characteristic of a scheduling problem is the notion of *preemption*. We say that a schedule is *preemptive* if there is at least one point in time where the execution of a job is stopped before its completion and continued in a later point in time. In the majority of problems discussed in this reading preemptions are not allowed. In this case we refer to the resulting schedule as *non-preemptive*. Likewise, we denote a schedule to be *migratory* (resp. *non-migratory*) if a job is allowed (resp. not allowed) to "migrate" to another machine during its execution.

1.2 Machine Environment

In the previous section, we defined $p_{i,j}$ to be the processing time of a job j if it is executed on the machine i. A very interesting question is, fixing a job j, what can one say about the distribution of its processing time over machines.

There are three basic types of machines that have been studied extensively:

- **Identical Machines**. In this case, all machines has the same capabilities. In other words, the processing time $p_{i,j}$ of a job j stays the same for every machine i that the job can be scheduled on. In this case, we can simplify the notation and denote by p_j the processing time of a job.
- **Uniform Machines**. This is the case where each machine is associated with a specific *speed* that is independent from the job it is executed on it. Again, we can simplify the notation by defining p_j the processing time of a job on a 1-speed machine and by fixing a vector of machines' processing speeds. For instance, a job j with $p_j = 6$ needs six units of time on a 1-speed machine and two units of time on a 3-speed machine.
- **Unrelated Machines**. In this type of machines, the processing time of a job depends completely on the machine it is scheduled on. In this case, the demand for a vector of processing times for each job is exigent, as the processing times are not correlated in any simple way. For example, imagine we have two machines and two jobs: one possible scenario of processing times is $p_{1,1} = 2$, $p_{2,1} = 3$, $p_{1,2} = 1$ and $p_{2,2} = 1$.

A quite straight-forward remark concerning the types of machines is the fact that each type generalizes all its previous. It is easy to see that unrelated machines can easily simulate uniform and identical machines and that uniform machines can simulate identical if we set all speeds to be equal to one. A very useful corollary of this remark is that every algorithm that works for a specific type of machines is definitely working for all its previous.

Throughout this reading, we will concentrate our attention to the unrelated and identical case, leaving aside the uniform machines case.

1.3 Objective Functions and Metrics

Before we discuss the various types of objective functions of different scheduling problems we would like to make a clarification of some terms. There are two main aspects one must always have in mind when designing a scheduling algorithm: *feasibility* and *optimality.*

We call a schedule *feasible* if it does not violate any constraints of the problem. For instance, in a typical problem, a feasible schedule must not exclude any job from scheduling and at the same time respect all the precedence constraints, release dates or deadlines. If a specific schedule does not respect all the constraints we call it *infeasible.* Note that the notion of feasibility extends to be a characteristic of the problem itself: we call a problem instance infeasible if there is no algorithm that can create a feasible solution. A quick example to illustrate the infeasibility of a problem is the following: consider a scheduling problem of jobs with release dates and deadlines on identical machines. Imagine there is a job j with $p_j = 3$, $r_j = 1$ and $d_j = 2$. In this case, it is obvious that no algorithm can schedule this job in order to meet its deadline.

While the feasibility issue of a scheduling problem is in most cases trivial, the main concern of an algorithm designer is that of optimality. What makes a schedule optimal or, in other words, given two schedules for a problem, which one is better? In order to compare schedules we need to define first what is called in the field of mathematical optimization an *objective (cost) function* or *metric.*
For each scheduling problem we define a function $f : S \mapsto \mathbb{R}$, where S is the set of all possible schedules and $\mathbb{R}$ the set of real numbers. We say that a schedule $s' \in S$ *minimizes* f if $f(s') \leq f(s)\ \forall s, s' \in S$ and *maximizes* f if $f(s') \geq f(s)\ \forall s, s' \in S$. It follows that given a scheduling setting and an objective function f, we denote as *minimization problem* (resp. *maximization problem*) the search for a schedule that minimizes (resp. maximizes) f. This schedule is called *optimal* and the value of f for this schedule is called *optimal value.*

Now we are ready to discuss some fundamental objective functions that are most studied in literature:

- **Makespan** or **Length**. Given a schedule on parallel machines, we denote as makespan the finishing time of the job that finishes last.
- **Average Completion Time**. This metric refers to the average completion time of all jobs in a schedule. Note that given that the number of jobs for a problem is constant, the optimization of the average completion time is equivalent to the optimization of the sum of the completion times of jobs. This is the reason why this metric in literature is frequently called *total completion time* or *sum of completion times.*
- **Average (Weighted) Completion Time**. This cost function is similar to the average completion time with the difference that each job is associated with a specific weight w_j which denotes its significance. In this case the completion time of a job is multiplied by its weight before the calculation of the average. Likewise, this objective function can be found as *total weighted completion time* in literature.

Of course, there is a variety of objective functions beyond the aforementioned such as *lateness*, *tardiness*, *absolute or squared deviation* and *unit penalty* for problems with due dates or *total (weighted) flow time* for problems with release dates.

It is crucial to realise that different metrics imply and represent different needs. Specifically, the minimization of the makespan is usually suitable for one-user multiple-job environments where the user demands all his jobs to be completed as soon as possible. On the other hand, the minimization of the average completion time is more compatible with multi-user environments, where the notion of *fairness* is more important. Obviously, the term "weighted", when used, implies users and jobs with different significance or priority.

1.4 Precedence Constraints

Another well-studied aspect of scheduling problems is that of scheduling jobs with respect to *precedence constraints.* In a few words, we say that a job i must precede a job j, denoted by $i \succ j$, if the execution of the latter can start only after the completion of the former job. The use of scheduling models with precedence constraints emerges directly from the nature of parallel processing systems. It is a very common phenomenon for a job to produce data which is prerequisite for another job to begin execution. Usually the precedence hierarchy of jobs is depicted as a *Directed Acyclic Graph.* Here is an example:

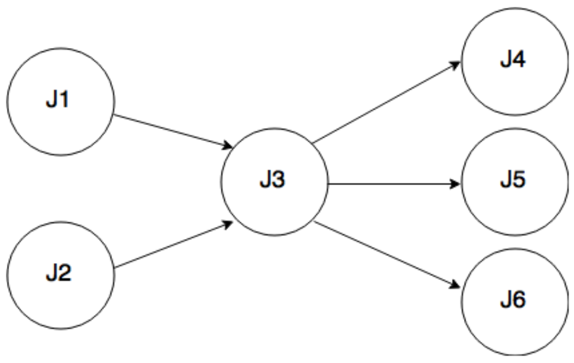

FIGURE 1.1: A graph representing the precedence constraints among jobs.

As we can see, the nodes of the graph are jobs and there is a directed edge from node i to j for each precedence constraint of the form $i \succ j$. The fact that the graph is acyclic is also obvious. Imagine there was a directed cycle of jobs, which one can be executed first? The answer of course is none, and the problem is clearly infeasible.

1.5 A Typical Notation for Scheduling Problems

Given the wide variety of scheduling problems in literature, Graham, Lawler, Lenstra and Kan introduced [33] a quick and elegant way of describing scheduling problems. Their 3-field problem classification is a set of three labels of the form $\alpha|\beta|\gamma$. Each label reflects different characteristics of a scheduling problem:

- **Field** α, describes the machine environment of the problem. Typical α values are 1, P, Q and R for the single machine, identical parallel, uniform parallel and unrelated parallel machines respectively. Moreover, another example of common notation is the use of an index that denotes the number of machines on parallel environments. For instance, P_5 represents an environment of five identical parallel machines.

- **Field** β, is the second field and reflects additional properties of a problem. Some of the most typical properties that belongs to this field are r_j for problems with release dates, d_j for due dates or deadlines, *prec* for precedence constraints or *pmtn* for problems where preemption is allowed. Another value of this field can be $p_{i,j} = 0 \; or \; 1$ denoting processing times with zero or unit value.

- **Field** γ, is the last field of the notation and represents the type of the objective function. Typical values of γ are C_{max} for makespan minimization, $\sum C_j$ for the total completion time and $\sum w_j C_j$ for the total weighted completion time.

For example, $1|prec|L_{max}$ is the problem of minimizing the maximum lateness on a single machine subject to given precedence constraints. In the same way, $R|pmtn|\sum C_j$ is the problem of minimizing the total completion time on unrelated machines when preemption is allowed.

1.6 Complexity Issues

At this point, we would like to discuss the complexity characteristics of different scheduling problems. Unfortunately, given the jungle of the studied scheduling problems, only a small percentage of those accepts a polynomial-time algorithm that estimates an optimal solution. We concentrate our attention to parallel machine scheduling. In this case, the simplest problem that accepts a polynomial-time algorithm is $P_m||\sum C_j$. Unfortunately we cannot argue the same for the problems $P_m||C_{max}$ and $P_m||\sum w_j C_j$, when $m \geq 2$.

Usually, we are able to prove that a problem does not accept a polynomial-time algorithm, unless $P \neq NP$, with the help of what is called a *reduction*. Intuitively, we say that a problem P_1 is reducible to problem P_2 if an algorithm for solving problem P_2 efficiently could also be used as a subroutine to solve P_1. An example is the following: think of the problems $P||C_{max}$ and $P|r_j|C_{max}$. Given that the first problem does not accept a polynomial time algorithm we can conclude that the same holds for the second problem. Imagine the contrary and that there is an efficient algorithm solving $P|r_j|C_{max}$; then if we take an instance of $P||C_{max}$ and add a "dummy" zero release date for every job then we would be able to solve the problem efficiently.

1.7 Approximating an Optimal Solution

As we have seen, the majority of parallel scheduling problems do not accept polynomial-time algorithms. This means that for a slightly more than "small" input, an optimal schedule cannot be estimated efficiently. So what can we do for those problems? Hopefully, the story does not end here. In many cases we can construct algorithms running in polynomial-time and producing an approximation to the optimal value.

More formally, a *ρ-approximation algorithm* for a minimization problem is an algorithm that runs in polynomial-time on the size of the input and produces a solution SOL such that:

$$OPT \leq SOL \leq \rho SOL$$

, where OPT the objective value of an optimal solution.

The value ρ is called *approximation ratio* or *guarantee* of the algorithm. Generally $\rho = f(I)$, where I a problem instance. In other words, the approximation ratio of an algorithm is a function of the problem instance. As a result, the main target in the area of approximation algorithms is to find algorithms with the lowest possible upper bounds of approximation ratios. In literature, there exist algorithms with constant, logarithmic, linear or other function of the input size approximation ratios.

1.8 Online and Offline Scheduling

In practice, there are many scheduling problems where jobs arrive over time and the scheduler does not know anything about their existence until their arrival. This is called an *online scheduler*. There are two subcategories of online scheduling depending on the existence or not of *clairvoyance*. A *clairvoyant scheduler*, learns all the relevant information for a job (processing time, weight, etc) by the time of its arrival while a *non-clairvoyant* learns nothing for a job but the fact that it has arrived. In order to clarify the difference between the two types of scheduler let us demonstrate with an example. The scheduler of an operating system is clearly an online scheduler. In the general case, this scheduler may accept and run a job that needs live interaction with the user without knowing the time the user will spend on the job or the importance of this job to the user. In this case the scheduler may be considered non-clairvoyant. If the user could inform in any way the scheduler about the time his job is going to spend or the weight-importance of this job, then this scheduler could be considered a clairvoyant one.

A standard way of measuring the effect of non-clairvoyance on an online scheduler is the *competitive ratio*. For a minimization problem we can formally define the competitive ratio of a schedule $S(I)$, produced on input I as following:

$$\rho(|I|) = \max_{I} \frac{S(I)}{A(I)}$$

, where $A(I)$ the objective value of an adversary A who can specify the input I and schedule I optimally.

Unfortunately, the notion of competitive ratio has been criticized as impractical as it usually produces "unrealistically" high ratios for usual inputs and consequently the designer or prospective user of an algorithm fails to differentiate between the performance of two online algorithms in practice.

In order to surpass the drawbacks of the competitive ratio, the notion of *resource augmentation* was introduced. Intuitively, resource augmentation suggests that if we give more resources in terms of quality or quantity to a non-clairvoyant scheduler then, the approximation ratio of this problem may eventually be bounded. Formally, we say that an online scheduler is *s-speed c-approximation* algorithm if:

$$\max_I \frac{S_S(I)}{A_1(I)} \leq c$$

, where $S_S(I)$ the produced schedule with $s > 1$ resources. Note that in these cases the approximation ratio may also be found as *competitive ratio* of an online algorithm.

Resource augmentation, gives to the user of a scheduling algorithm a practical way to balance the loss of clairvoyance by buying more or more powerful processors. A typical example of the power of resource augmentation is included in the seminal paper of Kalyanasundaram and Pruhs: "Speed is as Powerful as Clairvoyance". In their work [34], they propose a $(1+\epsilon)$-speed $f(\frac{1}{\epsilon})$-approximation algorithm for the classic uniprocessor CPU scheduling problem $1|r_i, pmtn|\sum F_i$.

Chapter 2

Scheduling Parallel Machines

The purpose of this chapter is to introduce the reader to the problem of scheduling parallel machines. For this reason, we present a collection of well-known algorithms and results on both cases of identical and unrelated machines and we discuss a variety of typical methods for the optimization of different metrics such as makespan and total (weighted) completion time.

2.1 List Scheduling and Longest Processing Time First

We begin our discussion on parallel scheduling algorithms with the simple *Graham's List Scheduling* algorithm [38]. Even though the algorithm's description and analysis may seem trivial even for the unfamiliar with the design of approximation algorithms reader, the purpose of this presentation is to present a general technique for proving approximation guarantees.

The problem we discuss is $P_m||C_{max}$, which, as we have stated in the previous chapter, is NP-hard for $m \geq 2$. The **List Scheduling** algorithm is nothing else but the following simple procedure:

List Scheduling: Take an arbitrary sequence of jobs and assign them one-by-one to the so-far least loaded machine.

Before we begin the analysis of the algorithm, we should make a significant remark. An approximation ratio is a guarantee that the objective function of the feasible solution found by an algorithm would lay within a factor of the optimal value. Since most of the problems where the demand for approximation algorithms is exigent are NP-hard, as we know from fundamental complexity, it is difficult to compute the optimal objective value. So how can we prove an approximation guarantee if we do not have a clue of what the optimal value is? As a matter of fact, many times we do have a clue. There are cases where it is an easy task to estimate a so-called *lower bound* on the optimal value. If we compare the result of an approximation algorithm with such a lower bound, then we can prove a guarantee without knowing the exact optimal value. Indeed, we are certain by definition that this value would be at least its lower bound. The proof of the following theorem about the **List Scheduling** algorithm demonstrates these ideas.

Theorem 2.1. ***List Scheduling*** *is a polynomial-time 2-approximation algorithm for the problem of makespan minimization on identical machines.*

Proof. For the problem of makespan minimization on identical machines there are two very useful lower bounds. If we denote by C^*_{max} the optimal makespan then the following inequalities hold:

$$C^*_{max} \geq \frac{1}{m} \sum_{j \in \mathcal{J}} p_j$$

and

$$C^*_{max} \geq \max_{j \in \mathcal{J}} p_j$$

It is quite straight-forward why these inequalities consist lower bounds for the makespan problem. The former suggests that the solution is greater or equal to the total processing demand divided by the number of machines, a quantity that is obviously the most "fair" load-balancing one can achieve. The latter implies that the makespan must be at least the maximum processing time of any job on a machine.

In order to prove the approximation ratio, consider a schedule produced with **List Scheduling** and define C_{max} to be its makespan. We fix a job j to be the job finishing last and, as a consequence, its completion time equals C_{max}. As we can see $C_{max} = s_j + p_j$, where s_j is the starting time of job j. By definition of our algorithm we can see that just before the time s_j, all the machines are working, producing a total amount of work ms_j, which is definitely smaller than the total processing requirement. In other words:

$$ms_j \leq \sum_{j \in \mathcal{J}} p_j$$

Consequently:

$$s_j \leq \frac{1}{m} \sum_{j \in \mathcal{J}} p_j \leq C^*_{max}$$

Therefore, for the cost of the produced schedule it follows:

$$C_{max} = s_j + p_j \leq \frac{1}{m} \sum_{j \in \mathcal{J}} p_j + \max_{j \in \mathcal{J}} p_j \leq 2C^*_{max}$$

□

Intuitively, the main drawback of the algorithm lies in the fact that a job with relatively large processing time has to be scheduled last. With the following example one can see that the analysis of this algorithm is tight: Imagine we have m machines, $m^2 - m$ unit time jobs and one job whose processing time equals m. In case the large job is at the end of the list the following problem rises: the unit time jobs would be equally distributed to the m machines giving a load of $m - 1$. As a result, when the large job is scheduled a solution with makespan $2m - 1$ is produced. If we compare this solution with the optimal $C^*_{max} = m$ we asymptotically get an approximation factor of 2.

A quite obvious optimization for the algorithm is to sort the list of jobs before the assignment in a non-increasing processing time order. This algorithm is called **Longest Processing Time First**. The following theorem holds:

Theorem 2.2. ***Longest Processing Time First*** *is a polynomial-time $\frac{3}{2}$-approximation algorithm for the problem of minimizing makespan on identical parallel machines.*

Proof. After the sorting procedure of the algorithm we can make the following observations: If we have at most m jobs then the result of the algorithm is optimal i.e. one job per machine. Now, if we have more than m jobs then, for the processing time of a job j where $j > m$ it holds:

$$p_j \leq \frac{C^*_{max}}{2}$$

It is easy to see that assuming the contrary, then it should be the case that a job scheduled last on a machine would have larger processing time than the sum of processing times of all other jobs scheduled on the same machine. This fact leads to a contradiction even for the case of two jobs per machine.

