On parallel computing for stochastic optimization models and algorithms

Unai Aldasoro Marcellan

Supervisors:
Gloria Pérez Sainz de Rozas and María Merino Maestre

Leioa, 2014

eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

Facultad de Ciencia y Tecnología
Departamento de Matemática Aplicada y Estadística e Investigación Operativa

Servicio Editorial de la Universidad del País Vasco (UPV/EHU)
Euskal Herriko Unibertsitateko Argitalpen Zerbitzua (UPV/EHU)
University of the Basque Country - Editorial Service (UPV/EHU)
ISBN: 978-84-9082-055-1

gure osaba Iñigo eta Enriqueri
to my uncles Iñigo and Enrique

Eguzkiak urtzen du goian	High on the peaks
gailurretako elurra,	the sun melts the snow
uharka da jausten ibarrera	gushing down to the valley
geldigaitza den oldarra.	an unstoppable force.

J. Artze & M. Laboa, *Martxa baten lehen notak*, First chords of a march

Acknowledgments

First and foremost I wish to thank my supervisors Gloria and María. Your dynamism, energy and reflexive attitude have been the greatest pleasure of this thesis. Without your critical view and support this research would have not been possible.

I offer my gratitude to Professor Laureano F. Escudero for the encouragement, insightful comments and all his advice. I would also like to thank Juan F. Monge for the generous research collaboration and Araceli Garín, Larraitz Aranburu and Aitziber Unzueta from the *Grupo de Optimización Estocástica* (GOE) research group for their warm welcome.

My sincerest thanks are owed to Professor Jacek Gondzio for the opportunity to work in the University of Edinburgh as visiting Ph.D. student and share research with the Edinburgh Research Group in Optimization (ERGO).

I would like to acknowledge the financial, academic and technical support of the University of the Basque Country, the Basque Government by the IT-567-13 project and also to the Ministry of Economy and Competitiveness by the MTM2012-31514 project that have partially financed my participations in conferences. The scientific computing service, SGI/IZO-SGIker from UPV/EHU, has been indispensable, specially the magnificent support by Txema Mercero and Edu Ogando.

I also thank the departments of *Matemática Aplicada y Estadística e Investigación Operativa*, for providing me with the necessary material for my research, *Máquinas y Motores Térmicos* and *Matemática Aplicada* for the excellent atmosphere during the time I spent with them.

Lagunei, beti hor egoteagatik eta bide luze hau egiten laguntzeagatik. Suziritarrak, albokalariak, interrailekoak, Toulousedarrak, zeharlanekoak, Garrotxakoak, Amara Zaharreko lagun zaharrak, Imanol, mila esker.

A mi familia, por el esfuerzo de tantos años para que me pueda dedicar a lo que más me gusta; este trabajo es vuestro. Gure aiton-amonei, irakatsi didazuen guztiagatik, amari eta aitari, naizena zor dizuet. Esker mila gure arreba Iratiri, alaitasun eta bizi-poza. Nire azken irrifarra Nahiarentzat, uneoro alboan izateagatik, dena errazago egiteagatik.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Machines take me by surprise with great frequency.
Alan Turing

1.1 Aims of the work

This thesis aims to connect Stochastic Optimization (SO) and Parallel Computing (PC) in order to solve large scale optimization problems under uncertainty. The proposed approach links mathematical perspective, by developing decomposition algorithms, and a computer science environment, by using parallel computing resources and implementations. Depending on the chosen objective, this connection allows problems to be solved faster, larger problems to be solved or/and better results to be obtained. A new paradigm for developing decomposition algorithms is therefore proposed.

Therefore, the proposal in this memoir not only executes serial algorithm steps in parallel, but more interestingly also adapts and extends serial algorithms to be computationally more efficient in a parallel computing environment. This is achieved by allowing deeper decomposition of the original problem, by sharing the feasibility area, by creating synchronization and communication phases among parallel executions or by defining divergent initial conditions.

Consequently, parallel computing approaches are presented for exact algorithms as well as for *matheuristic* ones (the interoperation of metaheuristic methodologies and Mathematical Programming). The scalability and performance of each implementation are analysed using four problem testbeds of different scales: small, medium, large and very large, respectively. Around 150 problems have been tested, where the largest problem has 53.9 million variables (15.4 million of which are binary ones) and 57.8 million constraints.

Finally, this research aims to contribute to the discipline related to parallel computing-conceived optimization algorithms where important domains of Mathematical Optimization could interact with a parallel computing environment, i.e., multiple aspects of the problem are exploited in parallel such as cuts, decomposition, heuristic search and so on.

1.2 Overview and contributions

The basic concepts of Parallel Computing and Stochastic Optimization are presented in this chapter. This memoir introduces the *path* concept that links mathematical algorithms with parallel computing executions, allowing more complex parallel environments to be defined in terms of designing dynamic algorithms, building hybrid algorithms or searching for multiple simultaneous objectives. Additionally, a state-of-the-art description of parallel computing algorithms being designed and implemented to solve stochastic mixed 0-1 multistage problems is included.

Chapter 2 considers the main ideas of Parallel Computing applied to Stochastic Optimization from two perspectives: solving several small-scale subproblems of large-scale models in parallel, leading to the so-named *inner parallelization* being defined, and solving a single mixed 0-1 problem by parallel branching, creating a so-named *outer parallelization* environment. These paradigms are applied in the next chapters to decomposition algorithms due to their flexibility in terms of possible model and task division, obtaining a remarkable speed-up and efficiency, see Aldasoro *et al.* [2012].

Chapter 3 presents three exact Parallel Branch-and-Fix Coordination (P-BFC) algorithms: Inner P-BFC (that reproduces the serial steps and performs tasks in parallel, when possible), Outer P-BFC (based on simultaneous execution of paths) and a combined Outer-Inner P-BFC as an innovative approach to solve medium-scale mixed 0-1 optimization problems by parallel branching and simultaneous subproblem solving. The proposed approaches obtain the optimal solution significantly faster than the serial BFC and solve problems that the state-of-the-art optimization engine cannot prove optimality, see Aldasoro *et al.* [2013a].

Chapter 4 introduces a version of the matheuristic decomposition algorithm Stochastic Dynamic Programming, that allows large and very large-scale stochastic problems to be solved. This research also contributes to parallel computing to obtain tight bounds in very large-scale problems where the state-of-the-art optimization engine cannot prove optimality, and for the largest instances, where that engine cannot obtain any feasible solution. To do so, a Parallel Stochastic Dynamic

Programming (P-SDP) approach is proposed in two paradigms: Inner P-SDP and Outer P-SDP. A production planning problem has been used as a testbed for the broad computational experience that is reported in the chapter, showing that the ad-hoc P-SDP implementations enable not only a faster solution to be obtained but also a tighter bound than the serial SDP, see Aldasoro *et al.* [2014].

Chapter 5 presents some perspectives on solving large-scale problems. The chapter shows that the deeper algorithmic understanding resulting from a parallel implementation leads to an improvement of the serial algorithm. Thus, the so-named *Dynamically guided and stage ordered* Branch-and-Fix-Coordination (D-BFC) improves the use of the stochastic tree structure. Additionally, several matheuristic versions of the D-BFC algorithm are presented, based on the relaxation of some steps of the exact algorithm, in order to solve very large-scale problems. See Aldasoro *et al.* [2013b] for a detailed description of the model ordering improvements

Finally, Chapter 6 summarizes the conclusions drawn from this research and outlines future research lines regarding the parallel computing approach of decomposition algorithms, such as massively decomposition algorithms solved by Graphic Processing Units (GPUs), data decomposition, simultaneous risk averse analysis and, in general, new perspectives on designing algorithms.

1.3 Basic concepts of Parallel Computing

Let us define several basic concepts of parallel computing:

- A **computer cluster** consists of multiple computers with software that allows the group to be viewed as a single system.
- A **computing node** is a single computer system (motherboard, one or more processors, memory, network interface) within a cluster.
- A **Central Processing Unit (CPU)**, commonly referred to as the **processor**, is the piece of hardware within a computing node that carries out the instructions of a computer program, such as the basic arithmetical, logical, and input/output operations. A computer (or computing node) can have more than one processor (CPU).
- A **core** is a physical processing unit within a CPU chip that reads and executes program instructions. Currently multi-core processors (CPUs) are widely used. Core-to-core communication is significantly faster than processor-to-processor communication.

- A **thread** is a single line of commands that is active in a core. Generally, a processor can only work on one thread per core. However, state-of-the art processors allow multithreading inside a core, i.e., core level parallelization.

Therefore, a thread is the most elemental processing unit and is used throughout the work to denote an independent line of commands execution.

In parallel computing the modeler can assign different tasks to the available threads, define a hierarchy, create thread groups or subgroups and assign communication environments. Let us present the naming convention used in this research regarding the programmer defined thread management:

The thread-algorithm link is considered as follows:

- A **path** is an independent execution of an algorithm under a given set of initial conditions. In a multiple path execution environment, the behaviour of a path may change depending on the results obtained by other paths. A serial algorithm execution can be considered as one-path.

The thread hierarchy is as follows:

- A **main thread** executes a copy of the implemented program and it is in charge of managing a path. The set of main threads is called the **main thread group**, with its cardinality being equal to the number of paths. The main thread performs the non-parallelized steps of an algorithm execution.

- A **task thread** executes a copy of the implemented problem and it is assigned to perform the parallelizable tasks of a path. Note that main threads are also task threads. A **task thread group** is composed of a main thread and its associated additional task threads.

- Any thread that works on a subproblem solving process is an **auxiliary thread**. In addition to main thread and tasks threads (which are also auxiliary threads), the state-of-the-art optimization engine of choice may use additional auxiliary threads for its internal parallelization. Therefore, an **auxiliary thread group** is composed of a main thread, a set of task threads and the auxiliary threads associated to each task thread. The solver-driven auxiliary threads are not linked to a copy of the implemented program and, therefore, are not directly controlled by the user.

- The **thread assignation** of a parallel environment is denoted as $(a \times b \times h)$, where a is the number of main threads (one for each path), b is the number of task threads associated to each main thread (including itself) and h is the number of auxiliary threads associated to each task thread (including itself).

Additionally, task threads can be divided into subgroups:

- A **task thread subgroup** is a subgroup of task threads within a task group.
- A **coordinator task thread** is a task thread that manages a task thread subgroup and leads the communication.
- A **subordinated task thread** is a task thread inside a task thread subgroup that performs the steps required by the corresponding coordinator task thread.

Four communication environments can be considered:

- The **global communication** environment allows the information exchange among all threads that execute a copy of the program, i.e., main threads and task threads.
- The **primary communication** environment allows the information exchange within the main group.
- The **secondary communication** environment allows the information exchange within a task group, i.e., among a main thread and its associated task threads.
- The **tertiary communication** environment allows the information exchange within a task thread subgroup.

The execution coordination can be performed in two ways:

- A **synchronized execution** implies that all task thread groups are always at the same algorithmic point.
- An **asynchronized execution** implies that all task thread groups are not necessarily always at the same algorithmic point, i.e., the executions of different groups may diverge.

1.4 A brief state-of-the-art on Parallel Computing

Parallel Computing is a computer science area that studies the necessary hardware and software aspects to perform simultaneous execution of tasks. Currently, this discipline is the prominent paradigm in high performance computing and also in computer architecture, due to the industry's shift to multicore processors at hardware level. That type of platform embraces architecture, algorithms and methods that brings an otherwise unreachable computing power under the programmer's control.

At the hardware level, the parallel computing era started in the late 1950s in the form of shared memory multiprocessor supercomputers. The development of this type of computers continued until the early 1980s when a new paradigm of massively parallel multiprocessors (MPPs) arrived. MPPs wee seen to perform significantly better and became dominant in high performance computing. In the late 1980s computing clustering technology emerged when a large numbers of off-the-shelf computers connected by an off-the-shelf network were linked. Nowadays, parallel computing is mainly based on clusters and multicore processors. Every six months, the *Top 500* website gathers the information on the best supercomputing sites globally http://www.top500.org/. For more in-depth information on parallel computing hardware, see Culler *et al.* [1997]; Hennessy and Patterson [2003], among others.