Using the last result and following the analysis of the previous theorem, for the last finishing job j it holds:

$$C_{max} = s_j + p_j \leq \frac{1}{m} \sum_{j \in \mathcal{J}} p_j + p_j \leq \frac{3}{2} C^*_{max}$$

□

Note that a more careful analysis of the **Longest Processing Time First** algorithm can yield a $\frac{4}{3}$-approximation guarantee. The proof lies on the fact that if the processing times are greater than $\frac{1}{3}C^*_{max}$ the algorithm produces an optimal schedule with at most two jobs per machine.

2.2 An Exact Algorithm for the Average Completion Time Problem on Identical Machines

In this section we present a greedy algorithm that produces an optimal solution for the problem of minimizing the total completion time on identical parallel machines. The algorithm is an extension of the well-known *Shortest Processing Time First* rule for the total completion time on a single machine.

Shortest Processing Time First: Order jobs in a non-increasing processing time order and assign them in a cyclic way to machines.

Theorem 2.3. ***Shortest Processing Time First*** *is a polynomial-time exact algorithm for the problem of minimizing total completion time on identical machines.*

Proof. Without loss of generality we assume that n is divided by m. If this is not the case, we can add to our problem instance an appropriate number of "dummy" jobs with zero processing time that clearly do not affect the resulting optimal schedule.

An alternative way one can express the resulting cost of an algorithm for this problem is the following:

$$\begin{aligned}\sum_{j\in\mathcal{J}} C_j = {} & \overbrace{\alpha_{n,1}n + \alpha_{n,2}n + \cdots + \alpha_{n,m}n}^{m \text{ times}} + \\ & + \overbrace{\alpha_{n-1,1}(n-1) + \alpha_{n-1,2}(n-1) + \cdots + \alpha_{n-1,m}(n-1)}^{m \text{ times}} + \\ & + \cdots + \overbrace{\alpha_{1,1} + \alpha_{1,2} + \cdots + \alpha_{1,m}}^{m \text{ times}}\end{aligned}$$

In the expression above $\alpha_{l,k}$ is variable denoting the processing time of a job which is scheduled on machine k and precedes $l-1$ jobs on the same machine. This is the reason why this processing time is multiplied by a factor of l, as it clearly contributes l times to the objective function, one for its own completion time and one for the completion time of each job it precedes. Given this new form of objective function, for any given schedule the objective value can be estimated by assigning every p_j to an $\alpha_{l,k}$ and by setting the unassigned $\alpha_{l,k}$ variables to zero. The problem now is to find an assignment that minimizes the objective function.

Hopefully, the Hardy-Littlewood-Pólya inequality, also known as rearrangement inequality [39], gives an elegant proof for the lower bound on any expression of this type. Specifically, we know that for every choice of real numbers $x_1 \leq x_2 \leq \cdots \leq x_n$ and

$y_1 \leq y_2 \leq \cdots \leq y_n$ and for any permutation $x_{\sigma(1)}, x_{\sigma(2)} \ldots x_{\sigma(n)}$ it is the case:

$$x_n y_1 + \cdots + x_1 y_n \leq x_{\sigma(1)} y_1 + \cdots + x_{\sigma(n)} y_n \leq x_1 y_1 + \cdots + x_n y_n$$

In a few words, the lowest possible value can be achieved if we multiply the variables in a reversed order of their indices i.e. if we multiply the greatest x_n with the lowest y_1 and so on. It is not hard to verify that the **Shortest Processing Time** algorithm achieves exactly this lower bound and as a result the produced objective value has to be optimal. □

2.3 Minimizing Total Weighted Completion Time on Identical Machines

In their seminal work: "Scheduling to Minimize Average Completion Time: Off-line and On-line Approximation Algorithms", Hall et al. present [40] a collection of approximation algorithms for classic scheduling problems. In this part, we are going to present an approximation algorithm for the $P||\sum w_j C_j$ problem. Before we describe the algorithm and the analysis, we are going to prove some useful lemmas concerning this problem.

If for any set $S \subseteq \{1 \ldots n\}$ of jobs we define $p(S) = \sum_{j \in S} p_j$ and $p^2(S) = \sum_{j \in S} p_j^2$, then the following lemma holds:

Lemma 2.4. *Let $C_1, C_2, \ldots, C_n$ the completion times of jobs in a feasible schedule for $P||\sum w_j C_j$. Then the C_j satisfy the inequalities:*

$$\sum_{j \in S} p_j C_j \geq \frac{1}{2m}(p(S)^2 + p^2(S)) \quad \textit{for each } S \subseteq N$$

Proof. Let us assume a feasible schedule where there is no unforced idle time. In this schedule the jobs are indexed without loss of generality in a way that $C_1 \leq \cdots \leq C_n$. In an induced schedule of jobs $\{1 \ldots j\}$, job j finishes last on machine i which, by definition, is the most heavily loaded machine with respect to this subset of jobs. Since there is no idle time, for the load of machine i it holds:

$$C_j \geq \frac{1}{m} p(\{1 \ldots j\}) = \frac{1}{m} \sum_{k \in \{1 \ldots j\}} p_k$$

If we multiply the last inequality with p_j, follow the same analysis for each induced subset of the form $\{1 \ldots j\}$, then by summation we get:

$$\sum_{j=1}^{n} p_j C_j \geq \frac{1}{m} \sum_{j=1}^{n} p_j \sum_{k=1}^{j} p_k$$

With usual arithmetic we can simplify the last inequality so that for $S = \{1 \dots n\}$ it holds:

$$\sum_{j \in S} p_j C_j \geq \frac{1}{2m}(p(S)^2 + p^2(S))$$

The lemma in its general case follows directly from the fact that for each possible subset of jobs we can apply the same analysis restricted to the schedule of these jobs. □

Using the previous lemma, we see that for a subset of jobs $S = \{1 \dots j\}$, where $C_k \leq C_j$ for each $k \in \{1 \dots j\}$ it is the case that:

$$C_j p(S) = C_j \sum_{k \in S} p_k \geq \sum_{k \in S} C_k p_k \geq \frac{1}{2m}(p(S)^2 + p^2(S)) \geq \frac{1}{2m} p(S)^2$$

From this analysis the following lemma immediately follows:

Lemma 2.5. *Let $C_1 \dots C_n$ with $C_1 \leq \cdots \leq C_n$ of n jobs satisfying $\sum_{j \in S} p_j C_j \geq \frac{1}{2m}(p(S)^2 + p^2(S))$. Then for each $j = 1, \dots, n$, where $S = \{1, \dots, j\}$ it is the case that:*

$$C_j \geq \frac{1}{2m} p(S)$$

Now, consider the following linear programming formulation:

$$\begin{array}{lll}
\text{Minimize:} & \displaystyle\sum_{j=1}^{n} w_j C_j & \\
\text{s.t:} & C_j \geq p_j & ,\forall j \in \{1 \dots n\} \\
& \displaystyle\sum_{j \in S} p_j C_j \geq \frac{1}{2m}(p(S)^2 + p^2(S)) & ,\forall S \subseteq \{1 \dots n\}
\end{array}$$

Given this LP formulation, the algorithm **Schedule-by-$\bar{C}_j$** works as follows: Given an optimal solution $\bar{C}_1 \dots \bar{C}_n$ to the LP, assuming without loss of generality that $\bar{C}_1 \leq \cdots \leq \bar{C}_n$, the algorithm sorts the jobs in a non-decreasing order of $\bar{C}_j$ and schedules iteratively each job j to the earliest available p_j units of time on any machine.

Theorem 2.6. *The schedule found by **Schedule-by-$\bar{C}_j$** is a $(3 - \frac{1}{m})$-approximation algorithm for the problem of minimizing total weighted completion time on parallel machines.*

Proof. If we denote as C_j the completion time of job j scheduled by **Schedule-by-$\bar{C}_j$**, then for the induced schedule of jobs $\{1 \dots j\}$ it is the case that:

$$C_j \leq s_j + p_j$$

, where s_j denotes the starting time of job j. Since by definition of the algorithm, before time s_j all machines are busy it holds $s_j \geq \frac{1}{m}p(S \setminus \{j\})$. Given this we have :

$$C_j \leq s_j + p_j \leq \frac{1}{m}p(S \setminus \{j\}) + p_j \leq \frac{1}{m}p(S) + (1 - \frac{1}{m})p_j$$

From the previous lemma and the constraints of the LP we know that $\bar{C}_j \geq p_j$ and $2\bar{C}_j \geq \frac{1}{m}p(S)$. Given that $\bar{C}_j$ is clearly a lower bound to the optimal solution it is the case:

$$C_j \leq (3 - \frac{1}{m})\bar{C}_j$$

If we apply the last inequality to the objective function the theorem follows. □

An interesting observation is that the second set of constraints of the LP formulation contains exponentially many constraints. Hopefully, it has been proven [41] that we can solve this exact LP via the Ellipsoid Method, if we use as constraints only the subsets of the form $\{1 \dots j\}$, $\forall j \in \mathcal{J}$.

2.4 Minimizing Makespan on Unrelated Machines

In this section we present a constant approximation algorithm for the problem of makespan minimization on unrelated machines. This result, apart from its significance in the field of approximation algorithms, is in fact a very useful tool for the MapReduce scheduling algorithms we examine in the next chapters.

As we can easily see, list scheduling algorithms cannot directly apply in the unrelated machine setting and definitely do not yield constant approximation guarantees. Hopefully, thanks to the work of Lenstra, Shmoys and Tardos, a 2-approximation algorithm [42] was proved using linear programming techniques.

The key tool used in this algorithm is the following *Rounding Theorem.* Let $J_i(t)$ denote the set of jobs that require no more than t units of time on machine i and $M_j(t)$ the set of machines that can process job j in no more than t units of time. Let $x_{i,j}$ a 0-1 assignment variable indicating whether job j is assigned to machine i. We consider the generalized decision version of our scheduling problem defining for each machine i a deadline d_i and restricting the schedule to include assignments from jobs to machines with processing time at most t. In this setting consider the following mathematical formulations:

$$
\begin{aligned}
&IP(P,\vec{d},t): \\
&\sum_{i\in M_j(t)} x_{ij} = 1 && ,\forall j \in \{1\dots n\} \\
&\sum_{j\in J_i(t)} p_{ij}x_{ij} \le d_i + t && ,\forall i \in \{1\dots m\} \\
&x_{ij} \in \{0,1\} && ,\forall j \in J_i(t)\,,\, i = 1\dots m
\end{aligned}
$$

$$
\begin{aligned}
&LP(P,\vec{d},t): \\
&\sum_{i\in M_j(t)} x_{ij} = 1 && ,\forall j \in \{1\dots n\} \\
&\sum_{j\in J_i(t)} p_{ij}x_{ij} \le d_i && ,\forall i \in \{1\dots m\} \\
&x_{ij} >= 0 && ,\forall j \in J_i(t)\,,\, i = 1\dots m
\end{aligned}
$$

Theorem 2.7 (Rounding Theorem)**.** *If the linear program $LP(P,\vec{d},t)$ has a feasible solution, then any vertex $\tilde{x}$ of this polytope can be rounded to a feasible solution $\bar{x}$ of the integer program $IP(P,\vec{d},t)$ and this rounding can be done in polynomial time.*

Proof. If we denote by u the number of $x_{i,j}$ variables involved for a specific t, we can see that the mathematical formulations above have $n+m+u$ constraints each. As we know from the theory of linear programming each optimal solution correspond to a vertex of the pointed polyhedron and each vertex is determined by u linearly independent rows of the constraints matrix, each one satisfied with equality. If we observe the constraints of the LP we can see that for each feasible solution at most $m+n$ variables may have non-zero value. Now, if we denote by α the number of integrally assigned jobs and by β the number of fractionally assigned ones, then by definition it holds $\alpha+\beta=n$. A job that is fractionally assigned is involved to at least two non-zero $x_{i,j}$ variables and, therefore, $\alpha+2\beta\le m+n$. Combining the two relations we can see that the number of fractionally assigned jobs is at most m.

For any feasible solution x, we define a bipartite graph $G(x)=(J,M,E)$, where J and M the sets of nodes corresponding to jobs and machines respectively and E the set of edges defined as $E=\{(i,j)|x_{i,j}>0\}$. From the previous observations we can see that $G(\tilde{x})$ has no more edges than vertices and in fact each connected component of the graph has the same property. This type of graph whose connecting components are trees or trees plus one edge is called *pseudoforest*. In order to prove this proposition, for each connected component of $G(\tilde{x})$, let C denote the set of its nodes and M_C, J_C the sets of machines and jobs involved respectively. Let $\tilde{x}_C$ the restriction of $\tilde{x}$ to variables $\tilde{x}_{i,j}$ such that $i \in M_C$ and $j \in J_C$. If we also denote by $\tilde{x}_{\bar{C}}$ the rest of the variables and reorder the columns of the variable vector, we see that $\tilde{x} = (\tilde{x}_C, \tilde{x}_{\bar{C}})$. In order for the connected component C to maintain the same property about the number of fractionally assigned jobs all we need to show is that $\tilde{x}_C$ is an extreme point of $LP(P_C, \vec{d}_C, t)$. Suppose it is not, then there should exist two points y_1 and y_2 enough close to $\tilde{x}_C$ such that $\tilde{x}_C = \frac{1}{2}(y_1 + y_2)$. In this case, it holds that $\tilde{x} = \frac{1}{2}((y_1, \tilde{x}_{\bar{C}}) + (y_2, \tilde{x}_{\bar{C}}))$, which leads to a contradiction given the fact that $\tilde{x}$ is should be an extreme point.

An example of a produced pseudoforest is shown in the next graph:

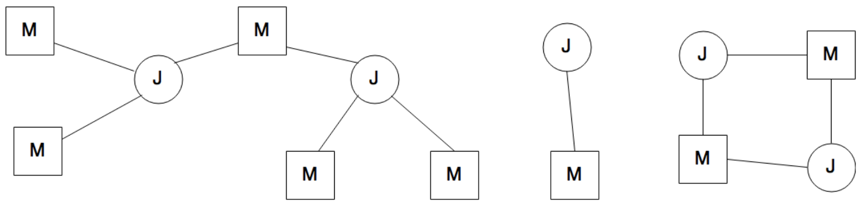

FIGURE 2.1: A Pseudoforest of Job and Machine Nodes.

The key observation is that we can use this pseudoforest to round the fractional solution of the LP to a feasible solution of the IP. This can be done quite easily: for each integrally assigned job keep this assignment to the solution. For each connected component that is a tree start from any job node and greedily assign one machine to each job. Lastly, for each connected component that contains a cycle, given the fact that our graph is a bipartite, this cycle must have an even length. Therefore, we can "break" the cycle if we arbitrarily begin from any edge and assign jobs to machines in an alternating way.

Clearly, with the aforementioned rounding one can see that if the LP has a feasible solution then $\bar{x}$ is a feasible solution to the IP such that for each machine i:

$$\sum_{j \in J_i(t)} p_{i,j}\bar{x}_{i,j} \leq \sum_{j \in J_i(t)} p_{i,j}\tilde{x}_{i,j} + \max_{j \in J_i(t)} p_{i,j} \leq d_i + t$$

Note that the rounding from a fractional to an integral solution this way can be done in polynomial time. □

Using this Rounding Theorem, we can construct a 2-approximation algorithm for the problem of minimizing makespan on unrelated machines. Before we present the algorithm, we introduce some useful definitions and theorems about ρ-relaxed decision procedures:

Definition 2.8. A *ρ-relaxed decision procedure* or *oracle* can be seen as a decision oracle with two possible outcomes: "no" and "almost". More specifically, for a minimization problem the oracle takes as input a problem instance and a target objective d. It returns either "no" or a solution with objective at most ρd. If the oracle returns "no", then there is not a solution with objective at most d.

In the following, we present a useful theorem that links the existence of a ρ-relaxed decision oracle with the existence of a ρ-approximation algorithm for the problem of makespan minimization on unrelated machines.

Lemma 2.9. *If there is a polynomial ρ-relaxed decision procedure for the minimum makespan problem on unrelated machines, then there is a polynomial ρ-approximation algorithm for this problem.*

Proof. In order to prove the lemma, we can create a ρ-approximation algorithm using binary search over the domain of possible makespan. Given a target objective d, then d is an upper bound and $\frac{d}{m}$ is clearly a lower bound to the problem's objective value. During the binary search procedure, if the oracle answers "no" for a target makespan t, then we restrict the search space to values over t. If the oracle returns a schedule with makespan at most ρt, then we keep the best schedule we have found so far and we restrict the search space to values less than t. It is easy to see that in $O(\log d)$ time, the algorithm converges to the highest value that the oracle answers "no". If we denote s this point, then the best objective value that we have kept during this procedure would be at most ρs. Given that the optimal value is definitely more than s then it is the case that the best objective value found would lay within a factor of ρ of the optimal. This procedure gives as a polynomial-time ρ-approximation algorithm for the problem of makespan minimization on unrelated machines. □

Now, the next theorem proves the existence of a 2-approximation algorithm for the problem we discuss:

Theorem 2.10. *There is a 2-approximation algorithm for the problem of makespan minimization on unrelated machines that runs in time bounded by a polynomial of the input size.*

Proof. From the lemmas we have already proved, it suffices to construct a 2-relaxed decision procedure for the problem. Let (P, d) a problem instance. If we recall the $LP(P, \vec{d}, t)$ of the rounding theorem and set $d_1 = d_2 = \cdots = d_m = t = d$ then we can see the following: if the $LP(P, \vec{d}, d)$ has not a feasible solution, then the $IP(P, \vec{d}, d)$ cannot have either. On the other hand if the $LP(P, \vec{d}, d)$ has a feasible solution, then it can be rounded to a feasible solution of the $IP(P, \vec{d}, d)$. As we have already shown in this rounded solution, the deadline of each machine can be extended for at most t units of time. Given we have set all deadlines and t equal to d, then the rounding yields a solution at most $2d$. This use of the rounding theorem clearly satisfies the definition of a 2-relaxed decision procedure. □

2.5 Minimum-Weight Bipartite Matching to Schedule Positions

In this section we examine the problem of total completion time on unrelated machines. Extending the analysis we have done for the total completion time on the identical machines case, we can reformulate the objective function in a similar manner using 0-1 assignment variables.