The cooperation among processors differ depending on the way processors exchange information. The basic parallel architectures correspond to *shared memory* and *distributed memory* (managed by message-passing). As described in Linderoth [1998] "one processor of a shared memory machine can communicate with another by writing the information into a global shared memory location and having the second processor read directly from that location" using a bus. So, communication is carried out by *shared variables*. This makes inter-processor communication very easy and fast, but it has the drawback of simultaneous access to a unique memory location. On the other hand, each processor on the distributed memory paradigm has its own local memory connected by a network with the others, so, the communication is based on message-passing.

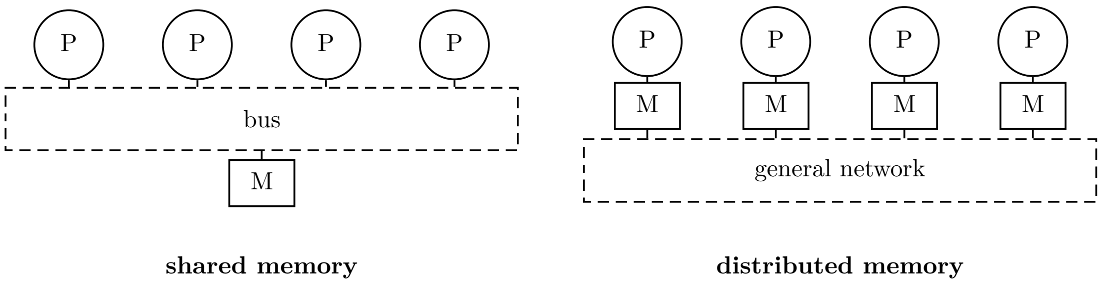

Figure 1.1: Parallel architectures (P: processor, M: memory)

Regarding mathematical optimization, we often find large-scale problems or we need to solve many of them, which cannot be performed efficiently by a single processor, even if computer science technology is constantly improving. Parallel

Computing is a powerful tool to manage this type of problems, and operate on the principle that large problems can often be divided into smaller ones. The idea is simple: to use p processors in cooperation in order to (ideally) be able to solve a problem p times faster or to solve a p times larger problem using the same amount of computing time.

1.5 Computing resources

The computing experiments in this research were conducted at the ARINA computational cluster at SGI/IZO-SGIker from Universidad del País Vasco/Euskal Herriko Unibertsitatea, UPV/EHU. During the research for this thesis ARINA provided 147 nodes connected by an QDR infiniband network with high *bandwidth* (maximum rate of successful messages delivered over a communication channel) and low *latency* (time interval between signal and response) and a 22 Tb high performance Lustre based file system for data storage. The 147 nodes and 1400 cores were divided as follows: 147 Intel Xeon processor nodes (1112 cores in overall), 30 Intel Itanium processor nodes (248 cores in overall) and 5 AMD Opteron processor nodes (40 cores in overall).

Table 1.1 summarizes the characteristics of the ARINA nodes used for the computational experience considered here. The headings are as follows: *Type*, processor brand in the computing node; *Proc.*, number of processors per computing node; *Core*, number of cores per computing node; *Frequency*, CPU clock rate; *RAM*, Random-access memory; *Disc*, hard disk data storage capacity; *max.use*, maximum resources of that type used in a single execution, where N indicates number of nodes and C number of cores; and finally *Ch.* indicates the chapters of the thesis in which each resources are used.

Table 1.1: Computational experience ARINA nodes

Type	**Proc.**	**Cores**	**Frequency**	**RAM**	**Disc**	**max. use**	**Ch.**
Intel Xeon	2	8	2.3 GHz	48 Gb	250 Gb	16N, 128C	2-3-5
Intel Xeon	2	12	2.4 GHz	48 Gb	250 Gb	8N, 96C	4

The algorithms in Chapter 2, Chapter 3 and Chapter 5 are implemented in a `C++` experimental code, whereas they are implemented in a `C` experimental code in Chapter 4. The state-of-the-art LP/MIP solver CPLEX (versions V12.2 and V12.5) called from the open source library COIN-OR (versions V1.3.1 and V1.6.0) is used as optimization engine. The *IntelMPI* library is used as the Message Passing Interface (MPI) standard for the message-passing communication in parallel environment.

1.6 A brief state-of-the-art on Stochastic Optimization

Stochastic Optimization (SO) is actually one of the most reliable tools for decision-making. Since the 1950s, it has been well known that traditional deterministic optimization is not appropriate for capturing the uncertain behaviour present in most real world applications. Beale [1955] and Dantzig [1955] began to study optimization problems under uncertainty, followed by Charnes and Cooper [1959] and others, see also Wets [1974, 1975]. Moreover, it was not until the 1980s when Stochastic Optimization was broadly applied in real-world applications.

Uncertainty is the key ingredient in many decision problems. There are several ways in which uncertainty can be formalized and different approaches to optimization under uncertainty have been developed over the past thirty years. The field of SO appears as a response to the need of uncertainty to be incorporated in mathematical optimization models. Basically, it deals with situations in which some parameters are random variables. It allows the management, partially at least, of the risk inherent to the random variables of the problem mainly in a time horizon environment.

New problem formulations appear almost every year and this variety is one of the strengths of the field. Very frequently, mainly in problems with a given time horizon to exploit, some coefficients in the objective function, the *right hand side* (*rhs*) vector and the constraint matrix are not known with certainty when the decisions have to be made, but some information is available. This circumstance allows to use Stochastic Integer Optimization (SIO) to solve multistage mixed 0-1 problems under uncertainty. The problem is formulated by the so-named Deterministic Equivalent Model (DEM), a term coined by Wets [1974], that was first solved by using Benders Decomposition (BD), see Benders [1962]; van Slike and Wets [1969]; Birge and Louveaux [2011]; Laporte and Louveaux [2002], among others. A generalization of BD is given in Carøe and Tind [1998] to deal with two-stage stochastic programs having 0-1 mixed-integer recourse variables and either pure continuous or pure first-stage 0-1 variables. A decomposition algorithm based on a branch-and-cut approach to solve two-stage stochastic programs having first-stage pure 0-1 variables and 0-1 mixed-integer recourse variables is proposed in Sen and Sherali [2006], where a modified BD method is developed. Branch-and-bound algorithms for problem solving having mixed-integer variables in both stages are presented in Carøe and Schultz [1999]; Hemmecke and Schultz [2001].

A general algorithm to solve two-stage stochastic mixed 0-1 problems is presented in Escudero *et al.* [2007, 2009b, 2010a]. In the general formulation of a multistage stochastic integer optimization problem, decisions in each stage have to be made stage-wise. At each stage, there are variables which correspond to decisions that have to be made without anticipation of some of future problem data, i.e., they take the same value under each scenario in a given group (then, the so-named *non-anticipativity constraints* (NAC) must be satisfied, stated in Wets [1975] and restated in Rockafellar and Wets [1991], see also Birge and Louveaux [2011]; Alonso-Ayuso *et al.* [2009]; Pflug and Pichler [2014], among many others). Some approaches have also been introduced to address multistage problems with 0-1 and continuous variables anywhere in the model, and where uncertainty appears only in the objective function coefficients and the *rhs*, see Alonso-Ayuso *et al.* [2003a, 2005].

Moreover, there are few attempts to solve up to optimality large-scale general multistage stochastic mixed 0-1 models, where both types of variables appear at any stage of the time horizon, and where uncertainty can appear anywhere in the problem, i.e., objective function, constraints and *rhs* coefficients at any stage. This type of problems is the most frequent one, but its complexity is the reason for there not being many attempts at developing solving algorithms. For this purpose, a decomposition-based multistage stochastic mixed 0-1 methodology so-named Branch-and-Fix Coordination (for short, BFC), has been introduced in a series of works in Escudero [2009]; Escudero *et al.* [2009a, 2010b, 2012a,b].

Good results have been obtained using the serial version of the Stochastic Dynamic Programming (SDP) based matheuristic introduced in Cristobal *et al.* [2009]; Escudero *et al.* [2013b]. However, the need to analyse the solution provided for a given problem under different alternatives on decisional parameters requires faster schemes than the serial one.

Multistage Stochastic Optimization problems in general require an intensive computing force. Parallel Computing offers an alternative to solve very large scale problems. It is the subject of this work. Actually, the process of solving complex computational problems needs hardware platforms with PC capabilities enabled.

Over the last two decades, papers have appeared in the open literature on Stochastic Optimization that take advantage of Parallel Computing for two-stage and multistage stochastic continuous and mixed 0-1 optimization, see Ruszczyinski [1993]; Birge *et al.* [1996]; Beraldi *et al.* [2000]; Fragniere *et al.* [2000]; Linderoth *et al.* [2006]; Lucka *et al.* [2008] and Al-Khamis and M'Hallah [2011], among others. Some works propose new decomposition methods that are suitable for parallelization, see Mulvey and Ruszczynski [1995]; Vladimirou [1998]; Blomval and Lindberg [2002]; Blomval [2003]. Others revisit classical ones by considering Benders Decomposition for two-stage environment, see Nielsen and Zenios [1997], and the nested approach for the multistage one, see Birge *et al.* [1996], among others. See an algorithmic review, classification and comparison in Vladimirou [1998] . See also several applications in Birge [1997], particular applications in planning under uncertainty in the seminal work Dantzig and Glynn [1990], hydroelectric generation unit commitment in Escudero *et al.* [1999] and finance in Gondzio and Kouwenberg [2001]; Hong *et al.* [2010]; among others. Recently, Aldasoro *et al.* [2013a] and Pagès-Bernaus *et al.* [2014] present parallel computing versions of the BFC algorithm.

In Dias *et al.* [2013], several parallelization strategies, the so-named static and dynamic task scheduling, have are adopted to speed up the stochastic dynamic programming solution to solve a large multiperiod hydrothermal planning problem under uncertainty on a long-time horizon. See Li *et al.* [2014]; Zhang *et al.* [2013] for parallel deterministic dynamic programming applied to reservoir operation optimization. A generic parallel approach is introduced in Stivala *et al.* [2010] considering dynamic programs for problems such as knapsack and shortest paths.

1.7 Multistage stochastic mixed 0-1 optimization

A general way to model a *mathematical programming* problem is as follows, see Kall and Wallace [1994]:

$$\begin{cases} min & f(x) \\ s.t. & g_i(x) \leq 0, \qquad i = 1, \ldots, m, \\ & x \in X \subseteq \mathbb{R}^n, \end{cases} \tag{1.1}$$

and the corresponding *stochastic program* notation is:

$$\begin{cases} min & f(x, \xi) \\ s.t. & g_i(x, \xi) \leq 0, \quad i = 1, \ldots, m, \\ & x \in X \subseteq \mathbb{R}^n, \end{cases} \tag{1.2}$$

where ξ is a random vector varying over a set $\Xi \subseteq \mathbb{R}^k$. More precisely, we assume throughout that a family $\mathcal{F}$ of events, i.e. subsets of Ξ, and the probability distribution P on $\mathcal{F}$ are given. Hence for every subset $A \subseteq \Xi$ that is an event, i.e. $A \in \mathcal{F}$, the probability $P(A)$ is known. Furthermore, we assume that the functions $g_i(x, \cdot) \ : \ \Xi \to \mathbb{R} \ \forall x, i$ are random variables themselves, and that the probability distribution P is independent of x. We will use a scenario analysis approach to model uncertainty. Discrete random variables, ξ, will be considered with a finite number of values, ξ^ω, $\omega \in \Omega$. To simplify notation, we will replace ξ^ω with ω and Ξ with Ω.

Definition 1.1. *A* **stage** *of a given time horizon is a set of consecutive periods where the realization of the uncertain parameters takes place.*

Definition 1.2. *A* **scenario** *is one realization of the uncertain parameters along the stages of the given time horizon.*

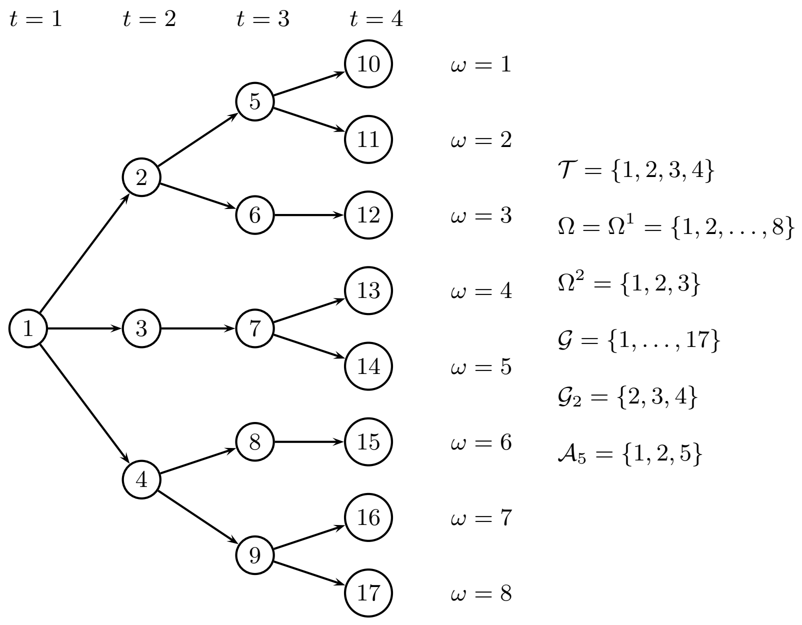

Figure 1.2: An example of scenario tree

Definition 1.3. *A* **scenario group** *for a given stage is the set of scenarios with the same realization of the uncertain parameters up to the stage.*

The notation related to the scenario tree to be used through this work, illustrated in Figure 1.2, is as follows:

$\mathcal{T}$, set of stages along the time horizon, such that $\mathrm{T} = |\mathcal{T}|$ is the last stage.

$\mathcal{T}^t$, set of ancestor stages to stage t (including itself) whose variables have nonzero elements in the constraints of stage $t \in \mathcal{T}$

Ω, set of scenarios, where $\omega \in \Omega$ represents a specific scenario.

$\mathcal{G}$, set of scenario groups, so that there is a tree whose set of nodes is given by $\mathcal{G}$.

$\mathcal{G}_t$, set of scenario groups in stage t, for $t \in \mathcal{T}$ ($\mathcal{G}_t \subseteq \mathcal{G}$).

Ω^g, set of scenarios in scenario group g, for $g \in \mathcal{G}$ ($\Omega^g \subseteq \Omega$).

$\mathcal{A}_g$, set consisting of scenario group g and its ancestors, for $g \in \mathcal{G}$. Note: $\mathcal{A}_g$ is only included by node g for $g \in \mathcal{G}_1$.

w^ω, likelihood that the modeler associates with scenario ω, $P(\xi = \xi^\omega) = w^\omega$, such that $\sum_{\omega \in \Omega} w^\omega = 1$.

w_g, weight factor representing the likelihood that is associated with scenario group g, for $g \in \mathcal{G}$. Note: $w_g = \sum_{\omega \in \Omega^g} w^\omega$

Let x_t and y_t denote the $nx(t)$- and $ny(t)$-vectors of the 0-1 and continuous variables, respectively, in the following deterministic model,

$$\begin{aligned} &\min \sum_{t \in \mathcal{T}} (a_t x_t + b_t y_t) \\ &\text{s.t.} \sum_{t' \in \mathcal{T}^t} (A^t_{t'} x_{t'} + B^t_{t'} y_{t'}) = h_t \qquad \forall t \in \mathcal{T} \\ &\qquad x_t \in \{0,1\}^{nx(t)},\ 0 \le y_t \le \hat{y}_t \quad \forall t \in \mathcal{T}, \end{aligned} \tag{1.3}$$

where a_t and b_t are the row vectors of the objective function coefficients, respectively, h_t is the m-column rhs, $A^t_{t'}$ and $B^t_{t'}$ are the $m \times nx(t)$ and $m \times ny(t)$ constraint matrices, respectively, for stages in $\mathcal{T}^t$, and $\hat{y}_t$ is the upper bound vector of variables in vector y_t, for $t \in \mathcal{T}$. Note: $\mathcal{T}^1$ is only included by stage $t = 1$. The model must be extended in order to deal properly with the uncertainty in the values of some parameters. Thus, an approach to model the uncertainty in the problem data is needed.

Model (1.3) can be extended to consider uncertainty in some of the main parameters. Many of today's approaches to stochastic optimization are scenario-based approaches to deal with the uncertainty. The information structure is visualized as a tree, where each root-to-leaf way represents one specific scenario and corresponds to one realization of the whole set of the uncertain parameters, see Figure 1.2. Each node in the tree can be associated with a scenario group, such that two scenarios belong to the same group in a given period, provided that they have the same realization of the uncertain parameters up to the period. Given the *non-anticipativity* principle, both scenarios should have the same value for the related variables with the time index up to the given period. This research does not distinguish between a scenario group and the corresponding node in the tree (with the same number). For basic concepts and detailed introduction to stochastic optimization, see the books by Birge and Louveaux [2011]; Alonso-Ayuso *et al.* [2009]; Pflug and Pichler [2014], among others.

Different types of models can be presented depending upon the type of recourse to consider, namely, *simple*, *partial* and *full recourse*. Let us consider the minimization of the objective function expected value with multiperiod full recourse, i,e., the risk neutral strategy. In this case, the *compact representation* of the stochastic version of model (1.3) has the following Deterministic Equivalent Model (DEM) by groups,

$$\begin{aligned} z^{DEM} &= \min \sum_{g\in\mathcal{G}} w_g(a_g x_g + b_g y_g) \\ \text{s.t. } &\sum_{q\in\mathcal{A}_g} (A_q^g x_q + B_q^g y_q) = h_g \qquad \forall g\in\mathcal{G} \\ &x_g \in \{0,1\}^{nx(g)},\, 0 \le y_g \ \le \hat{y}_g \qquad \forall g\in\mathcal{G}, \end{aligned} \tag{1.4}$$

where a_g and c_g are the row vectors of the objective function coefficients, A_q^g and B_q^g are the constrains matrices for vectors x_q and y_q in the system related to scenario group g, respectively, b_g is the *rhs*, x_g and y_g are the vectors of the variables of scenario group g, and $\hat{y}_g$ is the upper bound vector of variables in vector y_g, for $g \in \mathcal{G}$. All vectors and matrices have the appropriate dimensions. Hereafter, the components of x_q (whose stage is $t(q)$) will be referred as *linking variables*, since they will directly affect the decision in stage $t(g)$.

On the other hand, *splitting variable representation* by stages can be expressed as follows, see Escudero *et al.* [2010b, 2012a],

$$\begin{aligned} z^{DEM} &= \min \sum_{\omega\in\Omega}\sum_{t\in\mathcal{T}} w^\omega \left(a_t^\omega x_t^\omega + b_t^\omega y_t^\omega\right) \\ \text{s.t. } &\sum_{t'\in\mathcal{T}^t} \left(A_{t'}^{t,\omega} x_{t'}^\omega + B_{t'}^{t,\omega} y_{t'}^\omega\right) = h_t^\omega \quad \forall\omega\in\Omega, t\in\mathcal{T} \\ &x_t^\omega - x_t^{\omega'} = 0 \quad \forall \omega,\omega'\in\Omega^g : \omega\neq\omega',\ g\in\mathcal{G}_t, t\in\mathcal{T} \\ &y_t^\omega - y_t^{\omega'} = 0 \quad \forall \omega,\omega'\in\Omega^g : \omega\neq\omega',\ g\in\mathcal{G}_t, t\in\mathcal{T} \\ &x_t^\omega \in \{0,1\}^{nx_t^\omega} \quad y_t^\omega \in \mathbb{R}^{+ny_t^\omega},\ \ \forall\omega\in\Omega,\ t\in\mathcal{T}, \end{aligned} \tag{1.5}$$

where a_t^ω and b_t^ω are the objective function coefficients of the variables x_t^ω and y_t^ω for stage $t \in \mathcal{T}$ under scenario $\omega \in \Omega$, respectively, and $A_{t'}^{t;\omega}$ and $B_{t'}^{t,\omega}$ $\forall t' \in \mathcal{T}^t$ are the constraint matrices, for the $x_{t'}^\omega$ and $y_{t'}^\omega$ variables in the constraints related to stage $t \in \mathcal{T}$, respectively. Note that $x_t^\omega - x_t^{\omega'} = 0$ and $y_t^\omega - y_t^{\omega'} = 0$ are the NAC. Finally, nx_t^ω and ny_t^ω denote the dimensions of the vectors of the x and y variables, respectively, related to stage t under scenario ω.

Following the nonanticipativity principle, the corresponding equalities must be satisfied for stage t,

$$a_t^\omega = a^g, b_t^\omega = b^g, h_t^\omega = h^g,\ \ \forall\omega\in\Omega^g,\ g\in\mathcal{G}_t,\ t\in\mathcal{T} \tag{1.6}$$

$$A_{t'}^{t,\omega} = A_q^g, B_{t'}^{t,\omega} = B_q^g,\ \ \forall q\in\mathcal{G}_{t'},\ t'\in\mathcal{T}^t,\ \omega\in\Omega^g,\ g\in\mathcal{G}_t \tag{1.7}$$

Chapter **2**

Parallel computing in optimization

Diviser chacune des difficultés en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre.

Divide each difficulty into as many parts as is feasible and necessary to resolve it.

René Descartes, *Le Discours de la Méthode*

2.1 Introduction

This chapter presents the main ideas of Parallel Computing applied to Stochastic Optimization in two directions, namely, parallel solving of multiple problems and a parallel branching solving process of a single problem.

First, a general environment is presented for parallel computing when solving several optimization problems by a message-passing paradigm code and its extension to a combined distributed memory/shared memory parallelization. We present the corresponding software/hardware environment, the scheme of the parallelization strategy and the basics of the programming syntax used in its implementation.

We show the variation of computing time depending on the number of threads used and the selected solver in the computational experience we are reporting. This will lead to a comparison between the serial version and the different parallel strategies to solve a set of mixed integer optimization problems and, ultimately, to conclude that the parallel paradigm shows a significant improvement.

Second, the Parallel Computing based approach for improving the process to solve a single mixed 0-1 stochastic problem is analysed. We present an extension of the Branch & Bound (B&B) algorithm, the so-called Multipath Branch & Bound scheme in which the original binary variable tree is split among threads, i.e. parallel node branching, and iterative communication, allowing an earlier branch pruning by incumbent gathering. An illustrative example shows the reduction of branching nodes for each thread, due to the parallel features. In addition to the simultaneous branching, a stage perspective approach results in shorter computing time when applying the Multipath Branch & Bound scheme to a smaller scale problem; these features will be deeply exploited by the BFC algorithm proposed in Chapter 3 and Chapter 5. The conclusions of this chapter lead to define the inner and outer parallelization paradigms to be applied to decomposition algorithms presented in Chapter 3 and Chapter 4.

The rest of the chapter is organized as follows: Section 2.2 presents the main concepts of a message-passing parallelization environment. Section 2.3 describes the parallel solving of multiple optimization and reports a computational experience. Section 2.4 defines the inner parallelization paradigm based on the results of the previous section. Section 2.5 introduces the Multipath Branch & Bound algorithm intended to solve optimization problems by parallel branching. Section 2.6 defines the outer parallelization paradigm based on the Multipath Branch & Bound performance conclusions. Section 2.7 summarizes the main conclusions.

2.2 Message-passing parallelization

The main idea behind message-passing parallelization is that every task thread receives an exact copy of the executable and, as there are not shared variables, the information exchange is performed by a message-passing environment. The task distribution is achieved by a *thread rank* (thread identification number) based branching, in other words, defining the rank of the thread that will perform a specific statement or by branching data vectors by rank. This allows it to work on a *Single Program Multiple Data* (SPMD) paradigm.

Let us start the description of the MPI environment by underlining the three main phases that appear at machine level on a message-passing paradigm, see Pacheco [1996]. These phases are not directly executed by the user and they are automatically performed by the machine.

Phase a: The user issues a directive to the operating system which has the effect of placing a copy of the same executable program on each task thread.

Phase b: Each task thread begins executing its copy of the executable.

Phase c: Different task threads can execute different statements by branching within the program. Typically branching will be based on thread ranks.

The implementation of the corresponding executable supports different experimental codes. Figure 2.1 presents the general structure of an MPI implementation in C++, see Pacheco [1996]. This layout can be divided into five groups by considering the nature of the functions they use. Appendix A describes the layout groups of the general MPI program.

```
...
#include "mpi.h"
...
main(int argc, char **argv) {
...
  // Group 1: Declaring MPI variables.
  ...
  // Group 2: Beginning of the MPI environment
  (No MPI functions called before this).
  ...
  // Group 3: Functions controlling the number of processes.
  ...
  // Group 4: Communication functions.
  ...
  // Group 5: Finishing the MPI environment
   (No MPI functions called after this).
  ...
...
}
...
```

Figure 2.1: General structure of a MPI implementation

The full C++ code that we have developed for the parallel solving of several mixed integer optimization problems is described in Aldasoro *et al.* [2012]. In addition, a summarized and schematic version of the same code has been attached as Appendix A to this thesis. Note that the short version of the code is not an executable file but a useful tool for presenting its general structure.