Let $x_{i,k,j}$ denote the variable indicating that a job j is scheduled on a machine i in the k-th last place among the jobs scheduled on the same machine. It is clear that if $x_{i,k,j} = 1$, then the job is scheduled under these conditions and the opposite. Using this notation we can reformulate the objective function as following:

$$\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{n} kp_{i,j}x_{i,k,j}$$

In the previous relation we can see that if a job is scheduled on k-th from the last job position on a machine, it contributes k times its processing time to the objective function.

The solution of the following IP clearly gives the solution to the original problem:

$$\begin{array}{lll}
\text{Minimize:} & \displaystyle\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{n} kp_{i,j}x_{i,k,j} & \\
\text{s.t:} & \displaystyle\sum_{i=1}^{m}\sum_{k=1}^{n} x_{i,k,j} = 1 \quad ,\forall j \in \{1\ldots n\} & (1) \\
& \displaystyle\sum_{j=1}^{n} x_{i,k,j} \leq 1 \quad ,\forall i \in \{1\ldots m\}, \forall k \in \{1\ldots n\} & (2) \\
& x_{i,k,j} \in \{0,1\} \quad ,\forall i \in \{1\ldots m\},\ \forall j,k \in \{1\ldots n\} & (3)
\end{array}$$

In the previous formulation constraints (1) suggest that every job must be scheduled on exactly one position on some machine while constraints (2) restrict more than one jobs to be scheduled on the same position of a machine.

The interesting fact about this formulation is that it is the exact IP formulation of the *Weighted Bipartite Matching Problem*. Consider we have a bipartite graph $G(A, B, E)$, where A is the set of nodes corresponding to the jobs and B the set of nodes corresponding to all available scheduling positions. It holds that $|A| = n$ and $|B| = nm$, as

nm is the number all possible places a job can be scheduled on. This bipartite graph is complete and the weight of each edge from a job j to the node (i, k) is defined to be $kp_{i,k,j}$. It is clear now that finding a minimum weight matching on this graph immediately yields a schedule of minimum objective. Here is an example for the problem of scheduling two jobs on two machines:

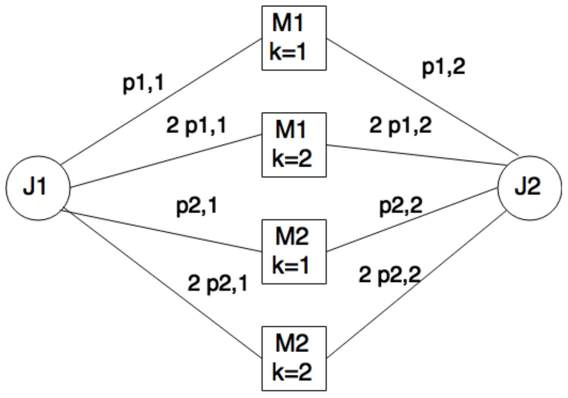

Figure 2.2: Bipartite Graph Modelling Positions.

Hopefully, it is a known result that the LP-relaxation if this IP, obtained by relaxing the values of $x_{i,k,j}$ to be non-negative, has the same feasible set of solutions with the original IP. In other words, it can be proven that the extreme points of the LP-relaxation only correspond to integral solutions. From this fact we immediately obtain a polynomial-time exact algorithm for $R||\sum C_j$. Note that a well-known combinatorial algorithm for the problem of minimum weight bipartite matching is the *Hungarian* algorithm [43].

2.6 Minimizing Total Weighted Completion Time of Tasks on Unrelated Machines

As far as the problem $R||\sum w_j C_j$ is concerned, Hall et al.[40] developed an LP-based $\frac{16}{3}$-approximation algorithm even for the presence of release dates. Instead of presenting this algorithm, we choose to discuss a harder version of this problem. The problem we present is that of scheduling *job orders* on unrelated machines under the same objective, which was proposed in the work of Correa, Skutella and Verschae: ”The Power of Preemption on Unrelated Machines” [44]. Because of the fact that this algorithm will be used as a subroutine for MapReduce scheduling algorithms in the next, we are going to use the notation from the work of Fotakis et al.: ”Scheduling MapReduce Jobs and Data Shuffle on Unrelated Processors” [27].

In this problem, we have a set $\mathcal{P}$ of processors and a set $\mathcal{J}$ of jobs. Each job j has a set $\mathcal{T}_j$ of tasks and each task $T_{k,j}$ can be executed on processor i in $p_{i,k,j}$ units of time. We denote by $\mathcal{T}$ the set of all jobs' tasks. The completion time of each job C_j is determined by the completion time of the job's task $C_{k,j}$ that finishes last i.e. $C_j = \max_{k|T_{k,j}\in\mathcal{T}}\{C_{k,j}\}$. Each job is associated with a weight w_j and our objective is to minimize the total weighted completion time.

In this algorithm, which we will refer to as **TaskScheduling** in the rest of this reading, we are going to schedule the tasks of jobs on exponentially growing time-intervals. For this reason, consider a parameter $\delta \in (0,1)$ and a maximum possible value P for the time horizon of this problem. To ensure that P is an upper bound to the completion time of each scheduling problem of this kind we define $P = \sum_{T_{k,j}\in\mathcal{T}} \max_{i\in\mathcal{P}} p_{i,k,j}$. Let L be the smallest integer such that $(1+\delta)^{L-1} > P$. We discretize the time horizon into a set of the following intervals: $\mathcal{L} = \{[1,1], (1,(1+\delta)], ((1+\delta),(1+\delta)^2], \dots, ((1+\delta)^{L-1},(1+\delta)^L]\}$. In the following, we denote by I_ℓ the time interval $((1+\delta)^{\ell-1},(1+\delta)^\ell]$. We assume without loss of generality that all processing times are positive integers. Clearly, the number of intervals is polynomial in the size of instance and in $\frac{1}{\delta}$.

We introduce a set of variables $y_{i,k,j,\ell}$ indicating that a task $T_{k,j}$ is completed on processor i within the interval I_ℓ. Using this notation we introduce the following linear programming formulation:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j\in\mathcal{J}} w_j C_j \\
\text{subject to:} \quad & \sum_{i\in\mathcal{P},\ell\in\mathcal{L}} y_{i,k,j,\ell} \geq 1, && \forall T_{k,j}\in\mathcal{T} \ (1)\\
& C_j \geq C_{k,j}, && \forall T_{k,j}\in\mathcal{T} \ (2)\\
& \sum_{i\in\mathcal{P}}\sum_{\ell\in\mathcal{L}} (1+\delta)^{\ell-1} y_{i,k,j,\ell} \leq C_{k,j}, && \forall T_{k,j}\in\mathcal{T} \ (3)\\
& \sum_{T_{k,j}\in\mathcal{T}} p_{i,k,j} \sum_{t\leq\ell} y_{i,k,j,t} \leq (1+\delta)^\ell, && \forall i\in\mathcal{P},\ell\in\mathcal{L} \ (4)\\
& p_{i,k,j} > (1+\delta)^\ell \Rightarrow y_{i,k,j,\ell} = 0, && \forall i\in\mathcal{P}, T_{k,j}\in\mathcal{T},\ell\in\mathcal{L} \ (5)\\
& y_{i,k,j,\ell} \geq 0, && \forall i\in\mathcal{P}, T_{k,j}\in\mathcal{T},\ell\in\mathcal{L}
\end{aligned}$$

In this linear program, constraints (1) ensure that every task is completed on a processor in some time interval. Constraints (2) assure that the completion time of a job must be at least the completion time of all its tasks. Constraints (3) impose a lower bound on the completion time of a task while constraints (4) are feasibility constraints indicating that the total processing time of jobs executed up to an interval I_ℓ should be at most $(1+\delta)^\ell$. Lastly, constraints (5) indicate that if a task $T_{k,j}$ has processing time greater than $(1+\delta)^\ell$ on a machine i cannot complete its execution within the interval I_ℓ.

In the algorithm **TaskScheduling**, we begin from a fractional solution of the LP: $(\bar{y}_{i,k,j,\ell}, \bar{C}_{k,j}, \bar{C}_j)$. We partition the set of tasks $T_{k,j}$ into sets $S(\ell) = \{T_{k,j}\in\mathcal{T} | (1+\delta)^{\ell-1} \leq$

$\alpha \bar{C}_{k,j} < (1+\delta)^{\ell}\}$, where $\alpha > 1$ a fixed parameter. After this, we schedule integrally on the processors the of tasks of each set $S(\ell)$ in an increasing order of ℓ using the rounding theorem of Lenstra, Shmoys and Tardos we have seen for the makespan minimization on unrelated machines.

In the following, we begin the analysis of the algorithm by presenting some useful lemmas and observations. First of all, we argue that this linear formulation, even if we restrict the $y_{i,k,j,\ell}$ variables to take integral values $\{0,1\}$ is a $(1+\delta)$ relaxation of the original problem. In order to see this, take any feasible schedule. From this schedule we can directly assign to the variables $y_{i,k,j,\ell}$ values zero or one depending on the interval and processor the tasks complete their execution. Then, from the constraints (2), (3) we see that the completion time of a job and therefore the objective function can be at most a factor of $(1+\delta)$ lesser than the optimal. As a result, this LP is a lower bound to the optimal schedule for this problem.

For the tasks in $S(\ell)$ the following lemma holds:

Lemma 2.11. *Tasks in $S(\ell)$ alone can be fractionally scheduled on processors $\mathcal{P}$ with makespan at most $\frac{\alpha}{\alpha-1}(1+\delta)^{\ell}$.*

Proof. First, we need to prove that for any task $T_{k,j} \in S(\ell)$ it must hold: $\sum_{i\in\mathcal{P}}\sum_{t\geq \ell+1} \bar{y}_{i,k,j,t} \leq \frac{1}{\alpha}$. Suppose it is not the case and $\sum_{i\in\mathcal{P}}\sum_{t\geq \ell+1} \bar{y}_{i,k,j,t} > \frac{1}{\alpha}$ then from (3) we have:

$$\begin{aligned}
\bar{C}_{k,j} &\geq \sum_{i\in\mathcal{P}}\sum_{\ell\in\mathcal{L}}(1+\delta)^{\ell-1}\bar{y}_{i,k,j,\ell} \\
&= \sum_{i\in\mathcal{P}}\sum_{t\geq\ell+1}(1+\delta)^{t-1}\bar{y}_{i,k,j,t} + \sum_{i\in\mathcal{P}}\sum_{t\leq\ell}(1+\delta)^{t-1}\bar{y}_{i,k,j,t} \\
&\geq \sum_{i\in\mathcal{P}}\sum_{t\geq\ell+1}(1+\delta)^{t-1}\bar{y}_{i,k,j,t} \\
&\geq (1+\delta)^{\ell}\sum_{i\in\mathcal{P}}\sum_{t\geq\ell+1}\bar{y}_{i,k,j,t} \\
&> \frac{1}{\alpha}(1+\delta)^{\ell}
\end{aligned}$$

Since $T_{k,j} \in S(\ell)$, then by definition of $S(\ell)$ this is a contradiction.

Now, using the fact that $\sum_{i\in\mathcal{P}}\sum_{t\geq \ell+1} \bar{y}_{i,k,j,t} \leq \frac{1}{\alpha}$, from (1) we can see that:

$$\sum_{i\in\mathcal{P}}\sum_{t\leq\ell}\bar{y}_{i,k,j,t} \geq \frac{\alpha-1}{\alpha}$$

Using these observations we can transform a $\bar{y}_{i,k,j,\ell}$ solution into $y^*_{i,k,j,\ell}$ by setting $y^*_{i,k,j,t} = 0$ for $t \geq \ell+1$ and $y^*_{i,k,j,t} = \frac{\alpha}{\alpha-1}\bar{y}_{i,k,j,t}$ for $t \leq \ell$. We can easily see that constraints (1) and (5) of the LP are satisfied for this transformation:

For (1):

$$\begin{aligned}
\sum_{i\in\mathcal{P},\ell\in\mathcal{L}} y^*_{i,k,j,\ell} &= \sum_{i\in\mathcal{P},t\leq\ell} y^*_{i,k,j,t} \\
&= \sum_{i\in\mathcal{P},t\leq\ell} \frac{\alpha}{\alpha-1}\bar{y}_{i,k,j,t} \\
&= \frac{\alpha}{\alpha-1}\sum_{i\in\mathcal{P},t\leq\ell} \bar{y}_{i,k,j,t} \\
&\geq \frac{\alpha}{\alpha-1}\frac{\alpha-1}{\alpha} \\
&= 1
\end{aligned}$$

In the same way we can see that constraints (4) are satisfied if we multiply the right-hand side of the inequality with $\frac{\alpha}{\alpha-1}$:

$$\begin{aligned}
\sum_{i\in\mathcal{P}}\sum_{\ell\in\mathcal{L}}(1+\delta)^{\ell-1}y^*_{i,k,j,\ell} &\leq \sum_{i\in\mathcal{P}}\sum_{t\leq\ell}(1+\delta)^{t-1}\frac{\alpha}{\alpha-1}\bar{y}_{i,k,j,t} \\
&= \frac{\alpha}{\alpha-1}\sum_{i\in\mathcal{P}}\sum_{t\leq\ell}(1+\delta)^{t-1}\bar{y}_{i,k,j,t} \\
&\leq \frac{\alpha}{\alpha-1}(1+\delta)^{\ell}
\end{aligned}$$

Thus, we see that tasks in $S(\ell)$ can be fractionally scheduled alone in $\mathcal{P}$ with makespan at most $\frac{\alpha}{\alpha-1}(1+\delta)^{\ell}$ and therefore the lemma holds. □

Now, using the rounding theorem for makespan minimization on unrelated processors we have seen in a previous section of this chapter we can turn this fractional schedule of tasks in $S(\ell)$ into an integral one with makespan at most $\frac{\alpha}{\alpha-1}(1+\delta)^{\ell}$ plus the maximum processing time of such a task. From (5) we know that this processing time is bounded by $(1+\delta)^{\ell}$. Therefore the following lemma holds:

Lemma 2.12. *Tasks in $S(\ell)$ alone can be integrally scheduled on processors $\mathcal{P}$ with makespan at most $(\frac{\alpha}{\alpha-1}+1)(1+\delta)^{\ell}$. Also, by definition of $S(\ell)$ the completion time of a task in this schedule is at most $\alpha(\frac{\alpha}{\alpha-1}+1)(1+\delta)\bar{C}_{k,j}$.*

If we take the union of these schedules in an increasing order of ℓ, by applying the algorithm for makespan minimization on each schedule we can see that for each task

$T_{k,j} \in S(\ell)$ executed on the machine i it holds:

$$\begin{aligned}
C_{k,j} &\leq \frac{\alpha}{\alpha-1}(1+\delta)^{\ell} + \sum_{t\leq\ell}(1+\delta)^{t} \\
&\leq \frac{\alpha}{\alpha-1}(1+\delta)^{\ell} + \frac{(1+\delta)^{\ell+1}-1}{\delta} \\
&\leq \frac{\alpha}{\alpha-1}(1+\delta)^{\ell} + \frac{(1+\delta)^{\ell+1}}{\delta} \\
&= \frac{\alpha}{\alpha-1}(1+\delta)^{\ell} + (1+\frac{1}{\delta})(1+\delta)^{\ell} \\
&= (\frac{\alpha}{\alpha-1}+1+\frac{1}{\delta})(1+\delta)^{\ell} \\
&= (\frac{\alpha}{\alpha-1}+1+\frac{1}{\delta})(1+\delta)(1+\delta)^{\ell-1} \\
&\leq \alpha(\frac{\alpha}{\alpha-1}+1+\frac{1}{\delta})(1+\delta)\bar{C}_{k,j}
\end{aligned}$$

Since the values $(\bar{C}_{k,j})$ consist a lower bound for the objective it follows that for the produced schedule:

$$\begin{aligned}
\sum_{j\in\mathcal{J}} w_j C_j &\leq \sum_{j\in\mathcal{J}} w_j \max_{T_{k,j}} C_{k,j} \\
&\leq \alpha(\frac{\alpha}{\alpha-1}+1+\frac{1}{\delta})(1+\delta)\sum_{j\in\mathcal{J}} w_j \max_{T_{k,j}} \bar{C}_{k,j} \\
&\leq \alpha(\frac{\alpha}{\alpha-1}+1+\frac{1}{\delta})(1+\delta)\sum_{j\in\mathcal{J}} w_j \bar{C}_j \\
&\leq \alpha(\frac{\alpha}{\alpha-1}+1+\frac{1}{\delta})(1+\delta)OPT
\end{aligned}$$

, where OPT the optimal objective value of the problem.

Choosing $\alpha = \frac{3}{2}$ and $\delta = \frac{1}{2}$ the quantity $\alpha(\frac{\alpha}{\alpha-1}+1+\frac{1}{\delta})(1+\delta)$ is minimized, leading to the following theorem:

Theorem 2.13. *Algorithm **TaskScheduling** is a polynomial-time $\frac{27}{2}$-approximation for the problem of minimizing the weighted completion time of jobs' tasks on unrelated processors.*

Chapter 3

Precedence Constraints and Shop Scheduling

In this chapter we present some basic types of scheduling problems where precedence constraints between jobs are involved. We introduce some definitions on some of the main scheduling issues in literature concerning shop scheduling and present some known algorithms as examples.

3.1 Chains, Flows and Shops

Before we start, let us recall the notion of precedence constraints between jobs. We say that a job J_1 must precede J_2 if J_2 cannot start its execution before the completion of J_1. In the next, we denote this fact by $J_1 \succ J_2$. Precedence constraints between jobs are very useful for modelling many real-world applications, varying from the concept of assembly line in factories to the classic fork-join model in operating systems.