2.3 Parallel computing to solve multiple optimization problems

The serial solving of multiple optimization problems consists of reading the data files, creating the optimization models, solving them and reporting the corresponding results, see Figure 2.2 for solving 44 problems.

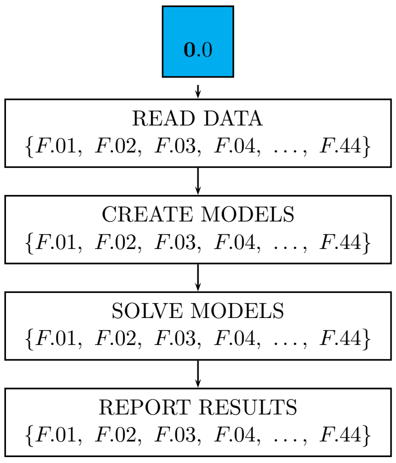
0.0
READ DATA
{F.01, F.02, F.03, F.04, ..., F.44}
CREATE MODELS
{F.01, F.02, F.03, F.04, ..., F.44}
SOLVE MODELS
{F.01, F.02, F.03, F.04, ..., F.44}
REPORT RESULTS
{F.01, F.02, F.03, F.04, ..., F.44}

Figure 2.2: Serial execution. Basic steps.

We can extend this descriptive diagram to a two thread parallel version as shown in Figure 2.3 for solving 44 problems. In this case, the first three tasks are performed by both threads sharing the total amount of problems; it significantly reduces the total amount of time. Once the solving process is finished in both threads, the main thread gathers the numerical results by message-passing functions and reports them.

The modeler can define the branching cited on the third block according to the desired parallel programming strategy. The following procedure illustrates the parallel computing key idea behind the example code given in Appendix A, see a parallel diagram illustrated in Figure 2.4. Observe that there is only a single main thread in this section (remember the basic concepts of Parallel Computing in Section 1.3).

Step 0: Declaring optimization and MPI variables. [All task threads]

Step 1: Definition of the global environment. [All task threads]

- Beginning the MPI environment using functions presented in Section A.2

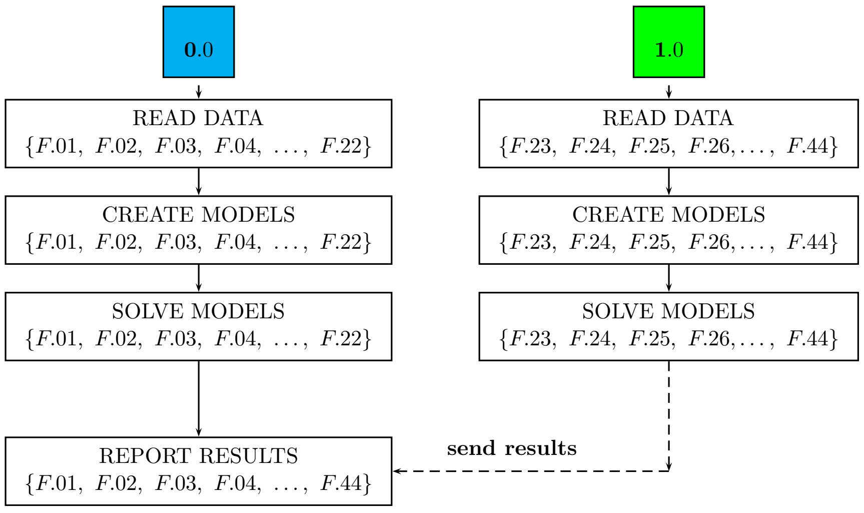
0.0
1.0
READ DATA {F.01, F.02, F.03, F.04, ..., F.22}
READ DATA {F.23, F.24, F.25, F.26,..., F.44}
CREATE MODELS {F.01, F.02, F.03, F.04, ..., F.22}
CREATE MODELS {F.23, F.24, F.25, F.26,..., F.44}
SOLVE MODELS {F.01, F.02, F.03, F.04, ..., F.22}
SOLVE MODELS {F.23, F.24, F.25, F.26,..., F.44}
REPORT RESULTS {F.01, F.02, F.03, F.04, ..., F.44}
send results

Figure 2.3: MPI based parallel execution. Basic steps.

Defining the total number of task threads, the thread rank of each of them and creation of a new communicator by using functions presented in Section A.3. By default, thread ranked 0 is defined as main thread.

Step 2: Presolve assigned models. [All task and auxiliary threads]

Every task thread reads the corresponding information and creates the associated optimization models.

- If selected optimizer allows multiple auxiliary threads: Each task thread starts a shared memory parallel computing environment with a chosen number of auxiliary threads. An illustrative diagram is depicted in Figure 2.4 that corresponds to using 2 task threads and 4 auxiliary threads per task thread. Assigned models are presolved.
- If selected optimizer only allows a single auxiliary thread: Each task thread presolves assigned models.

Step 3: Global MPI communication. [All task threads]

The main thread gathers the presolve information of all models and all task threads gather presolve status by using functions presented in Section A.4

Step 4: Check the presolve status [Main thread]

- If all models are feasible and bounded: Go to Step 5.

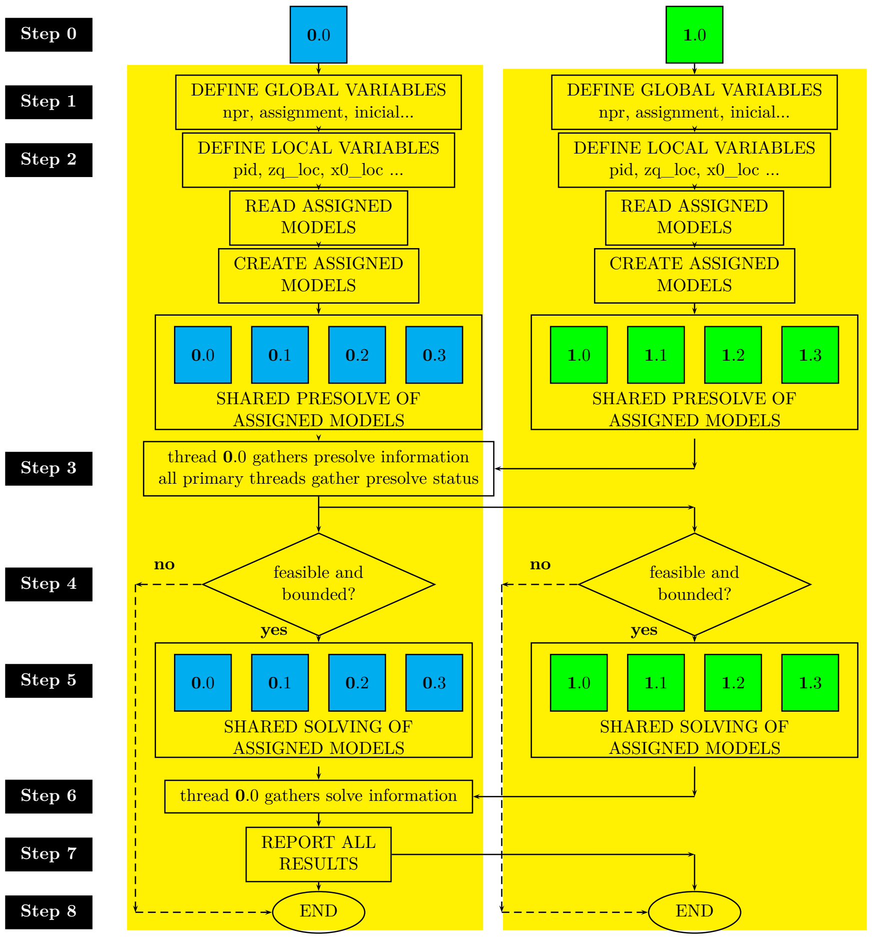

Figure 2.4: Parallelization diagram (1 main × 2 task × 4 auxiliary) threads

- Else: Go to Step 8.

Step 5: Solve assigned models. [All task and auxiliary threads]

- If selected optimizer allows multiple auxiliary threads: Each task thread starts a shared memory parallel programming environment with a chosen number of auxiliary threads (see Figure 2.4). Assigned models are solved.
- If selected optimizer only allows a single auxiliary thread: Each task thread solves assigned models.

Step 6: Global MPI communication. [All task threads]

The main thread gathers the presolve information of all models by using functions presented in Section A.4

Step 7: Report results [Main thread]

The main thread reports all results.

- Finishing the MPI environment using functions presented in Section A.5

Step 8: Execution ends in all task threads. [All task threads]

2.3.1 Preliminary computational testing

Before performing the experiments we conducted some preliminary testing, in order to have an initial characterization of the parallelization environment. To do so, we have implemented the simplest possible parallel strategy, i.e., the strategy with the steps described in Section 2.3 without the final gather of information. In this context threads work in a fully independent way, and are only connected at the very beginning to distribute the models to solve. As a consequence, the parallel executions will not necessarily end simultaneously.

The chosen solver is COIN-OR V1.3.1, which does not support multiple auxiliary threads. The work load consists of three different mixed 0-1 optimization problems repeated 3 times, i.e., a total amount of 9 instances to be solved by an increasing amount of threads in order to check the effect of the unbalanced work load, that is, the bottle-neck effect.

Another aspect to analyse is the behaviour of the computational cluster regarding the communication time. It should be reduced as far as possible since it impacts the efficiency of the execution. As preliminary testing we will allow the ARINA computational cluster to select the threads to be used, see Section 1.5. The results are summarized in Table 2.1.

Table 2.1: Preliminary test. COIN-OR. 9 instances.

	Available	**Maximum**	**execution**					**Elapsed time**	
	threads	**# prob/thread**	**1**	**2**	**3**	**4**	**5**	**Average**	**St. dev.**
Serial	**1**	9	206	159	205	214	238	204.4	28.7
MPI	**1**	9	213	212	254	222	265	233.2	24.6
	2	5	280	273	237	270	386	289.2	56.6
	3	3	118	153	229	225	135	172	51.7
	4	3	110	190	195	170	194	171.8	36
	5	2	81	104	120	102	184	118.2	39.3
	6	2	127	229	239	134	100	165.8	63.6
	7	2	112	118	162	174	287	170.6	70.4
	8	2	199	183	226	226	221	211	19.2
	9	1	97	105	72	86	60	84	18.3

To conclude, we can make several observations. First, the variability of the elapsed time is highly significant, in other words, the necessary time to create the MPI environment depends on the chosen threads and the machine work load situation. Consequently from now on in this thesis the user will specifically select the threads to be used, ideally being part of the same computing node in order to reduce communication time.

Second, the absence of further communication among threads once the solving process has started produces a chaotic behaviour with regard to the elapsed time. Therefore, the other implementations reported in this work consider intermediate communication among threads in order to guarantee the most simultaneous execution as possible. Additionally, following the criterion that is very frequently found in the general literature, the number of threads to be used will be a power of 2, since it is fast and convenient to implement in a computer.

Taking the above conclusions into account, the next subsection presents a broad computational experience on comparing parallel and serial versions of decomposition algorithms solve a set by instances of a medium-scale optimization problem.

2.3.2 Computational experience

The computational experiments were conducted in the ARINA computational cluster at SGI/IZO-SGIker, Universidad del País Vasco, UPV/EHU, see Section 1.5. For the reported experiments, Intel Xeon type computing nodes were used, consisting of 8 cores with 48Gb of RAM, see Table 1.1. The optimization problems were solved using COIN-OR V1.3.1 and CPLEX v12.2 from COIN-OR V1.3.1 environment.

The set of instances whose computational experience is analysed in this subsection is based on the scenario-cluster submodels obtained from the multistage stochastic mixed 0-1 problem P4 taken from the computational experience reported in Escudero *et al.* [2012a] and countered as one of the problems included in Testbed 2 whose results are reported in Table 3.2. The full model is a medium-scale problem with $|\Omega| = 217$ scenarios (structured in nonsymmetric scenario tree) and $|\mathcal{T}| = 4$ stages, where the nonanticipativity constraints are relaxed for the first and second stage, in order to generate the 44 instances considered in this subsection.

The dimensions of the original medium-scale optimization problem and the related average and standard deviation of the dimensions of the subproblems are shown in Table 2.2 and Table 2.3, respectively. The headings of the table are as follows: m, number of constraints; nx, number of 0-1 variables; ny, number of continuous variables; nel, number of nonzero coefficients in the constraint matrix and $dens$, constraint matrix density (in %). Table 2.4 shows the value of the optimal objective function for the 44 mixed integer problems.

Table 2.2: Medium-scale problem dimensions

m	nx	ny	nel	$dens$
9248	2176	4792	515768	0.64

Table 2.3: Subproblem average (standard deviation) dimensions.

	m	nx	ny	nel	$dens$
average	270	63	151	14263	7.93
(st. deviation)	(53.46)	(12.58)	(251.16)	(2981.38)	(1.57)

Table 2.4: Solution values for the 44 subproblems. Testbed 1.

F.1	-2953.46	F.12	-3215.94	F.23	-8945.76	F.34	-2915.80
F.2	-5067.97	F.13	-6662.71	F.24	-7271.52	F.35	-4291.44
F.3	-3957.70	F.14	-4078.15	F.25	-7136.25	F.36	-9412.94
F.4	-5310.86	F.15	-5315.24	F.26	-7046.85	F.37	-7899.80
F.5	-9219.36	F.16	-8504.21	F.27	-5561.61	F.38	-4851.96
F.6	-6539.62	F.17	-6592.45	F.28	-5327.66	F.39	-7735.64
F.7	-8196.80	F.18	-5186.78	F.29	-3991.81	F.40	-7764.66
F.8	-6582.53	F.19	-6291.11	F.30	-8051.70	F.41	-9828.10
F.9	-7811.68	F.20	-7378.17	F.31	-4887.34	F.42	-2786.57
F.10	-8126.43	F.21	-8450.21	F.32	-5434.41	F.43	-4311.35
F.11	-8420.17	F.22	-8558.38	F.33	-9432.48	F.44	-8135.21

Table 2.5 and Table 2.6 show the main execution times for COIN-OR optimizer and for CPLEX solver within COIN-OR (allowing one single auxiliary thread for CPLEX), respectively. The headings are as follows: *Available threads (th)*, the number of threads for the serial and parallel programming; *Maximum # prob/thread*, the bottleneck or maximum number of problems to be solved by thread, that is, $\left\lceil \frac{C}{th} \right\rceil$; *Min, Average, Max*, the minimum, average and maximum CPU time by thread (in seconds), respectively; *Average, St. dev.*, the average and standard deviation (in seconds) for the elapsed time after 15 executions of the same code. CPLEX being called from COIN-OR is between 3 and 5 five times faster than own COIN-OR optimizer (without any presolving or cut generation). Note that the *CPU time* measures the instruction processing time of the computer program whereas the *elapsed time* (also known as real time or wall-clock time) includes the CPU time, the input/ouput time and the communication delay. We can conclude that the CPU time and the elapsed time are not significantly different, therefore, throughout this memoir *elapsed time* will be the analysed magnitude.

Table 2.5: Computing times under COIN-OR. Testbed 1.

	Available threads	Maximum # prob/thread	CPU time Min.	CPU time Average	CPU time Max.	Elapsed time Average	Elapsed time St. dev.
Serial	**1**	44	17.651	17.651	17.651	17.780	0.013
MPI	**1**	44	18.013	18.013	18.013	18.239	0.093
	2	22	8.823	8.866	8.779	8.972	0.010
	4	11	5.018	5.165	5.271	5.373	0.029
	8	6	3.118	3.647	3.960	4.508	0.253
	16	3	1.735	2.077	2.272	2.632	0.270
	32	2	1.239	1.857	1.996	2.054	0.005
	64	1	0.839	1.403	1.517	1.590	0.260

Table 2.6: Computing times under CPLEX (single auxiliary thread). Testbed 1.

	Available threads	Maximum # prob/thread	CPU time Min.	CPU time Average	CPU time Max.	Elapsed time Average	Elapsed time St. dev.
Serial	**1**	44	6.107	6.107	6.107	6.163	0.019
MPI	**1**	44	6.344	6.344	6.344	6.401	0.005
	2	22	3.161	3.165	3.168	3.201	0.002
	4	11	1.647	1.667	1.685	1.706	0.005
	8	6	0.992	1.025	1.074	1.120	0.030
	16	3	0.521	0.546	0.564	0.604	0.058
	32	2	0.349	0.390	0.405	0.426	0.0175
	64	1	0.227	0.259	0.276	0.297	0.0857

Table 2.7 shows the execution times for CPLEX solver from COIN-OR considering a number of available threads and several combinations between the number of threads for distributed memory (task threads) and shared memory (auxiliary threads). The other headings are as denoted in the previous tables. For 8 available threads the average elapsed time for computing the serial execution is 7.699 seconds, which corresponds to the worst strategy. Figure 2.5 shows the elapsed time for COIN-OR in horizontal read line and the elapsed time for CPLEX in vertical blue bars, it is classified according to the available number of threads, as detailed in Tables 2.5 and 2.7. We can observe that for a fixed number of available threads the elapsed time to solve the 44 problems (a large number of them with small dimensions) decreases when the number of MPI threads increases. So, the fastest case corresponds to the highest number of task threads and a single auxiliary thread for CPLEX, which highlights the interest of MPI parallel computing.

Finally, Table 2.8 summarizes the best elapsed times for COIN-OR and CPLEX, with all the available threads considered. The headings are as follows: *Elapsed Time*, the wall clock time in seconds (average time over 15 realizations); S_{th}, *speedup*, serial elapsed time over parallel elapsed time; and $E_{th}\%$, *efficiency*, speedup over number of

Table 2.7: Computing times under CPLEX (multiple auxiliary threads). Testbed 1.

Available threads	**task × auxiliary threads**	**CPU time Min.**	**CPU time Average**	**CPU time Max.**	**Elapsed time Average**	**Elapsed time St. dev.**
1	1 × 1	6.107	6.107	6.107	6.163	0.019
2	1 × 2	4.328	4.328	4.328	5.683	0.046
2	2 × 1	3.161	3.165	3.168	3.201	0.002
4	1 × 4	2.382	2.382	2.382	6.076	0.055
4	2 × 2	2.141	2.184	2.227	2.986	0.022
4	4 × 1	1.647	1.667	1.685	1.706	0.005
8	1 × 8	2.590	2.590	2.590	7.699	0.222
8	2 × 4	1.184	1.229	1.258	3.229	0.245
8	4 × 2	1.095	1.124	1.176	1.632	0.032
8	8 × 1	0.992	1.025	1.074	1.120	0.030

MPI= 1 corresponds to serial execution

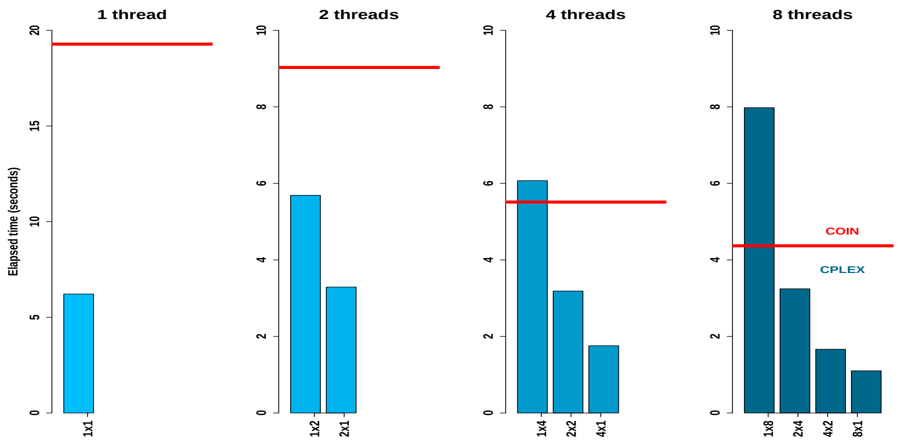

Figure 2.5: Classification for (task × auxiliary) threads execution time

threads, in percentage. The speedup and efficiency is high and similar in both solvers for *th*= 2 and *th* = 4 available threads, but it is better for CPLEX optimization solver for 8 or more threads. These results are illustrated in Figure 2.6.

Table 2.8: Efficiency of COIN-OR and CPLEX. Testbed 1.

	Available	**COIN-OR**			**CPLEX**		
	threads	**Elapsed time**	$\mathbf{S_{th}}$	$\mathbf{E_{th}}\%$	**Elapsed time**	$\mathbf{S_{th}}$	$\mathbf{E_{th}}\%$
Serial	**1**	17.780	1.000	100.0	6.163	1.000	100.0
MPI	**2**	8.972	1.982	99.1	3.201	1.925	96.3
	4	5.373	3.309	82.7	1.706	3.613	90.3
	8	4.508	3.944	49.3	1.120	5.503	68.8
	16	2.632	6.755	42.2	0.604	10.204	63.8
	32	2.054	8.656	27.1	0.426	14.670	45.8
	64	1.590	11.182	17.5	0.297	20.751	32.4

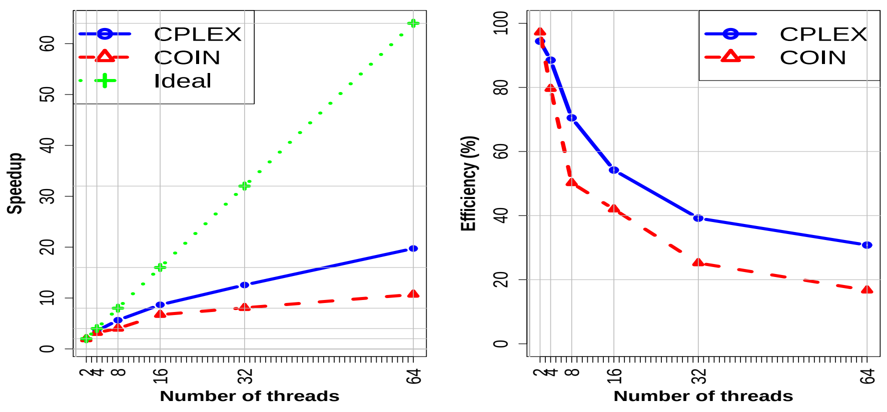

Figure 2.6: Speedup and efficiency vs number of threads

2.4 Inner parallelization paradigm

The truly potential of the strategy presented in Section 2.3 to solve multiple optimization problems in parallel consists of its insertion into an iterative algorithm. In other words, the parallelization environment can work as an iterative step of a decomposition algorithm to solve large-scale problems, parallelizing the resolution of the corresponding subproblems. We could define this new approach as *inner parallelization* since it works as an internal parallelization of a serial algorithm. The inner parallelization paradigm is applied to an exact decomposition algorithm in Chapter 3 as well as to a matheuristic decomposition algorithm in Chapter 4.

2.5 Multipath Branch & Bound Algorithm

Once the MIP optimization model is split among submodels whose optimization is assigned to the chosen solver executions, let us improve each solving process in large-scale problems. The internal parallel mode of any solver is an efficient black box for the user, however, two weak points can be identified, namely, its parallel resources are limited (up to 8 auxiliary threads are allowed under academic license for the state-of-the-art optimizer CPLEX) and what is more difficult to overcome in the near and medium future, the current solvers do not profit from the problem structure. Therefore, this section analyses potential improvements in this direction, leading to more complex solving environments.

In this section a parallel computing framework is presented to solve a large-scale MIP problem. Based on the serial Branch & Bound algorithm, it includes two new aspects drawn from weak points in the current solvers: simultaneous branching and stage perspective.

2.5.1 Simultaneous branching

The serial Branch & Bound algorithm is a tool to solve mixed 0-1 problems by obtaining its optimal solution. A linear programming problem is solved in every node of a branching tree where the branching and pruning criteria ensures that the non visited nodes are suboptimal. However, the number of visited nodes and, therefore, the execution time, increases significantly for medium and large-scale problems.

Parallel computing can reduce the execution time of the serial Branch & Bound algorithm, as different approaches can be found throughout the literature.