In the general case, the precedence relations define a *strict partial order* as the relation "$\succ$" satisfies the properties of irreflexibility, transitivity and asymmetry. Therefore, every set of precedence constraints can be depicted as a *Directed Acyclic Graph.* Typical examples of such graphs are the *chain* of jobs, modelling for example the assembly line or the *tree*, modelling the fork-join task model.

In scheduling theory, there is a wide class of problems under the term *shop scheduling.* In this shop scheduling problems, each job consists of a set of operations and each operation has an associated machine on which it has to be processed. The target of shop scheduling problems is to schedule the operations of all jobs in such a way, that at most one operation of a specific job can be processed at a time. There are three well-studied subcategories of shop scheduling: *Open Shop*, *Flow Shop* and *Job Shop.* In

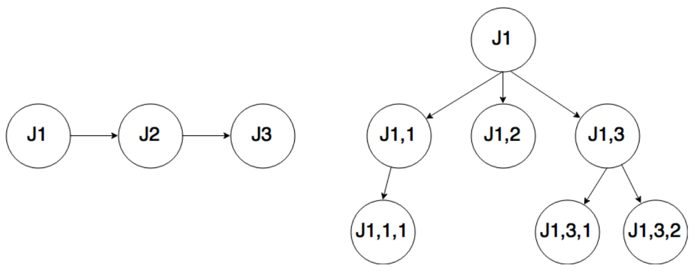

FIGURE 3.1: Examples of Chain- and Tree- Like Precedence Constraints.

the Open Shop problems the operations of a job can be performed in any possible order. In the Job Shop problems the operations of a job must be performed in a specific order while in the Flow Shop case for each job there is one operation per machine and the execution order is fixed and the same for all jobs. The objective of these problems is usually to minimize the makespan of the schedule with respect to the aforementioned constraints.

3.2 A Greedy 2-approximation Algorithm For Open Shops

In the following, we present a 2-approximation algorithm for the problem $O||C_{max}$. In this problem we have a set $\mathcal{M}$ of machines, a set $\mathcal{J}$ of jobs and a set $\mathcal{O} = \{O_1, \dots, O_k\}$ of operations. Each operation belongs to a job j and has to be executed on a specific machine i. For this problem, there is a trivial 2-approximation algorithm. This greedy rule suggests that whenever a machine becomes idle, schedule on this machine any available operation that has to be processed by this machine. Recall that for an open shop problem an operation is considered to be "available" at a time t if no other operation of the same job is running at this time.

Theorem 3.1. *The greedy algorithm is a 2-approximation for $O||C_{max}$.*

Proof. The proof of this theorem follows very naturally after we define two very important and useful lower bounds concerning open shop problems. First of all, if we denote by p_j the total processing need of the operations of a job j and define $P_{max} = \max_{j \in \mathcal{J}} p_j$, then it is definitely the case that $P_{max} \leq C^*_{max}$, where C^*_{max} the optimal makespan. In the same way, if we denote as π_i the total processing time of operations of all jobs that have to be executed on machine i, then the for the value $\Pi_{max} = \max_{i \in \mathcal{M}} \pi_i$ it holds $\Pi_{max} \leq C^*_{max}$.

Let us fix a job j whose operation o finishes last on machine i. There are two reasons why o may not have been scheduled on the same machine a previous time moment t: either machine i was busy at time t executing an operation of another job, either another operation of j was under processing at this time by another machine. From these observations we argue that:

$$C_{max} = C_{o,j} \leq p_j + \pi_i \leq P_{max} + \Pi_{max} \leq 2C^*_{max}$$

□

3.3 Two Machine Flow Shop Makespan Minimization

In this section, we examine the two-machines case flow shop problem, which contrary to the general case with an arbitrary number of machines, accepts a polynomial-time exact algorithm. More specifically, we have two machines M_1 and M_2 and set $\mathcal{J}$ of jobs, each consisting of two operations, one for each machine. We denote O^1_j and O^2_j the operations of job j for machine M_1 and M_2 respectively and p^1_j, p^2_j the processing times of the two operations. For each job j, the O^1_j operation must be executed before O^2_j. Using the usual notation this problem is denoted by $F_2||C_{max}$.

The polynomial-time exact algorithm for this problem is the well-known *Johnson's Rule* [49]. This rule schedules the operations on each machine in the same job-order, making greedy choices depending on the processing times of operations. In the following, we describe the algorithm:

```
**Data**: Set of jobs, and processing times of operations
**Result**: A schedule with minimum makespan
List the processing times of all operations.
while there are unscheduled jobs do
    From the unscheduled jobs, select the operation $O_j$ of a job $j$ with minimum processing time.
    if $O_j$ is for machine $M_1$ then
        Schedule $j$ at the first available position
    else
        Schedule $j$ at the last available position
    end
end
```

Algorithm 1: Johnson's Rule

Theorem 3.2. *Johnson's Rule is a polynomial-time exact algorithm for two-machine flow shop makespan minimization problem.*

Proof. First of all, it is clear that there exists an optimal schedule in which between operations on machine M_1 there is no idle time. Consequently, we are searching for an optimal schedule without idle time between O^1_js and our goal is to minimize the idle time on machine M_2. Furthermore, with a simple interchange argument we can see that there exists an optimal schedule in which the order of job operations remains the same for both the machines. In other words, for a job j, O_j^2 is scheduled after the execution of O_j^1 the first time machine M_2 becomes available.

We prove the optimality of **Johnson's Rule** by a simple interchange argument. We classify the jobs of an instance into two sets:

- *Set A*: The set of jobs j, where $p_j^1 \geq p_j^2$.
- *Set B*: The set of jobs j, where $p_j^1 < p_j^2$.

Suppose we have an optimal schedule that is exactly the same with a schedule produced by **Johnson's Rule** until a job ℓ and exactly after ℓ, there are two jobs j and k that the two algorithms have scheduled in a different order. Without loss of generality, suppose the optimal algorithm has scheduled k after j and the **Johnson's Rule** j after k. If we denote as C_j^i the completion time of a job j on machine i then for the optimal schedule it holds:

$$C_k^1 = C_\ell^1 + p_j^1 + p_k^1$$

$$\begin{aligned}
C_k^2 &= \max\{C_j^2, C_k^1\} + p_k^2 \\
&= \max\{\max\{C_\ell^2 + p_j^2, C_j^1 + p_j^2\}, C_\ell^1 + p_j^1 + p_k^1\} + p_k^2 \\
&= \max\{\max\{C_\ell^2 + p_j^2, C_\ell^1 + p_j^1 + p_j^2\}, C_\ell^1 + p_j^1 + p_k^1\} + p_k^2 \\
&= \max\{C_\ell^2 + p_j^2 + p_k^2, C_\ell^1 + p_j^1 + p_j^2 + p_k^2, C_\ell^1 + p_j^1 + p_k^1 + p_k^2\}
\end{aligned}$$

If we interchange the order of jobs j and k, then in the same way it holds:

$$C_j^{\prime 1} = C_\ell^1 + p_j^1 + p_k^1$$

$$C_j^{\prime 2} = \max\{C_\ell^2 + p_k^2 + p_j^2, C_\ell^1 + p_k^1 + p_k^2 + p_j^2, C_\ell^1 + p_k^1 + p_j^1 + p_j^2\}$$

Johnson's Rule would interchange the two jobs if any of the following conditions holds:

- j and $k \in A$ and $p_j^2 \leq p_k^2$.
- j and $k \in B$ and $p_k^1 \leq p_j^1$.
- $k \in B$ and $j \in A$.

Clearly, it is always the case that $C_k^1 = C_j^{'1}$. All we have to do is to examine the relation between C_k^2 and $C_j^{'2}$ for all possible scenarios.

For the first condition, it holds $p_j^2 \leq p_k^2$, $p_j^1 \geq p_j^2$ and $p_k^1 \geq p_k^2$. As we can see the first terms of C_k^2 and $C_j^{'2}$ are equal. Also, the second and third term of $C_j^{'2}$ are lesser than the third term of C_k^2. In a similar way, we prove the that $C_k^2 \geq C_j^{'2}$ for the second condition.

If the third condition holds then it is the case that $p_k^1 < p_k^2$ and $p_j^1 \geq p_j^2$. Therefore, the second term of $C_j^{'2}$ is smaller than the third term of C_k^2 and the third term of the former is smaller than the second term of the latter.

From the previous analysis we see that any optimal schedule can be turned into a scheduled produced by **Johnson's Rule** without loss. Consequently, **Johnson's Rule** is optimal. □

3.4 A Randomized Algorithm for the Flow Shop Scheduling Problem

In this section, we present a randomized algorithm for the flow shop scheduling problem [50] for a non-fixed number of machines. Recall that in a flow shop problem we have n jobs and m machines. Each job has a set of m operations, one for each machine. We denote by $p_{i,j}$ the processing time of the operation of job j on machine i and by $p_{max} = \max p_{i,j}$ the maximum processing time of all operations. In this kind of problems, the operations of all jobs must be executed in a specific machine order and this order is the same for all jobs. Therefore we can enumerate the machines without loss of generality with respect to this order. It is well-known that the problem of makespan minimization in flow shops is strongly NP-hard for more than two machines.

The first step of our algorithm is to divide the time domain of each machine into *time-slots* of size s, where $s \geq 2p_{max}$. Assuming that all operation have size at least 1, then each slot contains at least one operation. In order for our algorithm to work, we assume that the slots have the property of *independence* i.e. the order of operations that are executed on a slot-machine pair is independent of the order on other slot-machine pairs. For this reason, we restrict the operations of a slot not to overpass the slot boundaries as well as the operations of a job not to change machine in the middle of a slot. Given these restrictions we can prove the following lemma:

Lemma 3.3. *Any optimal flow shop schedule with makespan OPT can be transformed into a schedule satisfying the slotting constraints with makespan at most* $\frac{s}{s-p_{max}}OPT + (m-1)s$.

Proof. Firstly, we divide the time domain of the optimal schedule in time-slots of size $s - p_{max}$. This may probably lead to the straddling of two neighbouring time-slots of the same machine by some operations. Now, if we increase the size of each slot by p_{max}, then we can fit these "problematic" operations to the first slot they cross with safety. This transformation can augment the makespan of the optimal schedule at most by a multiplicative factor of $\frac{s}{s-p_{max}}$. In order to satisfy the second independence constrain of the slotting transformation, we need to ensure that no job will "migrate" in the middle of a slot. This can be accomplished if we shift all operations of a slot in machine i by $i - 1$ slots. Given that we begin the numbering of machines from $i = 1$, this can cost to the makespan an additive factor of $(m - 1)s$. □

We now obtain an approximation for the flow shop scheduling problem satisfying the slotting constraints. In this direction we define the following graph: Let $G(V, E)$ be a directed graph having a vertex for each pair of (time-slot,machine). We also add to the set of vertices two nodes s_j and t_j for each job. As far as the edges are concerned, we add a directed edge from each vertex (α, i) to vertex $(\beta, i + 1)$, where $i \in \{1 \ldots m - 1\}$ and $\alpha < \beta$. In addition, each vertex s_j has edges to all vertices corresponding to the first machine and each vertex t_j accepts edges from all vertices corresponding to the last machine.

Given $G(V, E)$ we define a multicommodity flow instance, where each job j is associated with a commodity and s_j and t_j the source and sink nodes of this commodity respectively. If for a vertex $u = (\alpha, i)$ we denote by $x_{u,j}$ the flow of the commodity j, then we require that:

$$\sum_{j \in \mathcal{J}} x_{u,j} p_{i,j} \leq s$$

Given this instance, we wish to route one unit of flow for each commodity j. It is now clear that an integral multicommodity flow for this instance corresponds to a flow shop schedule satisfying the slotting constraints. It is a known result [41] that the feasibility of a multicommodity flow can be determined in polynomial time using linear programming.

With the use of this structure, in our algorithm we try to "guess" the number of time-slots required by the schedule. Clearly, the infeasibility of a multicommodity flow implies that our guess is too small. If we denote by k the number of slots for which the multicommodity flow instance is feasible then for the minimum makespan T of the corresponding flow shop schedule under slotting constraints it is the case that $T \geq (k - 1)s$.

At this point, we are ready to describe the algorithm **Random-FS**. First, we find the minimum number of slots, where the corresponding multicommodity flow instance is feasible. Let F be this flow, then in F, we define the *weight* of a commodity j on an specific path from s_j to t_j to be the flow of this commodity passing through this path.

Clearly, the total weight of all paths corresponding to a specific commodity is one. Then, **Random-FS** picks for each commodity one path with probability equal to its weight. The set of n picked paths correspond to an integral multicommodity flow and therefore a slot constrained flow shop schedule, which may be infeasible in terms of time slot capacity. In other words, in this randomized rounded flow, for the sum of flow passing through a time-slot it may hold that: $\sum_{j \in \mathcal{J}} x_{u,j} p_{i,j} > s$.

For the algorithm **Random-FS**, the following theorem can be proven:

Theorem 3.4. ***Random-FS*** *is a polynomial time randomized algorithm for flow shop scheduling which with probability at least $\frac{1}{2}$ finds a schedule with makespan at most:*

$$2(1+\delta)OPT + m(1+\delta)p_{max}\log_c 2m(n+m-1)$$

, where $c = \frac{(1+\delta)^{(1+\delta)}}{e^{\delta}}$ and δ a constant chosen so that $c > 1$.

Chapter 4

Scheduling MapReduce Jobs Minimizing Total Completion Time

In this chapter, we survey some known results on the problem of scheduling MapReduce jobs. The algorithms we discuss in this part were mainly presented in the work of Moseley et al. "On Scheduling Map-Reduce Jobs and Flow-Shops" [24] and include online and offline algorithms for scheduling MapReduce tasks or identical as well as unrelated parallel machines minimizing the total flow time.

4.1 The General Model

In the next sections, unless stated otherwise, we are going to alternate the notation as it follows. Let $\mathcal{J}$ be the set of MapReduce jobs. In this setting, a *job* consists of two set of *tasks*, the *map* and the *reduce* set. Tasks in each set can run in parallel while the two sets must run sequentially. In other words, in order to start the processing of any reduce task, all map tasks of the same job must have completed their execution. Let $J \in \mathcal{J}$ denote a generic job. In this case, $\{J_i^m\}$ and $\{J_i^r\}$ denote the sets of map and reduce tasks of J. Let $p_x(\bullet)$ denote the processing time of a task or job on machine i, given that we may have a *single task case* if $\{J_i^m\}$ and $\{J_i^r\}$ are singletons or *multiple task case* elsewhere.

Throughout this section we use $b \in \{m, r\}$ to capture both map and reduce related statements. Let $J^{b,*} = \arg\max_i p(J_i^b)$ the task with maximum processing time in a b-set of tasks and let $J^* = \arg\max\{p(J^{m,*}), p(J^{r,*})\}$ be the task with maximum processing time in all sets.

Moreover, let α_J the time of arrival of job J. Given a schedule σ of jobs, let $s_\sigma(\bullet)$ denote the starting time of a job or task in σ and $C_\sigma(\bullet)$ the completion time respectively. We denote by $\mathcal{P}_{\mathcal{M}}$ both the set and the number of map machines and by $\mathcal{P}_{\mathcal{R}}$ the set and number of reduce machines respectively.

We define the *flowtime* of a job J, with respect to a schedule σ, to be $\text{flow}_\sigma(J) = C_\sigma(J) - \alpha_J$. Let $\text{flow}_\sigma = \sum_{J \in \mathcal{J}} \text{flow}_\sigma(J)$ be the *total (average) flowtime* of σ. Note here that for the offline case where jobs are available from the beginning, thus $\alpha_J = 0, \forall J \in \mathcal{J}$, the total flowtime is equal to the total completion time of a schedule.

A schedule σ is called *viable* in the MapReduce setting if all map (resp. reduce) tasks of a job J are schedule only on map (resp. reduce) machines and all map tasks of a job J must have completed their execution before the starting of a reduce task of the same job. Also this schedule is called *non-migratory* if each task is processed by exactly one machine.

4.2 Offline Algorithms

In this section we present two algorithms for scheduling offline MapReduce jobs on identical and unrelated machines respectively. The objective for both algorithms is to minimize the total completion time. The produced schedules in every case are viable and non-migratory.

4.2.1 Identical Machines

In order to create an algorithm for the identical machines case, we firstly simulate the schedules of map tasks alone (resp. reduce tasks alone), on a single P_M-speed (resp. P_R-speed) machine. Using the results of these simulations we create a viable MapReduce parallel schedule with the assistance of the **MR-Identical** algorithm.

We claim that if we use the Shortest Remaining Processing Time Rule (SRPT) for simulating schedules σ_M and σ_R the algorithm above produces a MapReduce schedule which is a $12-$approximation for the problem of minimizing total completion time. In order to prove this claim it is crucial to verify the correctness of the following lemma:

Lemma 4.1. *Given schedule σ_m and σ_r, there is a viable, non-migratory schedule σ such that for all jobs J it is the case that $C_\sigma(J) \leq 4\max\{C^m_{\sigma_m}(J), C^r_{\sigma_r}(J), p(J^*)\}$.*

Proof. A useful observation is that for any task J^b_i that was available for scheduling by our algorithm by time $a_\sigma(J^b_i)$ it holds that $C_\sigma(J^b_i) \leq a_\sigma(J^b_i) + 2\omega_J$. Assuming the contrary we can see that by definition of width, in the time interval $[a_\sigma(J^b_i), C_\sigma(J^b_i) -$

```
1: Simulate the schedules $\sigma_m$ of only the map tasks on a single $P_M$-speed machine
   and $\sigma_r$ of only the reduce tasks on a single $P_R$-speed machine.
2: $\omega_J \leftarrow \max\{C^m_{\sigma_M}(J), C^r_{\sigma_R}(J), p(J^*)\}$
3: for each job $J \in \mathcal{J}$ by $\omega_J$ non-decreasing do
4:    for each map task $J_i^m$ of job $J$ do
5:       Assign $J_i^m$ to the least loaded machine
6:    end for
7:    for each reduce task $J_i^r$ of job $J$ do
8:       Let $x$ be the earliest available reduce machine
9:       if $x$ is available before time $\omega_J$ then
10:         Idle $x$ till time $2\omega_J$
11:      end if
12:      Assign $J_i^r$ to $x$
13:   end for
14: end for
```

Algorithm 2: MR-Identical

$p(J^*)]$, only tasks of width at most ω_J are executed, representing a total volume of work strictly more than $P_b\omega_J$. However, since σ_m and σ_r are simulated by a single P_M-speed machine and a single P_R-speed machine respectively this implies that σ_b must complete strictly more than $P_b\omega_J$ volume of work by time ω_J, leading to a contradiction.

As we can see the algorithm schedules map task in a non-decreasing order of ω_J non-preemptively to the so-far least loaded map machine. Therefore, since for all map tasks $a_\sigma(J_i^m) = 0$, from the previous observation it is the case that $C_\sigma(J_i^m) \leq 2\omega_J$.

For the reduce tasks, by definition of the algorithm, the reduce tasks of a job of width ω_J are not consider for scheduling before time $2\omega_J$. Again, in the produced schedule, the reduce tasks are scheduled on reduce machines only in a non-preemptive, non-migrating way. By the previous observation, if we set $a_\sigma(J_i^m) = 2\omega_J$ then it is the case that $C_\sigma(J) = C^r_\sigma(J) \leq 4\omega_J = 4\max\{C^m_{\sigma_m}(J), C^r_{\sigma_r}(J), p(J^*)\}$. □

As we know, the SRPT rule is optimal for the average flowtime (completion time) for the single machine case where there is one task per job and no precedence constraints. Since having more than one task per job is irrelevant in our case, the total completion time of only map and only reduce on a P_M-speed and a P_R-speed respectively consist lower bounds for our objective. Let OPT denote the optimal schedule. Then it is the case that $\max\{flow_{\sigma_M}, flow_{\sigma_R}\} \leq flow_{OPT}$. Also, an obvious lower bound to the total completion time is the sum of the processing times of the "largest" tasks of each job. Thus $\sum_{J\in\mathcal{J}} p(J^*) \leq flow_{OPT}$. Keeping these in mind and using the previous lemma we can see that:

$$\begin{aligned}
flow_\sigma &= \sum_{J\in\mathcal{J}} C_\sigma(J) \\
&\leq \sum_{J\in\mathcal{J}} 4\max\{C^m_{\sigma_m}(J), C^r_{\sigma_r}(J), p(J^*)\} \\
&\leq 4(\sum_{J\in\mathcal{J}} C^m_{\sigma_m}(J) + \sum_{J\in\mathcal{J}} C^r_{\sigma_r}(J) + \sum_{J\in\mathcal{J}} p(J^*)) \\
&= 4(flow_{\sigma_m} + flow_{\sigma_r} + \sum_{J\in\mathcal{J}} p(J^*)) \\
&\leq 4(flow_{OPT} + flow_{OPT} + flow_{OPT}) \\
&\leq 12\sum_{J\in\mathcal{J}} C_{OPT}
\end{aligned}$$

Theorem 4.2. *There exists a non-migratory* 12*-approximation algorithm for flowtime (completion time) in the offline, identical machines, multiple task, MapReduce setting.*

4.2.2 Unrelated Machines

We now turn our attention to the unrelated machine setting. In this problem, restricting ourselves to the single task case, we work in a quite similar way as in the identical machine case. First, we simulate schedules σ_m and σ_r of only map and only reduce tasks on P_m and P_r machines respectively. Given these schedules we again define the width of each job as $\omega_J = \max\{C_{\sigma_m}(J), C_{\sigma_r}(J)\}$. In this case, the assignment of tasks to machines is maintained from schedules σ_m and σ_r to σ. The algorithm works as follows: first we reorder the map tasks on each machine in a non-decreasing order of ω_J. For the reduce tasks, on each time and on each reduce machine, we schedule the available reduce task with the smaller width. Similarly to the identical machines case the following lemma gives a generic lower bound to the completion time of each task in σ.

Lemma 4.3. *For any task J^b that becomes available at time $a_\sigma(J^b)$ it holds that $C_\sigma(J^b) \leq a_\sigma(J^b) + \omega_J$.*

Proof. Let us assume the contrary: there exists a job J assigned to a machine i such that $C_\sigma(J^b) > a_\sigma(J^b) + \omega_J$. Then in the time interval $[a_\sigma(J^b), C_\sigma(J^b)]$, by definition of width and since there are no idle times, machine i processes more than ω_j units of work of tasks with width at most ω_J. This is a contradiction since it would imply that σ_b processes strictly more than ω_J units of work by time ω_J. □

Using the previous lemma we can now prove the following theorem:

Theorem 4.4. *There exists a non-migratory 6-approximation algorithm for the flowtime (total completion time) in the offline, unrelated machine single task MapReduce setting.*

Proof. We can see that $a_\sigma(J^m) = 0$ for all map tasks and also $a_\sigma(J^r) = \max_{J^m \in J}\{C_\sigma(J^m)\} \leq \omega_J$. Applying the previous lemma we can prove that $C_\sigma(J) = C_\sigma(J^r) \leq 2\omega_J$.

In [51], Skutella presented a $\frac{3}{2}$-approximation algorithm for minimizing total completion time on unrelated machines, where we have single-task jobs and no precedence constraints. Therefore, if we use this algorithm to simulate schedules σ_m and σ_r we see that $\frac{2}{3}\max\{flow_{\sigma_m}, flow_{\sigma_r}\} \leq flow_{OPT}$. Thus:

$$\begin{aligned}
flow_\sigma &= \sum_{J \in \mathcal{J}} C_\sigma(J) \\
&\leq \sum_{J \in \mathcal{J}} 2\omega_J \\
&\leq \sum_{J \in \mathcal{J}} 2\max\{C_{\sigma_m}(J), C_{\sigma_r}(J)\} \\
&\leq 2\sum_{J \in \mathcal{J}} (C_{\sigma_m}(J) + C_{\sigma_r}(J)) \\
&\leq 2(flow_{\sigma_m} + flow_{\sigma_r}) \\
&\leq 6 flow_{OPT}
\end{aligned}$$

□

4.3 Online Scheduling

We turn our attention now to the case of scheduling online MapReduce jobs on identical and unrelated machines respectively. The objective for both algorithms is, again, to minimize the total completion time with the creation of viable and non-migratory schedules.

4.3.1 Identical Machines

In the identical machine case, we consider the online scheduling of a fixed sequence of jobs. Like the offline case, we simulate schedules σ_m and σ_r on a single $\mathcal{P}_\mathcal{M}$-speed and a single $\mathcal{P}_\mathcal{R}$-speed machine respectively. For this case, we are going to limit ourselves to a simple presentation of the algorithm and a brief proof sketch. The analysis of this algorithm is quite technical and the simple ideas used will be demonstrated in the analysis of the unrelated machine case.

As we have stated before, the algorithm firstly simulates the online schedules σ_m and σ_r. After the simulation, the *width* of a job J is defined as $\omega_J = \max\{(\max\{C^m_{\sigma_m}(J), C^r_{\sigma_r}(J)\} - a_J), p(J^*)\}$, where a_J the arrival time of a job J and $p(J^*)$ the processing time of the larger task of this job. A job is said to be in *class k* if $\omega_J \in [2^k, 2^{k+1})$. Let $U^{m,x}_{=k}(t)$

be the total processing time of map tasks in class k assigned to the map machine x by time t. In the same way, let $U^{m,x}_{=k}(t)$ denote total processing time of reduce tasks in class k assigned to the reduce machine x by time t. Schedule σ is created by the online algorithm **OMR-Identical(t)**. Moseley et al. [24] proved the following lemma for

```
1: Simulate the schedules $\sigma_m$ and $\sigma_r$.
2: if t is the first time all map tasks for job $J$ are finished in $\sigma_m$ and all reduce tasks
   for job $J$ are finished in $\sigma_r$ then
3:    Let $k$ be $J$'s class.
4:    for each map task $J_i^m$ of job $J$ do
5:       Assign $J_i^m$ to the map machine $x$ with the minimum $U^{m,x}_{=k}(t)$.
6:       $U^{m,x}_{=k}(t) \leftarrow U^{m,x}_{=k}(t) + p(J_i^m)$
7:    end for
8: end if
9: if t is the first time that all map tasks for job $J$ are finished in schedule $\sigma$ then
10:   Let $k$ be $J$'s class.
11:   for each reduce task $J_i^r$ of job $J$ do
12:      Assign $J_i^r$ to the reduce machine $x$ with the minimum $U^{r,x}_{=k}(t)$.
13:      $U^{r,x}_{=k}(t) \leftarrow U^{r,x}_{=k}(t) + p(J_i^r)$
14:   end for
15: end if
16: On each map and reduce machine, run the task assigned to that machine such that
    its associated job has minimum width.
```

Algorithm 3: OMR-Identical(t)

OMR-Identical:

Theorem 4.5. *Given online schedule σ_m and σ_r,* ***OMR-Identical*** *produces a viable, online, non-migratory $(1+\epsilon)$-resource augmented schedule σ such that $C_\sigma(J) \leq a_J + \frac{128}{\epsilon^2}\max\{(\max\{C^m_{\sigma_m}(J), C^r_{\sigma_r}(J)\} - a_J), p(J^*)\}$.*

Now, if we simulate σ_m and σ_r using the SRPT rule and following the analysis of the identical machines offline case, then applying the above theorem we can easily prove the following.

Theorem 4.6. ***OMR-Identical(t)*** *with σ_m and σ_r simulated using SRPT rule yields a non-migratory $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$ competitive algorithm for the average flowtime in the online, identical machines, multiple task, MapReduce setting, where $0 < \epsilon \leq 1$.*

4.3.2 Unrelated Machines

In this setting we again consider the single task case, where jobs arrive over time and a_J the arrival time of a job J. In this case, our algorithm simulates schedules σ_m and σ_r in an online way. We define as *width* of a job the quantity $\omega_J = \max\{C_{\sigma_m}(J), C_{\sigma_r}(J)\} - a_J$. Our algorithm, as expected, creates a schedule σ by scheduling at each time t on a machine i the task of the available job with minimum width. The resulting schedule is clearly online, non-migratory and viable.

It has been proven [52] that there is no online algorithm with bounded competitive ratio for the objective of flowtime. Therefore, like the previous case, we use resource augmentation, giving to our schedule a minimum advantage of ϵ over the adversary. The following lemma is a first step towards a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive online algorithm.

Lemma 4.7. *Let $\alpha > 0$ and suppose the task J^b, $b \in \{m, r\}$, is available for scheduling by our schedule σ at time $a_J + \alpha\omega_J$. The it is the case that $C_\sigma(J^b) \leq a_J + \frac{2\alpha}{\epsilon}\omega_J$.*

Proof. Suppose the job J^b is assigned to a machine x in σ_b. Let t_b be the earliest time such that every task processed by machine x in σ during the interval $[t_b, C_\sigma(J^b)]$ has width at most ω_J. Given that J^b is available at time $a_J + \alpha\omega_J$ and from the fact that machine x processes at any time the available jobs of minimum width, it must be the case that $t_b \leq a_J + \alpha\omega_J$.

Also, it must be the case that any task J' scheduled during $[t_b, C_\sigma(J^b)]$ arrived at earliest $t_b - \alpha\omega_J$. This is because J' must have $\omega_{J'} \leq \omega_J$ as it is scheduled before J^b by our algorithm. Therefore, $a_{J'} \geq t_b - \alpha\omega_{J'} \geq t_b - \alpha\omega_J$.

Now, given that in schedule σ machine x has speed $(1+\epsilon)$, then it produces a total work of $(1+\epsilon)(C_\sigma(J^b) - t_b)$. Then, it must hold that in σ_b, machine x must complete at least this amount of work during the interval $[t_b - \alpha\omega_J, C_\sigma(J^b)]$. Therefore:

$$\begin{aligned}
(1+\epsilon)(C_\sigma(J^b) - t_b) &\leq C_\sigma(J^b) - t_b + \alpha\omega_J \\
\epsilon C_\sigma(J^b) &\leq \epsilon t_b + \alpha\omega_J \\
C_\sigma(J^b) &\leq t_b + \frac{\alpha}{\epsilon}\omega_J \\
C_\sigma(J^b) &\leq a_J + \alpha\omega_J + \frac{\alpha}{\epsilon}\omega_J \\
C_\sigma(J^b) &\leq a_J + \frac{2\alpha}{\epsilon}\omega_J
\end{aligned}$$

, given that $0 < \epsilon < 1$. □

We now prove the following theorem:

Theorem 4.8. *Given online non-migratory schedules σ_m and σ_r, there is a viable, online, non-migratory $(1+\epsilon)$-resource augmented schedule σ such that all tasks for job J are completed by time $a_J + \frac{4}{\epsilon^2}(\max\{C_{\sigma_m}(J), C_{\sigma_r}(J)\} - a_J)$*

Proof. Given that that map task J_m is available to σ for scheduling at time $a_J + \frac{2}{\epsilon}\omega_J$, then from the above lemma we can see that by setting $\alpha = 1$ it holds that:

$$C_\sigma(J^m) \leq a_J + \frac{2}{\epsilon}\omega_J$$

Similarly, given now that all map tasks of a job J must have complete their execution in σ by time $a_J + \frac{2}{\epsilon}\omega_J$, then by setting again $\alpha = \frac{2}{\epsilon}$ for the completion time of tasks J^r and therefore for the completion time of job J it is the case that:

$$C_\sigma(J) = C_\sigma(J^m) \leq a_J + \frac{4}{\epsilon^2}\omega_J$$

□

Chadha et al. [53] proved a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive online non-migratory algorithm for the average flowtime on unrelated machines, when there is only one task per job and no precedence constraints. With the use of their algorithm we can prove the following theorem:

Theorem 4.9. *There exists a non-migratory $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive online algorithm for the average flowtime objective in the online, unrelated machines, single task, MapReduce setting.*

Proof. Given the result in [53], we can generate schedules σ_m and σ_r in a way such that if we denote by OPT the optimal flowtime it is the case that:

$$\Omega(\epsilon^2)\max\{flow_{\sigma_m}, flow_{\sigma_r}\} \leq OPT$$

By applying the previous theorem we can see that:

$$\begin{aligned}
flow_\sigma &\leq \sum_{J\in\mathcal{J}}(C_\sigma(J) - a_J) \\
&\leq \frac{4}{\epsilon^2}\sum_{J\in\mathcal{J}}\omega_J \\
&\leq \frac{4}{\epsilon^2}\sum_{J\in\mathcal{J}}(\max\{C_{\sigma_m}(J), C_{\sigma_r}(J)\} - a_J) \\
&\leq \frac{4}{\epsilon^2}\sum_{J\in\mathcal{J}}\max\{C_{\sigma_m}(J) - a_J, C_{\sigma_r}(J) - a_J\} \\
&\leq \frac{4}{\epsilon^2}\sum_{J\in\mathcal{J}}(C_{\sigma_m}(J) - a_J + C_{\sigma_r}(J) - a_J) \\
&\leq \frac{4}{\epsilon^2}(flow_{\sigma_m} + flow_{\sigma_r}) \\
&\leq O(\frac{1}{\epsilon^4})OPT
\end{aligned}$$

Since we can simulate σ_m and σ_r with the algorithm of Chadha et al. only with a minimum advantage ϵ over the adversary, the theorem follows. $\square$

Chapter 5

Scheduling MapReduce Jobs and Shuffle Tasks

In this chapter, we present a constant approximation algorithm for the MapReduce scheduling problem on unrelated processors, where the objective is to minimize the total weighted completion time of jobs. To the best of our knowledge, this problem formulation is the most general version of MapReduce scheduling so-far, as we enable each job to have an arbitrary number of map and reduce tasks. Furthermore, we propose a fast heuristic for the same problem and we include in our analysis the modelling of *data shuffle*, i.e. the overhead of the communication cost between map and reduce tasks.

5.1 A constant approximation algorithm.

In the following we present **Algorithm-MR** for the problem of minimizing total weighted completion time of MapReduce tasks on unrelated machines. Before we begin the presentation we introduce some notation: We denote by $\mathcal{M}$ and $\mathcal{R}$ the sets of map and reduce tasks respectively and by $\mathcal{P}_{\mathcal{M}}$ and $\mathcal{P}_{\mathcal{R}}$ the disjoint sets of map and reduce machines. The notation $\mathcal{T}_{k,j}$ represent the task k of a job j and $p_{i,k,j}$ represent the processing time of $\mathcal{T}_{k,j}$ on machine i.