Note that the paradigm presented in Section 2.4 cannot be directly applied, since it solves several problems in parallel and each node of the Branch & Bound consists of a single problem. Nevertheless, this idea can be adapted to visiting several nodes at the same time, that is, a parallel visiting of Branch & Bound nodes. This simultaneous branching paradigm can be defined in several ways, such as a main thread can lead the tree branching and several task threads can solve the neighbour nodes or, more interestingly, the original problem tree can be split into parts. The second option allows several simultaneous Branch & Bound executions to be worked with, in other words, the original tree is branched by several paths, each managed by a main thread. Thus, the serial version can be considered as a one-path Branch & Bound.

Multipath B&B Algorithm

As previously mentioned, the Multipath Branch & Bound algorithm splits the branching tree among paths. Consequently, each resulting tree will have a smaller size since a modeler-driven set of integer variables from the original problem will be fixed to integer values.

Additionally, the communication capability of MPI can be used for interacting among the paths. The incumbent solution could be updated by gathering the best feasible solution from among all main threads, allowing an earlier pruning of the branches. Moreover, as in general different paths need to visit a different number of nodes they will not usually end simultaneously. Once a main thread ends the algorithm execution, it is considered a *dead path*, but it does not mean that its computational power is not to be used any more, since the *Synchronization Phase* can be started. It consists of splitting an *active path* (a non-dead one) into two paths, with one of them being assigned to the original main thread and the second one to a main thread in dead status. Algorithm 2.1 and Figure 2.7 schematically presents the steps of the algorithm.

This parallelization paradigm assures that the full parallel power is used during the whole execution. It behaves very efficiently for big branching trees for medium-scale or non divisible problems, see below.

A step-by-step numerical example

Considering that a numerical example can make it easier to understand the mechanism of the outer parallelization by multipath branching, let us consider a 0-1 Knapsack optimization problem. This family of problems has the characteristic of being purely binary and having only a single inequality constraint. We consider that it is an appropriate environment for following the steps of the Multipath Branch & Bound algorithm without loose of generality.

Let us consider the following Knapsack problem:

$$\begin{aligned} z = \max \quad & 23x_1 + 19x_2 + 28x_3 + 14x_4 + 44x_5 \\ s.t. \quad & 8x_1 + 7x_2 + 11x_3 + 6x_4 + 19x_5 \leq 25 \\ & x_1, x_2, x_3, x_4, x_5 \in \{0,1\} \end{aligned} \tag{2.1}$$

Table 2.9 shows the branching nodes and their solution for a serial Branch & Bound algorithm. Figure 2.8 depicts the related branching tree. Table 2.10 and Table 2.11 and Figure 2.9 and Figure 2.10 resume the resolution by two-path and four-path, respectively, from the application of the Multipath Branch & Bound algorithm.

Before starting, let us describe the branching criteria, valid for all strategies. The first branching side for any variable non yet 0-1 valued is the 0 value. Additionally, if the resolution of a LP relaxed problem gives as a result 0-1 values for some variables, then the algorithm temporally fixes those variables to the solution values and branch on the next one.

Algorithm 2.1: Multipath Branch & Bound

```
Every main thread executes:

Step 0: (Initializations) Define the initial starting point of each path and
        fix the variables for each of them.

Step 1: (Branch) The branching criterion could be modified by the user,
        such as depth-first, 0-first etc.

Step 2: (Solve relaxed model) The corresponding relaxation of the
        original problem is solved in serial by a unique main thread. The
        value of the objective function is stored in the local $z_{path}$ variable.

Step 3: (Gather $z_{path}$) All paths gather $z_{path}$ values. Note that this step
        works as a synchronization phase since all main threads must
        arrive here in order to execute the communication.
        The values are stored at a solution vector $z_{vect}$.
        If there is not any feasible solution of the original problem in
        $z_{vect}$, then go to Step 1.
        If there is a feasible solution of the original problem, at least,
        in $z_{vect}$, then store the best solution value at the local variable $z^*$.

Step 4: (Update incumbent) Update the incumbent value,
        if minimization problem then $z^{DEM} := min(z^{DEM}, z^*)$.
        if maximization problem then $z^{DEM} := max(z^{DEM}, z^*)$.

Step 5: (Prune) Check if the branch can be locally pruned, (i.e., at the
        current node), or if the branch can be pruned from an upper
        position (this can be done when the incumbent solution value is
        updated with a solution coming from a different main thread).

Step 6: (Gather execution status) All paths gather execution status,
        knowing which paths are dead and which are still active.
        If all paths are dead, then STOP.

Step 7: (Redefine path platform)
        If there are death paths, the same amount of active path are split
        in two active paths and the main threads related to dead paths
        are reassigned to active paths.
```

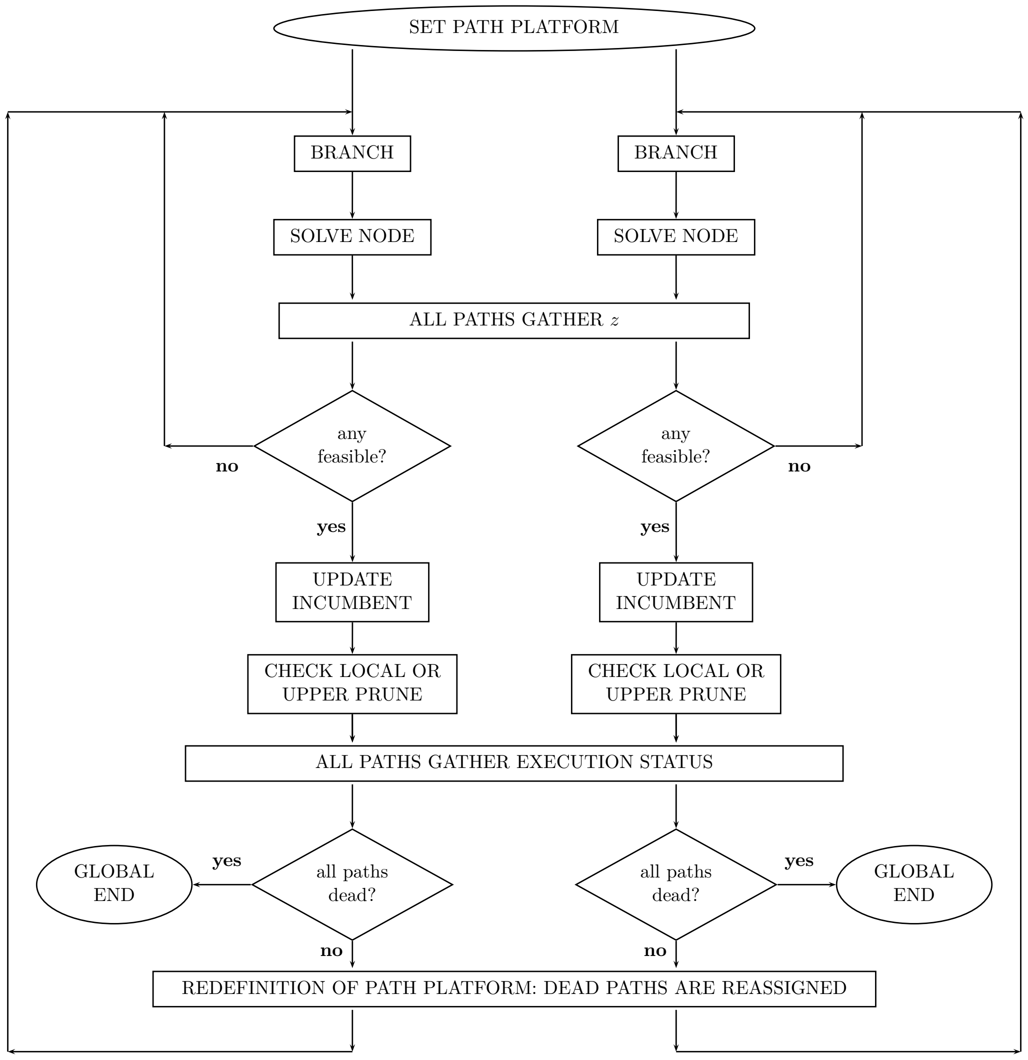

Figure 2.7: Multipath Branch & Bound algorithm. Basic steps

One-path Branch & Bound (serial B&B)

The serial Branch & Bound algorithm, understood as the one-path particular case of the Multipath Branch & Bound algorithm, starts at the root node of the tree. The first LP relaxed problem is solved obtaining a solution value of 67.45 and 0-1 values for x_1 and x_2 (see Table 2.9). Following the branching criteria the algorithm fixes x_1 and x_2 to 1 and branches x_3 to 0. Once the second relaxation is solved we obtain to the third node, since the chosen problem is very small we are already at the last tree level with all variables fixed to a binary value. The corresponding resolution gives an objective value of 56, and moreover, this solution is feasible at the original problem (which is why the node is coloured in green in Figure 2.8) and we can update the incumbent solution from $-\infty$ to 56.

Table 2.9: One-path solving process. Solutions

node	x_1	x_2	x_3	x_4	x_5	z_{path}	z^{DEM}
1^a	1	1	0.91	0	0	67.45	$-\infty$
2^a	1	1	0	1	0.21	65.26	$-\infty$
3^a	1	1	0	1	0	56	56
4^a	1	1	0	1	1	infeasible	56
5^a	1	1	0	0	0.53	65.16	56
6^a	1	1	0	0	0	42	56
7^a	1	1	0	0	1	infeasible	56
8^a	1	1	1	0	0	infeasible	56
9^a	1	0	1	1	0	65	65
10^a	0	1	1	1	0.05	63.32	65

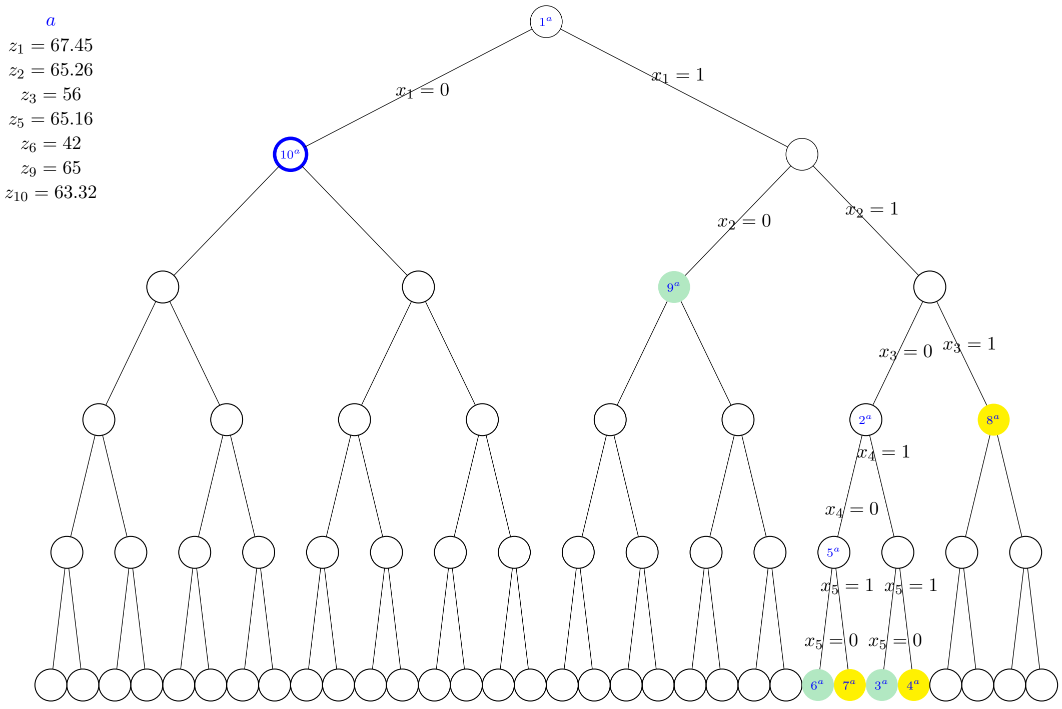

Figure 2.8: Knapsack problem example. One-path B&B

Then we branch variable x_5 to 1, obtaining an infeasible problem (yellow coloured node) so we go up to the first non branched node (in this case node 2) and we keep the same branching criteria. After visiting several nodes we obtain to node 9, having x_1 and x_2 branched to 1 and 0 respectively, the resolution of the relaxation shows a feasible solution of the original problem, updating the incumbent solution to 65. Finally we branch the starting x_1 variable to 0, analysing the second side of the tree. The relaxed solution gives an objective function value of 63.32, being smaller than the incumbent solution we can prune the branch (the blue ring at Figure 2.8 indicates the prune) and the algorithm will stop.