Algorithm-MR works as follows. At first, the algorithm schedules separately the map and reduce tasks on the corresponding machines, using the algorithm **TaskScheduling** as a subroutine. Recall from chapter 2 that the algorithm **TaskScheduling** is a $\frac{27}{2}$-approximation algorithm for the problem of scheduling jobs with multiple tasks on unrelated machines in order to minimize the total weighted completion time. We denote by $\sigma_{\mathcal{M}}$ and $\sigma_{\mathcal{R}}$ the two produced schedules and by $C_{k,j}^{\sigma_{\mathcal{M}}}$ and $C_{k,j}^{\sigma_{\mathcal{R}}}$ the completion time of a map or reduce task $\mathcal{T}_{k,j}$ in the respective σ_b schedule, where $b \in \{m, r\}$. After the simulation of map and reduce schedules, we use the completion times of jobs in these schedules to merge them, keeping a constant approximation guarantee using a simple routine. In this direction, we define for each job the quantity $\omega_J = \max\{C_j^{\sigma_\mathcal{M}}, C_j^{\sigma_\mathcal{R}}\}$.

```
1: Run **TaskScheduling** on both the sets of tasks $\mathcal{M}$ and $\mathcal{R}$ creating schedules $\sigma_\mathcal{M}$
   and $\sigma_\mathcal{R}$ respectively.
2: Assign each task on the same processors as they are in schedules $\sigma_\mathcal{M}$ and $\sigma_\mathcal{R}$.
3: for each job $j \in \mathcal{J}$ do
4:   Fix $\omega_j = \max\{C_j^{\sigma_\mathcal{M}}, C_j^{\sigma_\mathcal{R}}\}$ to be the *width* of job $j$
5: end for
6: for each time $t$ where a processor $i \in \mathcal{P}$ becomes available do
7:   if $i = \mathcal{P}_\mathcal{M}$ then
8:     Among the unscheduled map tasks in $i$, schedule task $\mathcal{T}_{k,j} \in \mathcal{M}$ with the
       smallest $\omega_j$ with processing time $p_{i,k,j}$. Let $C_{k,j}$ be the completion time of task
       $\mathcal{T}_{k,j}$.
9:   else
10:    Among the unscheduled reduce tasks, which have $\omega_j > t$, schedule task
       $\mathcal{T}_{k,j} \in \mathcal{R}$ with the smallest $\omega_j$ with processing time $p_{i,k,j}$. Let $C_{k,j}$ be the
       completion time of task $\mathcal{T}_{k,j}$.
11:  end if
12:  Let $C_{k,j}$ be the completion time of task $\mathcal{T}_{k,j}$.
13: end for
14: for each job $j \in \mathcal{J}$ do
15:   Compute the completion time $C_j = \max_{\mathcal{T}_{k,j'} \in \mathcal{R} | j'=j}\{C_{k,j'}\}$.
16: end for
```

Algorithm 4: Algorithm-MR

At this point, we are going to prove the following theorem about **Algorithm-MR**:

Theorem 5.1. ***Algorithm-MR*** *is a 54-approximation algorithm for the MapReduce scheduling problem of minimizing total weighted completion time on unrelated processors.*

Proof. In the following, we denote by C_j the completion time of a job produced by **Algorithm-MR**, by C_j^{OPT} the completion time in an optimal schedule, by $C_j^{OPT_M}$ and $C_j^{OPT_R}$ the optimal completion times for the problems of scheduling only map and only reduce tasks respectively and by $C_j^{\sigma_M}$ and $C_j^{\sigma_R}$ the completion times for the schedules σ_M and σ_R produced using **TaskScheduling**.

Clearly, the optimal solutions to the problems of scheduling only map or only reduce tasks consist lower bounds to the optimal solution of the problem. Therefore, it holds:

$$\sum_{j \in \mathcal{J}} w_j C_j^{OPT_M} \leq \sum_{j \in \mathcal{J}} w_j C_j^{OPT}$$
$$\sum_{j \in \mathcal{J}} w_j C_j^{OPT_R} \leq \sum_{j \in \mathcal{J}} w_j C_j^{OPT}$$

Directly from the analysis of **TaskScheduling** on chapter 2 it follows that for schedules $\sigma_{\mathcal{M}}$ and $\sigma_{\mathcal{R}}$ it is the case:

$$\sum_{j\in\mathcal{J}} w_j C_j^{\sigma_{\mathcal{M}}} \leq \frac{27}{2} \sum_{j\in\mathcal{J}} w_j C_j^{OPT_M}$$
$$\sum_{j\in\mathcal{J}} w_j C_j^{\sigma_{\mathcal{R}}} \leq \frac{27}{2} \sum_{j\in\mathcal{J}} w_j C_j^{OPT_R}$$

For the merging routine, we first need to prove that the produced schedule is a non-preemptive one. While for the map tasks this argument is obvious, for the reduce tasks we need to be more careful. The only way we can have preemption during the execution of a reduce task $\mathcal{T}_{k,j}$ is the case where another task $\mathcal{T}_{k',j'}$ of width less than ω_j becomes available. However this cannot be the case, because by definition of the algorithm, $\mathcal{T}_{k',j'}$ should be available before $\mathcal{T}_{k,j}$ and therefore should have been scheduled first.

Furthermore, we can prove that a map task of width ω_j is completed before time ω_j. Suppose this is not the case and a map task, finishes its execution at time $t > \omega_j$. Then, it should be the case that during the interval $[0, t]$, all machines are busy processing tasks of width less than ω_j. This is a contradiction, as in this case the machine i where the task is scheduled should process more than ω_j units of work by time ω_j in schedule $\sigma_{\mathcal{M}}$.

In the same way, we can prove that a reduce task of width ω_j that is available for scheduling at time r on processor i must complete its execution by time at most $r + \omega_j$. Suppose that a reduce task finishes its execution at time $t > r+\omega_j$ on processor i. Then, in the interval $[r, t]$ on processor i there is not idle time and only tasks of width at most ω_j are executed. This is again a contradiction as in this case in schedule $\sigma_{\mathcal{R}}$ by time at most ω_j there must have been processed more than ω_j units of work by the processor i.

A useful remark is that even for the case of non necessarily disjoint map and reduce processors the same analysis applies if we define $\omega_j = C_j^{\sigma_{\mathcal{M}}} + C_j^{\sigma_{\mathcal{R}}}$.

Now, if we set $r = 0$ for the map tasks it follows that in the produced schedule σ, for a map task $\mathcal{T}_{k,j}$ it holds that: $C_{k,j} \leq \omega_j$. Similarly, since all map tasks of a job of width ω_J have completed their execution by time ω_J, then the reduce tasks of the same job are released by time $r \leq \omega_j$. Using the previous argument we can see that for the completion time of these reduce tasks it is the case that $C_{k,j} \leq 2\omega_j$.

Therefore, for the resulting schedule it must be the case that:

$$
\begin{aligned}
\sum_{j\in\mathcal{J}} w_j C_j &= \sum_{j\in\mathcal{J}} w_j \max_{\mathcal{T}_{k,j'}\in\mathcal{R}|j'=j}\{C_{k,j'}\} \\
&\leq \sum_{j\in\mathcal{J}} w_j 2\omega_j \\
&= 2\sum_{j\in\mathcal{J}} w_j \max\{C_j^{\sigma_\mathcal{M}}, C_j^{\sigma_\mathcal{R}}\} \\
&\leq 2\sum_{j\in\mathcal{J}} w_j (C_j^{\sigma_\mathcal{M}} + C_j^{\sigma_\mathcal{R}}) \\
&\leq 2(\frac{27}{2}\sum_{j\in\mathcal{J}} w_j C_j^{OPT_M} + \frac{27}{2}\sum_{j\in\mathcal{J}} \omega_j C_j^{OPT_R}) \\
&\leq 54\sum_{j\in\mathcal{J}} w_j C^{OPT}
\end{aligned}
$$

With this analysis we conclude that this **Algorithm-MR** is a polynomial time 54-approximation for the problem of scheduling MapReduce tasks on unrelated processors minimizing the weighted completion time. □

5.2 Data Shuffle

An important aspect affecting the performance of MapReduce systems is the overhead of data transmission. For this reason, we include in our model another phase which we will refer to as *Shuffle phase*. Shuffle phase takes place between the execution of map and reduce phases. In this phase, the key-value pairs are transmitted from map to reduce tasks of each job. In this section, we are going to extend the analysis of our algorithm in order to include shuffle tasks, modelling in this way the time overhead of data transmission. In the following we will refer to this problem as *MapShuffleReduce* problem.
In this extended model, the following properties must hold:

- Each shuffle task can start its execution only after the completion of the corresponding map task.
- For every map task the number of shuffle tasks with produced data is equal to the number of reduce tasks of the same job. Of course, when there is no data to be transmitted between a map and a reduce task, the corresponding shuffle task has zero processing time.
- The shuffle tasks must be executed non-preemptively.

- The processing times of shuffle tasks transmitted to the same reduce processor cannot overlap with each other.

In order to include the shuffle tasks we introduce some additional notation. We introduce a set of shuffle tasks $\mathcal{T}_{r,k,j}$, where $1 \leq r \leq |\mathcal{T}_{k,j} \in \mathcal{R}|$ for each map task $\mathcal{T}_{k,j} \in \mathcal{M}$ of job j. We denote by $\mathcal{H}$, the set of shuffle tasks. Each of these tasks is associated with a transfer time $t_{r,k,j}$, which is independent from the assignment of the involved map and reduce tasks to processors.

In the following, we discuss two different variations of the MapShuffleReduce problem and present two constant approximation algorithms for both.

5.2.1 The Shuffle Tasks are Executed on their Reduce Processors

In the first case, we consider the problem where the shuffle tasks are executed on the same reduce processor as the corresponding reduce task. In this case, all we have to do is to increase the processing time of each reduce task by the sum of the processing times of the correlated shuffle tasks. For this reason, we consider a reduce task $\mathcal{T}_{r,k} \in \mathcal{R}$ of a job j and let $s_j^r = \{\mathcal{T}_{r,k,j} | \mathcal{T}_{k,j} \in \mathcal{M}\}$, the set of shuffle task that must complete before $\mathcal{T}_{r,j}$ starts its execution. In other words, we can reformulate the input in the following way. For each reduce task $\mathcal{T}_{r,k} \in \mathcal{R}$ running on processor i we set:

$$p'_{i,r,j} \leftarrow p_{i,r,j} + \sum_{\mathcal{T}_{r,k,j} \in s_j^r} t_{r,k,j}$$

For this new input, we can now use **Algorithm-MR** to obtain a feasible schedule. The question here is whether the approximation factor of 54 we have proved for the simple MapReduce problem holds also for this case. It suffices to show that there exists an optimal schedule for this version of the MapShuffleReduce problem, where the shuffle tasks are executed on the reduce processors exactly before the execution of the corresponding reduce task. We show this in the following lemma:

Lemma 5.2. *There is an optimal schedule of shuffle tasks and reduce tasks on processors of the set $\mathcal{P}_{\mathcal{R}}$ such that:*
(i) There are no idle periods.
(ii) All shuffle tasks in s_j^r are executed together and complete exactly before the reduce task $\mathcal{T}_{r,j}$ starts its execution.

Proof. Consider a feasible schedule σ. In this schedule there are three cases when idle time can occur: either between the execution of two shuffle tasks or two reduce tasks, either between a shuffle and a reduce task. In the first two cases, since there are no precedence constraints between shuffle or reduce tasks and given the fact that we assume that these types of tasks are available from time zero, skipping the idle time can only reduce the completion time of these tasks and therefore the objective value of our problem. For the third case, it suffices to notice that since the completion of each shuffle task must precede the completion of the corresponding reduce task, skipping the idle time again can only reduce the objective function of σ.

In order to prove (ii), consider a feasible schedule σ violating this condition. Consider a task $\mathcal{T}_{r,j} \in \mathcal{R}$ to be the last reduce task of a job j completed on a processor $i \in \mathcal{P}_{\mathcal{R}}$. Then, if we fix its completion time and schedule all the corresponding shuffle tasks to be executed just before $\mathcal{T}_{r,j}$ in an arbitrary order, then it is easy to see that the completion time of j remains unchanged, while the completion time of the tasks $\mathcal{T}_{r,j'} \in \mathcal{R}$ preceding $\mathcal{T}_{r,j}$ on the same processor may decrease. Thus, it follows that any feasible schedule σ of the reduce and shuffle tasks can be transformed into a schedule σ', satisfying the properties of the lemma. □

From the previous lemma we see that a schedule without idle times and with the shuffle tasks executed just before the reduce task consists a lower bound for every feasible schedule σ and thus for the optimal schedule.

Therefore, the following theorem holds:

Theorem 5.3. *There exists a 54-approximation for the MapShuffleReduce scheduling problem, when the shuffle tasks are executed on reduce processors.*

5.2.2 The Shuffle Tasks are Executed on Different Input Processors

In the second variation of the MapShuffleReduce problem we discuss, the shuffle tasks are executed on a set of different "input" processors $\mathcal{P}_{\mathcal{S}}$. When the shuffle tasks are executed on different processors, we prove that we lose only a factor of 2 in the approximation ratio of the ShuffleReduce schedule.

Lemma 5.4. *Consider two optimal schedules σ and σ' of shuffle tasks and reduce tasks on processors of the set $\mathcal{P}_{\mathcal{R}} \cup \mathcal{P}_{\mathcal{S}}$ and on processors of the set $\mathcal{P}_{\mathcal{R}}$ respectively. Let also $C^{\sigma}_{k,j}$, $C^{\sigma'}_{k,j}$ be the completion times of any reduce task $\mathcal{T}_{k,j}$ in σ and σ' respectively. Then, it holds that $C^{\sigma'}_{k,j} \leq 2C^{\sigma}_{k,j}$*

Proof. Consider an optimal schedule σ on the $\mathcal{P}_{\mathcal{R}} \cup \mathcal{P}_{\mathcal{S}}$ processors. We fix a reduce task $\mathcal{T}_{k,j} \in \mathcal{R}$ of a job j, the reduce processor $i^R \in \mathcal{P}_{\mathcal{R}}$ where it is executed on σ and the corresponding input processor $i^S \in \mathcal{P}_{\mathcal{S}}$. Let $B(k)$ the set of reduce tasks that are executed on i^R before $\mathcal{T}_{k,j}$ and $Sh(k)$ the set of shuffle tasks that are executed on i^S corresponding to the reduce tasks of the set $B(k) \cup \{\mathcal{T}_{k,j}\}$.

In σ it is the case that:

$$C^{\sigma}_{k,j} \geq \max\{ \sum_{\mathcal{T}_{k,j} \in B(k)} p_{i^R,k,j}, \sum_{\mathcal{T}_{q,l,j} \in Sh(k)} t_{q,l,j} \}$$

Now, we transform σ into a new schedule σ' by maintaining the order and assignment of reduce tasks and by scheduling the shuffle tasks of each reduce task on its reduce processor and exactly before its starting. In this schedule, for the completion time of a reduce task $\mathcal{T}_{k,j}$ it holds:

$$C^{\sigma'}_{k,j} = \sum_{\mathcal{T}_{k,j} \in B(k)} p_{i^R,k,j} + \sum_{\mathcal{T}_{q,l,j} \in Sh(k)} t_{q,l,j} \leq 2C^{\sigma}_{k,j}$$

$\square$

Using this lemma we see that we can schedule only map tasks with a $\frac{27}{2}$-approximation factor and the shuffle-reduce tasks with an approximation factor of 27. Now applying the same analysis as for **Algorithm-MR** the next theorem follows:

Theorem 5.5. *There exists a 81-approximation for the MapShuffleReduce scheduling problem, when the shuffle tasks run on independent "input" processors.*

5.3 A Greedy Heuristic

The time complexity of **Algorithm-MR** combined with **TaskScheduling** is clearly dominated by the time needed for the optimization of the linear programs. In this section we present a natural greedy heuristic not burdened by any LP solving complexity. In the following, we will refer to this heuristic as **Greedy-MR**.

In order to find a "satisfying" solution to the MapReduce problem, there are two things one must take into account. The fair load-balancing of tasks to processors and the efficient sequencing of task in every processor are both crucial aspects for the quality of a schedule. Algorithm **Greedy-MR** creates a feasible schedule by separating the load-balancing from the sequencing nature of the problem and by using known "best-effort" heuristics for the optimization of each one.

As we can see the algorithm proceeds in two basic steps: the load balancing and the sequencing. The idea in the assignment part is based on the work of Aspnes et al. "On-line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling" [54]. Using a parameter $\alpha \in (0, 1)$ for tuning the sensitivity of the assignment, each map or reduce task is assigned to the map or reduce machine respectively that minimizes the quantity $\alpha^{\Delta^b(i)+p_{i,k,j}} - \alpha^{\Delta^b(i)}$, where $b \in \{m, r\}$ and $\Delta^b(i)$ the current load of a map or reduce machine i.

Greedy-MR: Creates a fast feasible schedule for the MapReduce problem.

```
1: Fix a parameter $\alpha \in (0, 1)$
2: Take an arbitrary order $\mathcal{O}$ of the jobs.
3: Let $\Delta^b(i)$ be the current load of a $b \in \mathcal{P}_\mathcal{M}, \mathcal{P}_\mathcal{R}$ processor. In this phase all these
   variables are equal to zero.
4: for each job $j \in \mathcal{O}$ do
5:     for each task $\mathcal{T}_{k,j} \in \mathcal{M}$ of job $j$ do
6:         Assign $\mathcal{T}_{k,j}$ to the processor $i$ such that
           $i = \arg\min_{i \in \mathcal{P}_\mathcal{M}}\{\alpha^{\Delta^M(i)+p_{i,k,j}} - \alpha^{\Delta^M(i)}\}$.
7:         $\Delta^M(i) \leftarrow \Delta^M(i) + p_{i,k,j}$
8:     end for
9:     for each task $\mathcal{T}_{k,j} \in \mathcal{R}$ of job $j$ do
10:        Assign $\mathcal{T}_{k,j}$ to the processor $i$ such that $i = \arg\min_{i \in \mathcal{P}_\mathcal{R}}\{\alpha^{\Delta^R(i)+p_{i,k,j}} - \alpha^{\Delta^R(i)}\}$
11:        $\Delta^R(i) \leftarrow \Delta^R(i) + p_{i,k,j}$
12:    end for
13: end for
14: Reorder the tasks in each processor using the following rule:
15: for each job $j \in \mathcal{J}$ do
16:     Let $p_{k,j}$ be the processing time of a task on the processor it has been assigned.
17:     Define the quantity $\omega_j \leftarrow \frac{w_j}{\sum_{\mathcal{T}_{k,j} \in \mathcal{M}} p_{k,j} + \sum_{\mathcal{T}_{k,j} \in \mathcal{R}} p_{k,j}}$.
18: end for
19: for $i \in \mathcal{P}_\mathcal{M}$ do
20:     Reorder the tasks assigned to $i$ in a non-increasing order of $\omega_j$.
21: end for
22: for $i \in \mathcal{P}_\mathcal{R}$ do
23:     Reorder the tasks assigned to $i$ in a non-increasing order of $\omega_j$, with respect to
        the precedence constrains emerged by the completion of map tasks.
24: end for
```

Algorithm 5: Greedy-MR

After the assignment, the algorithm proceeds with the sequencing of tasks on each processor using a simple rule. For the tasks of job j we define the quantity $\omega_j \leftarrow \frac{w_j}{\sum_{\mathcal{T}_{k,j} \in \mathcal{M}} p_{k,j} + \sum_{\mathcal{T}_{k,j} \in \mathcal{R}} p_{k,j}}$, where $p_{k,j}$ the processing time of a map or reduce task given the assignment. We then schedule the map tasks of each map processor in a non-increasing order of ω_j. After this, we schedule the reduce tasks of each reduce processor again in non-increasing order of ω_j, taking into account at each time t only the reduce tasks that have been released. Recall that in the MapReduce setting, by "released" we refer to the tasks of jobs whose map tasks have already complete their execution. The sequencing part of **Greedy-MR** uses the same idea as the well-celebrated Smith Rule whose optimality is known for the single machine context. In this direction, we try to schedule jobs of high weight first while we schedule jobs of high accumulative processing time last.

Chapter 6

Experimental Evaluation

At this point, we present the experimental evaluation of the MapReduce scheduling algorithm we present in the previous chapter. The study of the "empirical" approximation ratio of the algorithm, i.e. how the algorithm performs on "normal" inputs, is the main contribution of this thesis. In this chapter we describe implementation issues, different models of inputs and technical information. We close this thesis with the presentation and analysis of the results.

6.1 Experimental Experience

In order to estimate the performance of **Algorithm-MR** we compare the produced objective value with the solution produced by the heuristic **Greedy-MR** as well as with a lower bound derived from an LP-relaxation for the combined case of map and reduce tasks. This comparison was performed for a fixed number of disjoint map and reduce machines and a fixed number of map and reduce tasks per job. The parameters of the problem we experiment on are the number of jobs and the distribution of processing times.

More specifically, for the experiments we consider a fixed number of 40 map and 40 reduce machines. Each job has a fixed number of 30 map tasks and 10 reduce tasks. The weight w_j of each job j is selected uniformly at random from the interval $[1, |\mathcal{J}|]$, where $|\mathcal{J}|$ the total number of jobs. We experiment for a varying number of jobs from 5 to 50 increasing the number each time by 5 jobs. For each possible configuration of the problem we run ten experiments using ten randomly generated instances with these configurations.

6.2 Lower Bound

In order to estimate the empirical approximation ratio of the two algorithms we create an LP formulation similar to the one of algorithm **TaskScheduling** but for the mixed problem of map and reduce tasks. Again, we denote as $\mathcal{L} = \{[1,1],(1,(1+\delta)],((1+\delta),(1+\delta)^2],\dots,((1+\delta)^{L-1},(1+\delta)^L]\}$, where $(1+\delta)^L$ is an upper bound to the time horizon of any possible MapReduce schedule. Recall that we denote as $\mathcal{P}_\mathcal{M}$ and $\mathcal{P}_\mathcal{R}$ the map and reduce processors respectively and with $\mathcal{M}$ and $\mathcal{R}$ the set of map and reduce tasks.

The lower bound is computed using the following LP:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j\in\mathcal{J}} w_j C_j && \\
\text{subject to:} \quad & \sum_{i\in\mathcal{P}_\mathcal{M},\ell\in\mathcal{L}} y_{i,k,j,\ell} \geq 1, && \forall T_{k,j}\in\mathcal{M} \ (1a)\\
& \sum_{i\in\mathcal{P}_\mathcal{R},\ell\in\mathcal{L}} y_{i,k,j,\ell} \geq 1, && \forall T_{k,j}\in\mathcal{R} \ (1b)\\
& C_j \geq C_{k,j}, && \forall T_{k,j}\in\mathcal{R} \ (2a)\\
& C_{k,j} \geq C_{m,j} + \sum_{i\in\mathcal{P}_\mathcal{R}}\sum_{\ell\in\mathcal{L}} p_{i,k,j} y_{i,k,j,\ell} && \forall j\in\mathcal{J}, \forall T_{k,j}\in\mathcal{R}, \forall T_{m,j}\in\mathcal{M} \ (2b)\\
& \sum_{i\in\mathcal{P}_\mathcal{M}}\sum_{\ell\in\mathcal{L}} (1+\delta)^{\ell-1} y_{i,k,j,\ell} \leq C_{k,j}, && \forall T_{k,j}\in\mathcal{M} \ (3a)\\
& \sum_{i\in\mathcal{P}_\mathcal{R}}\sum_{\ell\in\mathcal{L}} (1+\delta)^{\ell-1} y_{i,k,j,\ell} \leq C_{k,j}, && \forall T_{k,j}\in\mathcal{R} \ (3b)\\
& \sum_{T_{k,j}\in\mathcal{M}} p_{i,k,j} \sum_{t\leq\ell} y_{i,k,j,t} \leq (1+\delta)^\ell, && \forall i\in\mathcal{P}_\mathcal{M}, \ell\in\mathcal{L} \ (4a)\\
& \sum_{T_{k,j}\in\mathcal{R}} p_{i,k,j} \sum_{t\leq\ell} y_{i,k,j,t} \leq (1+\delta)^\ell, && \forall i\in\mathcal{P}_\mathcal{R}, \ell\in\mathcal{L} \ (4b)\\
& p_{i,k,j} > (1+\delta)^\ell \Rightarrow y_{i,k,j,\ell} = 0, && \forall i\in\mathcal{P}_\mathcal{M}\cup\mathcal{P}_\mathcal{R}, T_{k,j}\in\mathcal{M}\cup\mathcal{R}, \ell\in\mathcal{L} \ (5)\\
& y_{i,k,j,\ell} \geq 0, && \forall i\in\mathcal{P}_\mathcal{M}\cup\mathcal{P}_\mathcal{R}, T_{k,j}\in\mathcal{M}\cup\mathcal{R}, \ell\in\mathcal{L}
\end{aligned}$$

Clearly, this linear relaxation consists a lower bound to the MapReduce problem. The role of the constraints is the same as in the LP of the **TaskScheduling** algorithm. Again here, constraints (2a) denote that the completion time of a job is the maximum completion time of its reduce tasks while constraints (2b) denote the precedence constraints of between map and reduce tasks. The term $\sum_{i\in\mathcal{P}_\mathcal{R}}\sum_{\ell\in\mathcal{L}} p_{i,k,j} y_{i,k,j,\ell}$ added to the completion time of each map task is in fact the "squashed area" lower bound to the execution of a reduce task.

6.3 Processing Time Distributions

For the sake of the experiments, we model the processing time of tasks using two different approaches: the *Uniform* or *Uncorrelated* distribution and the *Processor-Job Correlated* distribution.

In the first uncorrelated case, the processing times $\{p_{i,k,j}\}_{i\in\mathcal{P}_\mathcal{M}}$ of the map tasks $T_{k,j} \in \mathcal{M}$ of each job $j \in \mathcal{J}$ are selected uniformly at random (u.a.r) from the interval [1,100]. In the same way, the processing times $\{p_{i,k,j}\}_{i\in\mathcal{P}_\mathcal{R}}$ of the reduce Tasks are set to thrice a value selected u.a.r from [1,100] plus some "noise" selected u.a.r from [1,10].

In order to capture the issues of data locality on machines as well as the mean processing time of tasks of different jobs the need for a more sophisticated distribution of processing times is exigent. For this reason, in the processor-job correlated case [55] the processing times $\{p_{i,k,j}\}_{i\in\mathcal{P}_\mathcal{M}}$ of the Map tasks $T_{k,j} \in \mathcal{M}$ of each job j are uniformly distributed in $[\alpha_i\beta_j, \alpha_i\beta_j + 10]$, where α_i , β_j are selected u.a.r. from $[1, 20]$, for each processor $i \in \mathcal{M}$ and each job $j \in \mathcal{J}$ respectively. As before, the processing time of each reduce task is set to three times a value selected u.a.r. from $[\alpha_i\beta_j, \alpha_i\beta_j + 10]$ plus some "noise" selected u.a.r from $[1, 10]$.

Note that in both cases the rule of thumb of three times more processing time requirement on average for the reduce tasks is based on the model of Chang et al. in "Scheduling in mapreduce-like systems for fast completion time" [21].

6.4 Shuffle Tasks

Apart from the two aforementioned processing time distributions, we include in our experiments a third set of benchmarks, modelling the existence of shuffle tasks. In this case, the processing times of map and reduce tasks follow the processor-job correlated distribution while for each possible pair of map and reduce task of a job j there exists a shuffle task with specific transfer time. This processing time of the shuffle task is equal to $\frac{20}{3}\beta_j$. Recall that β_j is a job-specific value selected u.a.r from $[1, 20]$. In our experiments we examine the case where shuffle tasks are executed on the same reduce processor as the corresponding reduce task and exactly before its execution. For this reason, we compare the performance of algorithms **Algorithm-MR** and **Greedy-MR** with the lower bound, after we increase the processing time of each reduce task by the sum of processing times of the shuffle tasks.

6.5 Implementation

In the following, we describe in a few words the implementation of the algorithms we compare, **Algorithm-MR** and **Greedy-MR**.

As we can see in figure 6.1 for the **Algorithm-MR**, the "main" script begins with the input reading and the initialization of variables. After this, the **TaskScheduling** subroutine is called two times, one for the map and one for the reduce tasks of the instance. Given the two produced partial schedules, the script calls the **Merge** subroutine in order to combine the two schedules.

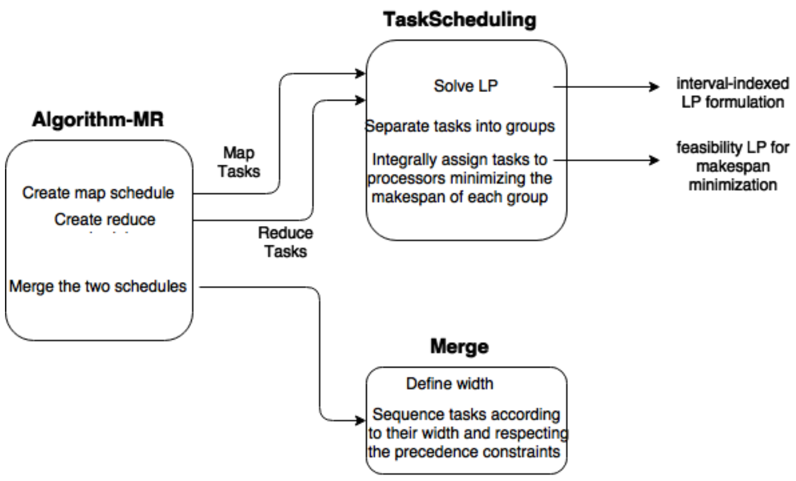
TaskScheduling
Solve LP
Separate tasks into groups
Integrally assign tasks to processors minimizing the makespan of each group
interval-indexed LP formulation
feasibility LP for makespan minimization
Algorithm-MR
Create map schedule
Create reduce
Merge the two schedules
Map Tasks
Reduce Tasks
Merge
Define width
Sequence tasks according to their width and respecting the precedence constraints

FIGURE 6.1: Implementation sketch of **Algorithm-MR**.

The **TaskScheduling** subroutine creates schedules for only map or only reduce tasks on map and reduce processors respectively. In this direction, as we have described in previous chapter, algorithm **TaskScheduling** solves the corresponding LP with exponentially growing time intervals. Given this fractional solution, the algorithm separates tasks into groups with respect to their completion times. The last step of this subroutine, is to integrally assign the tasks of each group on the processors trying to minimize the makespan of every group. For this reason, **TaskScheduling** uses the algorithm for makespan minimization on unrelated machines we have described in a previous chapter. Note that, for the tasks of each group, the algorithm performs a binary search over the possible makespan, starting with the upper bound provided by the theoretical analysis of this algorithm.

Given the two produced schedules, the **Merge** subroutine, greedily merges the two schedules into one, with the use of the width of each job, as described in the previous chapter.

The implementation of **Greedy-MR** follows in general the same basic phases, separating completely this time the assignment of tasks to processors and the sequencing of the assigned tasks of each processor. As we can see in figure 6.2, the assignment of tasks on processors is performed by the subroutine **Greedy Assign**, using the simple rule we described in the previous chapter. After the creation of the two assignment schedules for the map and the reduce tasks, the subroutine **Generalized Smith Merge** merges the two schedules using the weight to total processing time ratio for sequencing and respecting the precedence constraints between map and reduce tasks of each job.

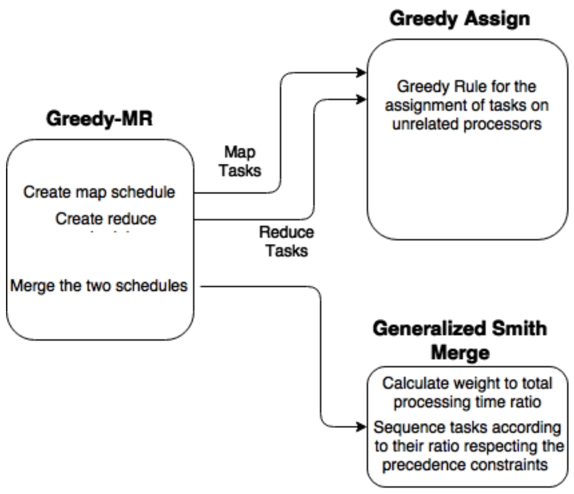
Greedy-MR
Create map schedule
Create reduce
Merge the two schedules
Map Tasks
Reduce Tasks
Greedy Assign
Greedy Rule for the assignment of tasks on unrelated processors
Generalized Smith Merge
Calculate weight to total processing time ratio
Sequence tasks according to their ratio respecting the precedence constraints

FIGURE 6.2: Implementation sketch of **Greedy-MR**.

6.6 Technical Information

The algorithms **Algorithm-MR**, **Greedy-MR** as well as the lower bound were implemented using Python 2.7. The solver used for the linear programs was Gurobi Optimizer 6.0. The experiments were performed on a machine with 4 packages (Intel(R) Xeon(R) E5- 4620 @ 2.20GHz) of 8 cores each (16 threads with hyperthreading) and a total memory of 256 GB. The operating system was a Debian GNU/Linux 6.0. The used scripts as well as the benchmarks of the results are available at: `http://www.corelab.ntua.gr/~opapadig/mrexperiments/` .

6.7 Results

In the following, we present the results of the experiments. In all cases, we present scatter graphs of the objective values produced by all our algorithms as well as the empirical approximation ratios in all trials. Furthermore, for each number of job we present graphically the average value of objective values for all trials as well as the average approximation ratios.

6.7.1 Uncorrelated Input

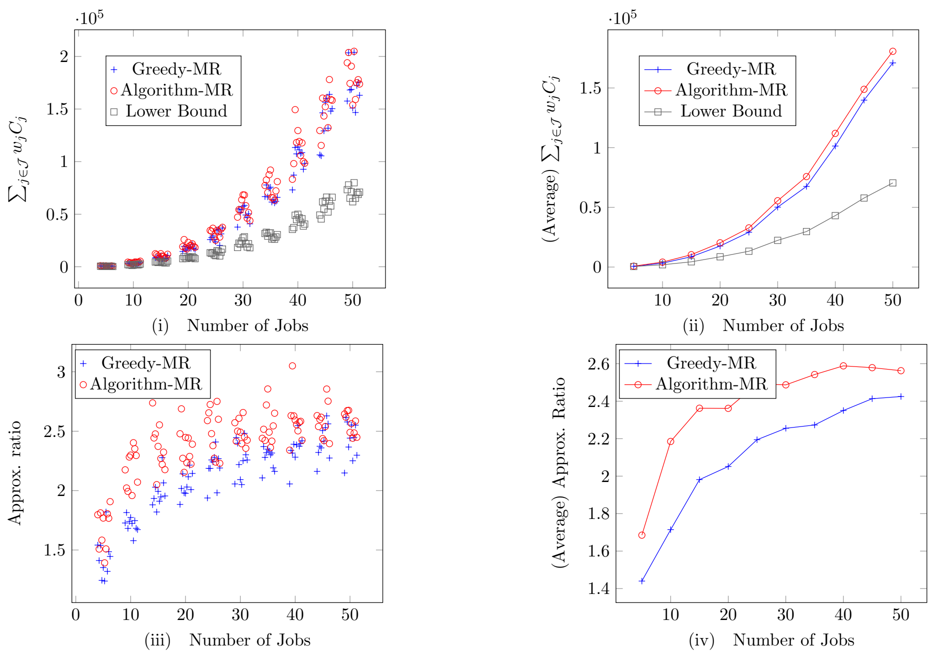

Figure 6.3: Uncorrelated Results

Comparing (i)-(ii) solutions of **Greedy-MR** with **Algorithm-MR** and a lower bound on the optimal cost, and (iii)-(iv) (empirical) approximation ratios of **Greedy-MR** and **Algorithm-MR**, for uncorrelated tasks' processing times.

By figure 6.3 (i)-(ii) we note that **Greedy-MR** performs quite better than **Algorithm-MR** in general. For number of jobs $n \leq 10$, **Greedy-MR** gives up to 21% better solutions on average. However, as the number of jobs increases, the gap between **Greedy-MR** and **Algorithm-MR** is shrinking, e.g., for $n = 45$ and $n = 50$ **Greedy-MR** gives 6% and 5% (on average) better solutions, respectively. In terms of performance guarantee, as we can see in figure 6.3 (iii)-(iv) the (empirical) approximation ratio of **Algorithm-MR** ranges from 1.68 to 2.58 (on average), while the approximation ratio of **Greedy-MR** ranges from 1.43 to 2.42 (on average). Clearly, both algorithms are far away from **Algorithm-MR**'s approximation guarantee of 54.