Two-path Branch & Bound

Now let us apply the same criteria to a two-path Branch & Bound, denoted with a and b superscripts respectively, see Table 2.10 and branching tree in Figure 2.9.

Table 2.10: Two-path solving process. Solutions

node	x_1	x_2	x_3	x_4	x_5	z_{path}	z^{DEM}
1^a	0	1	1	1	0.05	63.32	$-\infty$
2^a	0	1	1	1	0	61	61
3^a	0	1	1	1	1	infeasible	61
4^a	0	1	1	0	0.37	63.21	61
5^a	0	1	1	0	0	47	61
6^a	0	1	1	0	1	infeasible	61
7^a	0	1	0	1	0.63	60.79	61
8^a	0	0	1	1	0.42	60.53	61
9^a	1	0	1	1	0	65	65
1^b	1	1	0.91	0	0	67.45	$-\infty$
2^b	1	1	0	1	0.21	65.26	61
3^b	1	1	0	1	0	56	61
4^b	1	1	0	1	1	infeasible	61
5^b	1	1	0	0	0.53	65.16	61
6^b	1	1	0	0	0	42	61
7^b	1	1	0	0	1	infeasible	61
8^b	1	1	1	0	0	infeasible	61
9^b	1	0	0	1	0.58	62.47	65

The first change corresponds to the starting node, which is situated at the second level of the algorithm, saving the branch of the x_1 variable. Both paths obtain an infeasible solution for the original problem (since not all variables have 0-1 values) so they keep branching. At the second node Path a obtains a feasible solution whose value is 61, whereas Path b obtains an infeasible solution of the original problem. Since Path a obtains a feasible solution for the original problem, its solution value 61 is the incumbent for both paths.

Consequently at the z gather phase Path b will the 61 solution, both paths update

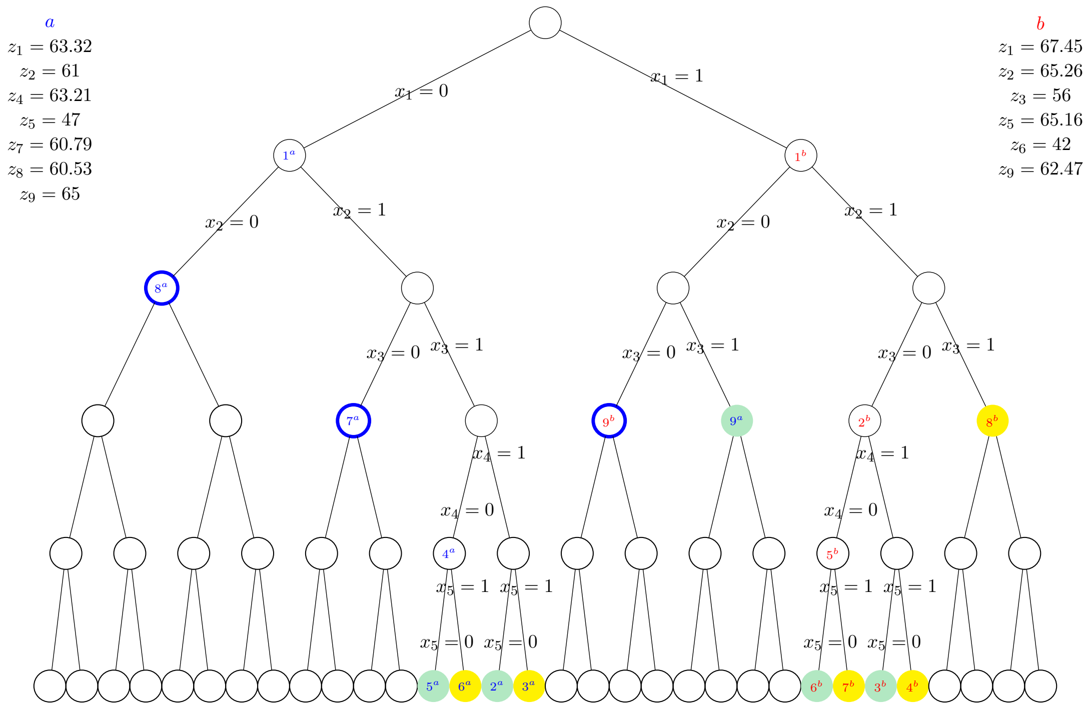

Figure 2.9: Knapsack problem example. Two-path B&B

the incumbent to 61. They both check if it is possible to perform a local o upper prune of the branch, as it is not the case they keep branching. After visiting several nodes, Path a is capable of pruning nodes 7 and 8. At this point Path a is dead and Path b needs to branch x_3 to 0, this is when the *Synchronization Phase* takes place, moving Path a to the Path b side and branching x_3 to 1. Now nodes 9^a and 9^b are solved and the solutions 65 and 62.47 are obtained respectively, gathering those values both branches update incumbent value to 65. Consequently, both paths can locally prune the branch and both paths are dead, so the algorithm stops.

In this example the three advantages of the multipath version of the Branch & Bound algorithm can be observed: The starting node has a lower level than in the serial version, paths exchange solutions allowing a faster update of the incumbent solution and finally the reassignment phase ensures that main threads are working without a break. On the other hand, in this case the number of visited nodes by the longest branch is only slightly smaller, 9 compared to 10. Let us check if the use of more paths improves that behaviour.

Four-path Branch & Bound

Table 2.11: Four-path solving process. Solutions

node	x_1	x_2	x_3	x_4	x_5	z_{path}	z^{DEM}
1^a	0	0	1	1	0.42	60.53	65
2^a	1	1	0	1	0	56	65
3^a	1	1	1	1	0	infeasible	65
1^b	0	1	1	1	0	61	65
2^b	1	1	0	0	1	infeasible	65
3^b	1	1	1	0	1	infeasible	65
1^c	1	0	1	1	0	65	65
2^c	1	1	0	1	1	infeasible	65
3^c	1	1	1	1	1	infeasible	65
1^d	1	1	0.91	0	0	67.45	65
2^d	1	1	0	0	0	42	65
3^d	1	1	1	0	0	infeasible	65

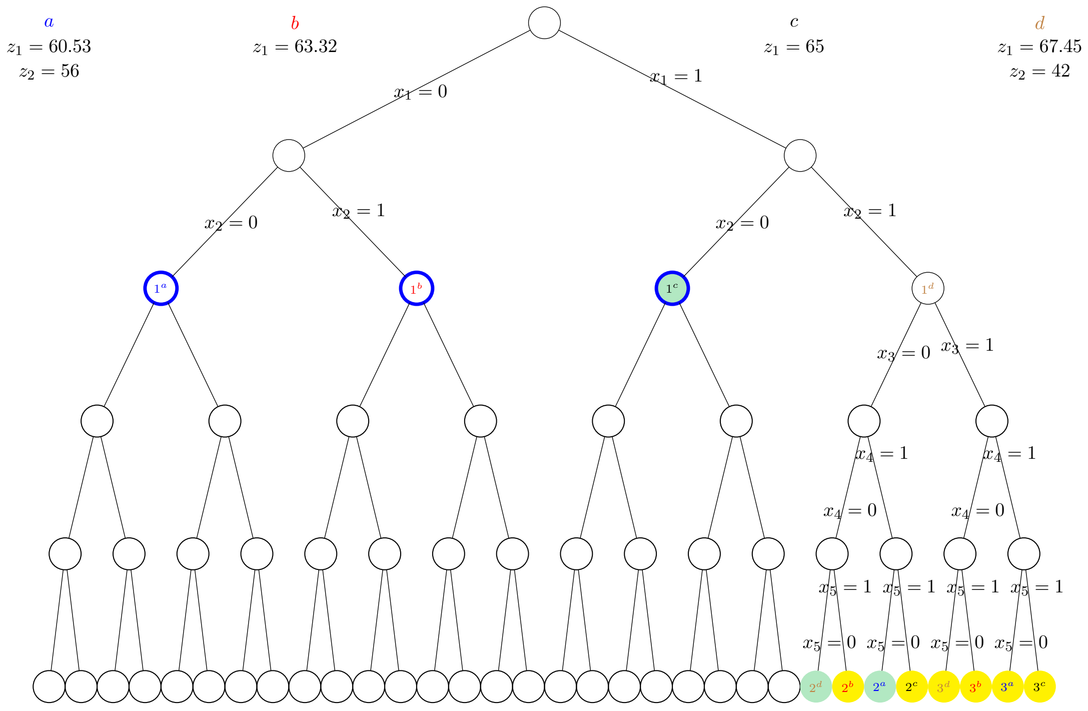

Figure 2.10: Knapsack problem example. Four-path B&B

Let the four paths be notated a, b, c and d, and keep the same branching criterion. In this context, the starting point of the branching will be situated at the second level of the tree, saving the branching of variables x_1 and x_2. The resolution of the LP relaxation of the first nodes gives the function values: 60.53, 63.32, 65 and 67.45. All solutions are gathered by the paths and note that the solution of Path *c* is feasible and the value is greater than those from Path *a* and Path *b*, consequently Path *a*, Path *b* and Path *c* can be locally pruned and they are now dead. These three paths are reassigned to Path *d* side, creating a same level branching for nodes 2^a, 2^b, 2^c and 2^d. Once the related LP relaxation is solved two feasible solutions (2^a and 2^d) have been obtained but none of them improves the incumbent solution value. Given that those nodes are already at the last level, Path *a*, Path *b* and Path *c* are again dead, and then they are reassigned to Path *d*. All 3^a, 3^b, 3^c and 3^d are infeasible and, therefore, all paths are dead, and so, the algorithm stops. See Table 2.11 and branching tree in Figure 2.10

In addition to the aspects underlined at the previous strategies, the four-path strategy shows a significant reduction of visited nodes for each path (from 10 in serial to 3 in this case).

2.5.2 Stage perspective

The second key aspect referred to at the beginning of Section 2.5, corresponds to the stage perspective. Most of the state-of-the-art solvers (with CPLEX being one of them) have no structural vision of the stochastic problem, and therefore, they do not benefit from the variable hierarchy. However, decomposition algorithms can take advantage of this information.

Let us analyse the potential effect of the variable order when solving a stochastic mixed 0-1 problem. To do so let us take the Multipath Branch & Bound algorithm as reference. In fact, preliminary tests can be enriched by adding preprocessing to Algorithm 2.1, see also Figure 2.7; so that initially tighter bound can be obtained and the values of the variables in a quasioptimal solution can be very useful for reducing the computations in the algorithm to obtain the optimal solution.

Four possible set ups regarding the variable fixing criterion: always to zero (0), as presented on the knapsack example, always to one (1), the values obtained at the quasioptimal solution (Q) and the rounded solution of the LP relaxation (round(LR)).

Computational experience

Let us consider a mixed 0-1 problem in order to test the effect of set up parameters. Specifically, let problem P11 taken from the computational experience reported in Escudero *et al.* [2012a] and later included in Testbed 2 in Table 3.2. By applying a break stage clustering process with t^* (see Section 3.2), 5 subproblems are generated. The selected subproblem is the one related to cluster number 1.

The computational experiments were conducted in the ARINA computational cluster at SGI/IZO-SGIker from UPV/EHU; using two Intel Xeon nodes with 8 cores and 48Gb of RAM (remember Table 1.1). The LP relaxation problems were solved by using CPLEX V12.2 from COIN-OR V1.3.1. The environment corresponds to 16 paths (using a single CPLEX auxiliary thread each) of Algorithm 2.1 (see also Figure 2.7) extended with an initial CPLEX preprocessing of 1% optimality gap.

Table 2.12 shows the computing time (in sec) required for each fixing criterion and variable ordering. The headings are as follows: *nominal*, variables ordered by indexes; *max to min difference*, variables with largest difference between values at LP relaxation and quasioptimal solution first; *min to max difference*, variables with lowest difference between values at LP relaxation and quasioptimal solution first.