6.7.2 Job-Processor Correlated Input

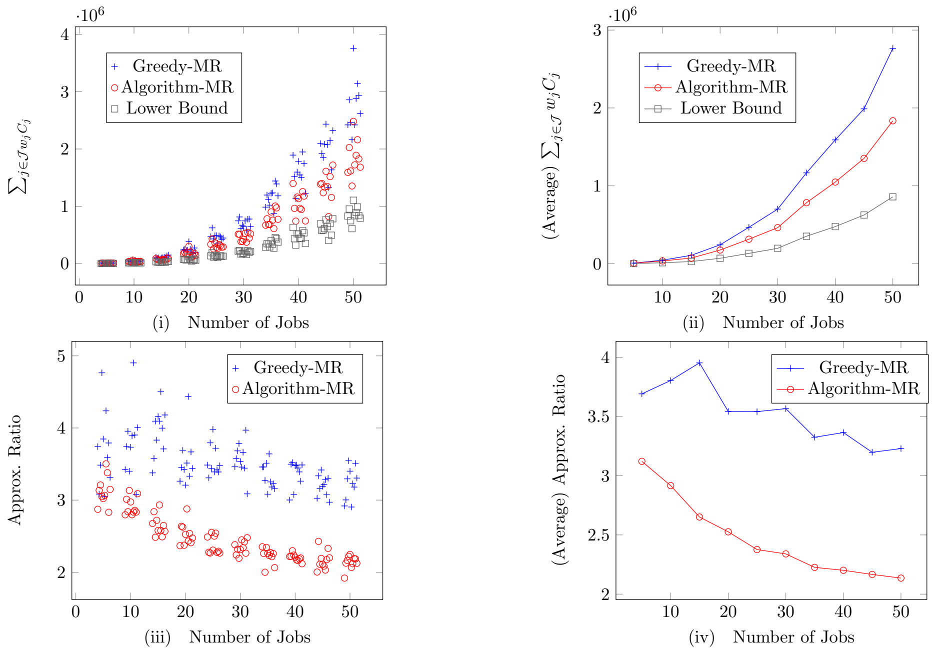

FIGURE 6.4: Processor-Job Correlated Results

Comparing (i)-(ii) solutions of **Greedy-MR** with **Algorithm-MR** and a lower bound on the optimal cost, and (iii)-(iv) (empirical) approximation ratios of **Greedy-MR** and **Algorithm-MR**, for processor-job correlated tasks' processing times.

By figure 6.4 (i)-(ii) it is clear that **Algorithm-MR** outperforms **Greedy-MR** for all different values of n. More specifically, **Algorithm-MR** leads to $11\% - 34\%$ (on average) smaller values of the objective function, compared to **Greedy-MR**. This is mainly due the fact that by generating processor-job correlated tasks' processing times the assignment and sequencing procedure becomes more sophisticated. So, both the online assignment and the common WSPT policy, are not quite efficient; actually, even when there is a small number of jobs, $n = 5$, **Algorithm-MR** gives on average 11% (on average) smaller solutions. The approximation ratio of **Algorithm-MR**, in figure 6.4 (iii)-(iv), ranges from 2.13 to 3.12 (on average), while, for **Greedy-MR**, the approximation ratio ranges from 3.19 to 3.95 (on average). Again, both algorithms are very far from **Algorithm-MR**'s approximation guarantee of 54. Furthermore, it is important to note that **Algorithm-MR** improves its performance guarantee as the input becomes more and more involved (for $n \geq 40$), while simultaneously produces better solutions (of more than 30%) than **Greedy-MR**.

6.7.3 Job-Processor Correlated Input with Shuffle Tasks

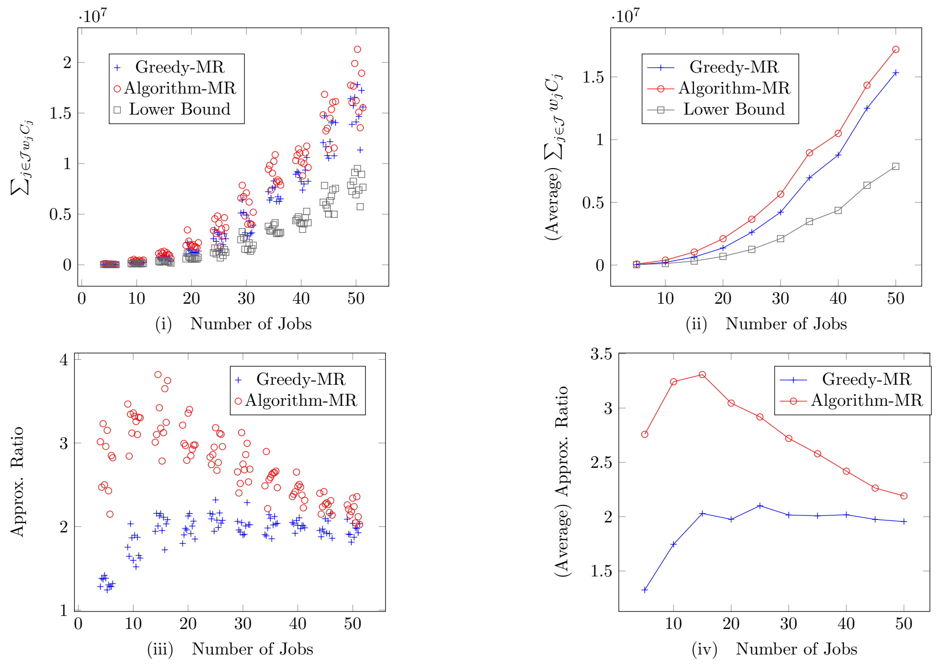

FIGURE 6.5: Data Shuffle Results
Comparing (i)-(ii) solutions of **Greedy-MR** with **Algorithm-MR** and a lower bound on the optimal cost, and (iii)-(iv) (empirical) approximation ratios of **Greedy-MR** and **Algorithm-MR**, for processor-job correlated tasks' processing times including shuffle tasks.

By Fig. 6.5 (i)-(ii) it is clear that **Greedy-MR** is better than **Algorithm-MR** for all different values of n. More specifically, **Greedy-MR** leads to $9\% - 60\%$ smaller values of the objective function, compared to **Algorithm-MR**. However, as the number of jobs is increasing the gap between the performance of the two algorithms is shrinking. For example for $n = 5$, **Greedy-MR** produced a 60% better solution, while for $n = 50$ the percentage falls to 10% on average, with some trials where the **Algorithm-MR** appears to give slight better solutions. The approximation ratio of **Algorithm-MR**, in figure 6.5 (iii)-(iv), ranges from 2.02 to 3.81, while, for **Greedy-MR**, the approximation ratio ranges from 1.24 to 2.31. In this case, both algorithms are again very far from **Algorithm-MR**'s approximation guarantee of 54. Moreover, it is important to note that both algorithms' empirical ratios converge to 2 as the number of jobs is increasing.

6.8 Evaluation

Considering the results of the previous section, one can see that the performance of **Algorithm-MR** and **Greedy-MR** depends highly on the type of processing time distribution, on the number of jobs as well as on the existence or not of shuffle tasks.

More specifically, we see that in the uncorrelated case the performance of **Greedy-MR** is slightly better. This can be explained and attributed to the distribution of processing times. In the uniform case we can see that for all jobs, the estimated mean processing time is constant. In other words, given that the processing times are selected u.a.r from $[0, 100]$, the probability for a task to have processing time p is equal to the probability to have $100 - p$. This "rough" smooth analysis suggests that for the load balancing part of **Greedy-MR**, the processors may behave in the mean case as identical. Moreover, in this case where there are not many "anomalies" in the load balancing part, the Weighted Shortest Processing Time First rule we apply for sequencing is known for behaving well on identical processors.

The previous arguments about the performance of **Greedy-MR** in the uncorrelated case are reinforced by the experimental results on the job-processor correlated case. In this case, since many processors may have advantage over others for all jobs, the load balancing part of **Greedy-MR** in not working very well this time. The relatively sophisticated load balancing of **Algorithm-MR** seems now to outperform that of **Greedy-MR**.

With the introduction of shuffle tasks in our experiments we see that the **Greedy-MR** gives, again, much better solutions than **Algorithm-MR**. This fact can also be attributed to the distribution of processing times. In this case, since the processing times of shuffle tasks do not depend on the corresponding machines and due to the fact that these tasks are executed on the reduce processors, two conclusions can be deduced. The first is that the total contribution of the map schedule to the objective becomes negligible given the severe enlargement of the reduce tasks. The second is that, in this case, since by definition, the shuffle tasks' processing times depend only on β_j and therefore are constant for a fixed job j, then the processing times of the extended reduce tasks behave also as constant. As we can see, the load balancing of constant size tasks is the easiest case so-far for by the load balancing part of **Greedy-MR**. This fact, together with the Weighted Shortest Processing Time First rule, can explain the good performance of **Greedy-MR** over **Algorithm-MR**.

Conclusion

The theory as well as the experimentation have been proven fundamental tools for the evaluation of MapReduce scheduling algorithms. From linear programming relaxations and greedy rules to randomized approaches and graph theoretic reductions are some of the ways theory produces algorithms with various time complexities and approximation guarantees. It is the experimentation, however, that shows us how close or far from the reality these guarantees are and whether the trade-off between speed and optimality is worth considering. What we have seen in this work, is that an approximation ratio proven using the scheduling theory may be partially misleading given the performance of the algorithm in practice.

More specifically, we have shown that the algorithm for scheduling MapReduce tasks on unrelated processors minimizing the total weighted completion time performs quite well for normal and "rational" inputs. Indeed, it produces schedules with objective values close to the optimal, despite the proven worst-case approximation factor of 54. The same conclusion holds for case where we model the intermediate data exchange between map and reduce tasks. Moreover, we have seen that a really fast and simple heuristic for the same problem, despite its probably non-constant approximation ratio, may also produce schedules with noteworthy objective values, even better than its sophisticated competitor for specific instances.

In the pursuit of optimality and efficient use of the available hardware, the model of scheduling MapReduce jobs can be extended in various directions. One of them is the introduction of malleable MapReduce jobs. In this case, the scheduler, apart from the assignment and sequencing of tasks to processors, has the ability to decide the so-called "grain size" i.e. the number of tasks where the total volume of work is going to be divided into. Another possible extension is to study the effect of the natural topology of processors on the cost of data transferring between tasks. More specifically, in non-uniform memory access (NUMA) architectures the communication time between two tasks may be crucially affected from the location of the nodes where these tasks are executed on.

As we have seen, massive parallelism is no more the future; it is now a reality relentlessly producing problems and demanding results and solutions: both theoretical and empirical.

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782.

[2] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.

[3] apache.org. `http://apache.org`. Accessed: 2015-06-12.

[4] discoproject.org. `http://discoproject.org/`. Accessed: 2015-06-12.

[5] infinispan.org. `http://infinispan.org`. Accessed: 2015-06-12.

[6] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009. ISBN 978-1-60558-752-3.

[7] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, 2010. ISBN 978-1-60558-577-2.

[8] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[9] Thomas Sandholm and Kevin Lai. Mapreduce optimization using regulated dynamic prioritization. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 299–310, 2009. ISBN 978-1-60558-511-6.

[10] Dongjin Yoo and Kwang Mong Sim. A comparative review of job scheduling for mapreduce. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 353–358, Sept 2011.

[11] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey balmin. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 978-3-642-16954-0.

[12] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Trans. Algorithms*, 6(4):66:1–66:19, September 2010. ISSN 1549-6325.

[13] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, 2010. ISBN 978-0-898716-98-6.

[14] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *CoRR*, abs/1206.4377, 2012.

[15] Jeffrey D. Ullman. Designing good mapreduce algorithms. *XRDS*, 19(1):30–34, September 2012. ISSN 1528-4972.

[16] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 231–240, 2010. ISBN 978-1-60558-799-8.

[17] Cheng tao Chu, Sang K. Kim, Yi an Lin, Yuanyuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y. Ng. Map-reduce for machine learning on multicore. In B. Schölkopf, J.C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2007.

[18] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, 2010. ISBN 978-1-60558-945-9.

[19] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec. Hadi: Fast diameter estimation and mining in massive graphs with hadoop. *CMU-ML*, 2008.

[20] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010. ISBN 1608453421, 9781608453429.

[21] Hyunseok Chang, M. Kodialam, R.R. Kompella, T.V. Lakshman, Myungjin Lee, and S. Mukherjee. Scheduling in mapreduce-like systems for fast completion time. In *INFOCOM, 2011 Proceedings IEEE*, pages 3074–3082, April 2011.

[22] Monaldo Mastrolilli, Maurice Queyranne, Andreas S. Schulz, Ola Svensson, and Nelson A. Uhan. Minimizing the sum of weighted completion times in a concurrent open shop. *Oper. Res. Lett.*, 38(5):390–395, September 2010. ISSN 0167-6377.

[23] Fangfei Chen, Murali S. Kodialam, and T. V. Lakshman. Joint scheduling of processing and shuffle phases in mapreduce systems. In *INFOCOM'12*, pages 1143–1151, 2012.

[24] Benjamin Moseley, Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. On scheduling in map-reduce and flow-shops. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 289–298, 2011. ISBN 978-1-4503-0743-7.

[25] CarlosD. Paternina-Arboleda, JairoR. Montoya-Torres, MiltonJ. Acero-Dominguez, and MariaC. Herrera-Hernandez. Scheduling jobs on a k-stage flexible flow-shop. *Annals of Operations Research*, 164(1):29–40, 2008. ISSN 0254-5330. doi: 10.1007/s10479-007-0257-2. URL `http://dx.doi.org/10.1007/s10479-007-0257-2`.

[26] Petra Schuurman and Gerhard J. Woeginger. A polynomial time approximation scheme for the two-stage multiprocessor flow shop problem. *Theoretical Computer Science*, 237(1–2):105 – 122, 2000. ISSN 0304-3975. URL `http://www.sciencedirect.com/science/article/pii/S0304397598001571`.

[27] Dimitris Fotakis, Ioannis Milis, Orestis Papadigenopoulos, Emmanouil Zampetakis, and Georgios Zois. Scheduling mapreduce jobs and data shuffle on unrelated processors. *CoRR*, abs/1312.4203, 2013. URL `http://arxiv.org/abs/1312.4203`.

[28] Top500.org. `http://top500.org`. Accessed: 2015-06-11.

[29] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[30] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30 (3)*, 2005. URL `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[31] Parallel processing systems, lecture notes, cslab ece ntua. URL `http://www.cslab.ece.ntua.gr/courses/pps`.

[32] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

[33] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979. URL `http://www.sciencedirect.com/science/article/pii/S016750600870356X`.

[34] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, July 2000. ISSN 0004-5411.

[35] David Karger, Cliff Stein, and Joel Wein. Algorithms and theory of computation handbook. chapter Scheduling Algorithms, pages 20–20. Chapman & Hall/CRC, 2010. ISBN 978-1-58488-820-8. URL `http://dl.acm.org/citation.cfm?id=1882723.1882743`.

[36] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008. ISBN 0387789340, 9780387789347.

[37] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 2001. ISBN 3540415106.

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

[39] G.H. Hardy, J.E. Littlewood, and G. Pólya. *Inequalities*. Cambridge Mathematical Library. Cambridge University Press, 1952. ISBN 9780521358804.

[40] Leslie A. Hall, Andreas S. Schulz, David B. Shmoys, and Joel Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Math. Oper. Res.*, 22(3):513–544, August 1997. ISSN 0364-765X.

[41] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011. ISBN 0521195276, 9780521195270.

[42] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46(3):259–271, February 1990. ISSN 0025-5610.

[43] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[44] José R. Correa, Martin Skutella, and José Verschae. The power of preemption on unrelated machines and applications to scheduling orders. *Math. Oper. Res.*, 37(2): 379–398, 2012.

[45] H. Karloff. *Linear Programming.* Progress in Computer Science and Applied Series. Birkhäuser, 1991. ISBN 9783764335618. URL `https://books.google.gr/books?id=XYfvf5sCx3gC`.

[46] Christos M. Papadimitriou. *Computational complexity.* Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0201530821.

[47] Vijay V. Vazirani. *Approximation Algorithms.* Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 3-540-65367-8.

[48] Mikhail J. Atallah and Susan Fox, editors. *Algorithms and Theory of Computation Handbook.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1998. ISBN 0849326494.

[49] S. M. Johnson. Optimal Two- and Three-stage Production Schedules with Setup Times Included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

[50] Naveen Garg, Chaitanya Swamy, and Sachin Jain. A randomized algorithm for flow shop scheduling.

[51] Martin Skutella. Convex quadratic and semidefinite programming relaxations in scheduling. *J. ACM*, 48(2):206–242, March 2001. ISSN 0004-5411.

[52] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[53] Jivitej S. Chadha, Naveen Garg, Amit Kumar, and V. N. Muralidhara. A competitive algorithm for minimizing weighted flow time on unrelatedmachines with speed augmentation. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 679–684, 2009. ISBN 978-1-60558-506-2.

[54] James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *J. ACM*, 44(3):486–504, May 1997. ISSN 0004-5411.

[55] A. M. A. Hariri and C. N. Potts. Heuristics for scheduling unrelated parallel machines. *Comput. Oper. Res.*, 18(3):323–331, March 1991. ISSN 0305-0548.

[56] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Math. Program.*, 62(3):461–474, December 1993. ISSN 0025-5610.

[57] Jyh-Han Lin and Jeffrey Scott Vitter. e-approximations with minimum packing constraint violation (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 771–782, 1992. ISBN 0-89791-511-9.