It can be observed that the stage ordering is significantly faster than the group ordering. The fast execution of the stage ordering does not allow behaviour conclusions to be extracted, regarding the variable order or the fixing criterion. On the other hand, the group ordering shows that nominal variable ordering is more efficient than the ones linked to the increasing and decreasing differences between quasioptimal and LP relaxation solutions. Additionally, the fastest executions correspond to fixing variables according to the quasioptimal or rounded LP relaxation solution. However, as no branching criterion appears to significantly outperform the others, further research and computational experience is analysed in Chapter 5.

Table 2.12: Effect of the variable order and the fixing criterion

	Elapsed time (in seconds)					
	Group-wise ordering			**Stage-wise ordering**		
FIXING CRITERION	**nominal**	**max to min difference**	**min to max difference**	**nominal**	**max to min difference**	**min to max difference**
0	220	242	552	9	9	10
1	62	118	1001	7	9	8
Q	81	172	551	10	9	10
round(LR)	78	176	552	9	10	10

2.6 Outer parallelization paradigm

The Multipath Branch & Bound approach needs to analyse the whole variable tree in order to ensure optimality. If larger scale problems are considered, this aspect turns to be the main drawback and a significant obstacle to obtain the optimal solution in an acceptable elapsed time. However, the parallel branching environment presented in this chapter is applicable to other branching algorithms that can deal with larger problems; notably, parallel branching algorithm based on decomposition.

More generally, let us name *outer parallelization* paradigm to an environment where simultaneous executions of algorithms are interconnected. As in the Multipath Branch & Bound algorithm, each algorithm execution, or path, can have a different initial condition and can search solutions in a different part of the feasibility region. Communication among paths will allow each algorithm execution to take profit from more information than the one locally obtained by the algorithm.

However, the enormous potential of the outer parallelization is based on the possibility of having different nature serial algorithms co-working in parallel by communication. The hybrid environments create a large research area where parallel-conceived algorithms will profit from different paths working with a different role, such as exact and heuristic combinations.

2.7 Conclusions

As the results of the preliminary testing have shown, the hardware environment control is a key aspect in terms of obtaining satisfactory results. This control can be summarized in three aspects. First, user selection of the involved threads, trying to use architecturally close threads such as threads from the same computing node. Second, the behaviour tests should increase the number of threads by power of two, following the way threads are built to be used. Third, in order to avoid chaotic simultaneous executions it is highly recommended to introduce intermediate communication functions, used as synchronization phases.

Regarding the parallel execution of several mixed 0-1 optimization problems the most important conclusion that we can withdraw is that distributed memory parallelization by MPI is a very powerful tool for reducing the execution time. The improvement is consistent when increasing the number of task threads in both COIN-OR and CPLEX solvers that we have experimented with, as shown in Table 2.5 and Table 2.6.

Furthermore, Table 2.8 reports that CPLEX is a significantly faster solver than COIN-OR in the serial execution and has a better speedup and efficiency in parallel environments. Consequently, from now on in this work problems will be solved by CPLEX (from COIN-OR environment) and, then the use of COIN-OR as solver is discarded.

The results showed in Table 2.8 for CPLEX are highly encouraging for further research, since its scaling behaviour is highly significant in both computing time and elapsed time scales, achieving a speedup factor of 20.751 and an efficiency of 32.4 % when using 64 threads.

Additionally, as previously reported in Figure 2.4, it is possible to combine user computed distributed memory with solver internal shared memory parallelization (i.e., CPLEX in the computational experience that has been reported). A group of executions has been carried out and summarized in Table 2.7 by fixing in each case the number of threads and varying the number of them associated to distributed and shared memory. In this case we can conclude that it is more efficient to assign threads to the distributed memory paradigm, i.e., task threads, rather than to combine it with CPLEX internal share memory paradigm, i.e., auxiliary threads. However, it is very important to underline that we are considering a group of small scale optimization problems and consequently we cannot conclude that its behaviour will be the same for large scale problems.

We can therefore conclude that an efficient way has been introduced for parallel solving of several small scale mixed 0-1 optimization problems. Interestingly, this environment can be used for parallel subproblem solving of decomposition algorithms, i.e., an inner parallelization strategy; see Chapter 3 and Chapter 4 for detailed implementations.

The outer parallelization strategy and logic differs from the inner parallelization by changing the focus from the micro scale to the macro scale, particularly in our context, from solving in parallel stochastic problems to branching the tree with multiple paths. Based on an illustrative example of the 0-1 Knapsack problem we have numerically presented the algorithm steps and diagram of the outer parallelization of the Branch & Bound algorithm. From that example we can observe that the Multipath B&B approach allows to reduce the elapsed time by visiting nodes in parallel, gathering intermediate solutions and reassigning dead paths. The use of that scheme seems really appropriate for small and medium scale B&B node problems since it has a more intensive use of threads than the inner parallelization.

Additionally, we have analysed the effect of the variable ordering and fixing criterion on the algorithm performance. The stage-wise variable ordering shows a significantly faster behaviour but no clear branching criterion has been identified from the preliminary computational experience.

In this chapter we have shown that parallel branching executions can be managed in a coordinated way and it can profit from the problem structure. It seems appropriate to go further in this analysis to try to solve larger problems, so the coordinated branching and the break stage concepts will work in this direction for the Branch-and-Fix Coordination algorithm considered in Chapter 3.

Chapter 3

Parallel Computing Branch-and-Fix Coordination algorithm

Caminante, son tus huellas	Wanderer, your footsteps are
el camino, y nada más;	the road, and nothing more;
caminante, no hay camino,	wanderer, there is no road,
se hace camino al andar.	the road is made by walking.

Antonio Machado, *Campos de Castilla.*

3.1 Introduction

The aim of this chapter is to present the parallelized version of the Branch-and-Fix Coordination multistage (BFC) algorithm, see Escudero *et al.* [2012a], referred to as Parallel Branch-and-Fix Coordination multistage (P-BFC) procedure, so that the reduction in the elapsed time in problem solving is analyzed. The parallelization is performed at two levels. The inner level parallelizes the optimization of the Mixed Integer Programming (MIP) submodels attached to the set of scenario clusters to be generated by the modeler-defined break stage. The concept of *break stage* was introduced in Escudero *et al.* [2012a] as a way of decomposing the original problem, in which the nonanticipativity constraints (NAC) are partially relaxed from the mixture of the splitting and compact representations of the DEM of the stochastic problem. Several strategies are presented for analyzing the performance of using inner parallel computing based on MPI strategies for solving scenario cluster based subproblems versus the serial version of the BFC methodology. The outer level of parallelization is based on the path concept, see Section 1.3. In this chapter a path is defined by a main thread managing a serial Branch-and-Fix Coordination algorithm and the combinations of a set of 0-1 variables as initial condition, such that each one can itself be internally optimized with the inner parallelization scheme.

The main results of a broad computational experience are reported to assess whether the performance of the parallel computing approach compares favorably to the serial one. The elapsed time required by outer-inner parallelization is very frequently some orders of magnitude smaller than that of the serial version of the algorithm, depending on the available computer resources . So, the larger the number of paths and task threads (in addition to the number of auxiliary threads that the MIP solver allows for itself), the smaller the elapsed time for problem solving.

The rest of the chapter is organized as follows: Section 3.2 presents the main concepts of the break stage scenario clustering. Section 3.3 introduces the serial version of the scenario cluster BFC algorithm introduced in Escudero *et al.* [2010b, 2012a], additionally, the Parallel Computing implementations of the BFC algorithm are described, Inner (Section 3.4), Outer (Section 3.5) and combined Outer-Inner (Section 3.6). Section 3.7 reports the main results of a broad computational experience to assess the validity of the parallel version of the BFC algorithm versus its serial version and the plain use of a state-of-the-art MIP solver. Section 3.8 summarizes the main conclusions.

3.2 Break stage scenario clustering

The *Break stage scenario clustering* methodology consists of breaking the stochastic model in a set of independent scenario cluster subproblems with respect to a fixed stage, which is so named the *break stage*, due to the relaxation of the nonanticipativity constraints (NAC) up to that stage, see Escudero *et al.* [2012a].

It is clear that the explicit representation of the NAC $x_t^\omega - x_t^{\omega'} = 0$ and $y_t^\omega - y_t^{\omega'} = 0$ in model (1.5) is not desirable for all pairs (ω, ω') of scenarios in order to reduce the model's dimensions. So, we can represent implicitly the NAC for some pairs of scenarios in order to gain computational efficiency. We decompose the scenario tree into a set of scenario cluster subtrees, each one for a scenario cluster in the set denoted as $\mathcal{C}= \{1,...,C\}$ with $C = |\mathcal{C}|$, see below the reason for it. Let Ω_c denote the set of scenarios that belongs to cluster c, such that $\Omega_c \bigcap \Omega_{c'} = \emptyset$, $c, c' \in \mathcal{C} : c \neq c'$ and $\Omega = \cup_{c \in \mathcal{C}} \Omega_c$.

We propose to choose the number of scenario clusters C as any value from the subset $\{|\mathcal{G}_1|, |\mathcal{G}_2|, \dots, |\mathcal{G}_T|\}$, so, let us consider the following definitions.

Definition 3.1. *A* **break stage** t^* *is a stage t such that the number of scenario clusters is $C = |\mathcal{G}_{t^*+1}|$, where $t^*+1 \in \mathcal{T}$. In this case, any cluster $c \in \mathcal{C}$ is induced by* *a group $g \in \mathcal{G}_{t^*+1}$ and contains all scenarios belonging to that group, i.e., $\Omega_c = \Omega^g$.*

Without loss of generality, the break stage concept assumes that for multiperiod stages, the scenario cluster decomposition is considered at the first period of the corresponding stage.

Definition 3.2. *The* ***scenario cluster decomposition*** *submodels are those that result from the relaxation of the NAC until the break stage t^* in model* (1.5).

Notice that the choice of $t^* = 0$ corresponds to the full DEM and $t^* = T - 1$ corresponds to the scenario partitioning. The reason for the decomposition of the scenario set into scenario clusters is based on the way in which the decomposition algorithm BFC works, see Escudero *et al.* [2010b, 2012a,b]. It considers the explicit NAC related to the stages t from 1 until t^*, since the NAC from its next stage are implicitly considered while solving the scenario cluster MIP submodels.

Let us assume that we have broken down the scenario set into C scenario clusters. Now, let us formulate the cluster submodels and the full DEM via a mixture of the splitting variable and compact representations, see Section 1.7, so that the submodels are linked by the explicit NAC up to stage t^*. For doing so, let us slightly abuse the notation such that $\mathbf{x}_t^c$ and $\mathbf{y}_t^c$ denote the vectors of the 0-1 and continuous variables, respectively, for scenario cluster $c \in \mathcal{C}$ and stage $t \in \mathcal{T}$, $\mathbf{a}_t^c$ and $\mathbf{b}_t^c$ are the vectors of the objective function coefficients of the variables vectors $\mathbf{x}_t^c$ and $\mathbf{y}_t^c$, and $\mathbf{nx}_t^c$ and $\mathbf{ny}_t^c$ denote the number of 0-1 and continuous variables, respectively, for the pair (c, t). Similarly, let $\mathbf{h}_t^c$ denote the new *rhs*. Additionally, let $\mathcal{G}^c \subseteq \mathcal{G}$ denote the set of scenario groups for cluster c, such that $\Omega^g \cap \Omega_c \neq \emptyset$ means that $g \in \mathcal{G}^c$, and let $\mathcal{G}_t^c = \mathcal{G}_t \cap \mathcal{G}^c$ denote the set of scenario groups for cluster $c \in \mathcal{C}$ in stage $t \in \mathcal{T}$.

For example, if we consider the break stage $t^* = 2$ in Figure 3.1, we obtain $C = 4$ clusters submodels, where for cluster $c = 3$, $\mathcal{G}^3 = \{1, 3, 6, 11, 12, 13\}$, $\mathcal{G}_4^3 = \{11, 12, 13\}$ and $\Omega_3 = \{4, 5, 6\}$.

The set of constraints in model (1.5) for each scenario cluster can be split into two blocks. The first one represents the constraints related to the vectors of variables from stage $t = 1$ until stage $t^* + 1$ (i.e., stages with explicit NAC), since those variables must be linked with their own replicas in the appropriate clusters in set $\mathcal{C}$. The second block represents the constraints related to the vectors of variables from stage $t^* + 2$ until the last stage T (i.e., stages with implicit NAC). Accordingly, the MIP submodel for cluster $c \in \mathcal{C}$ can be formulated as follows,

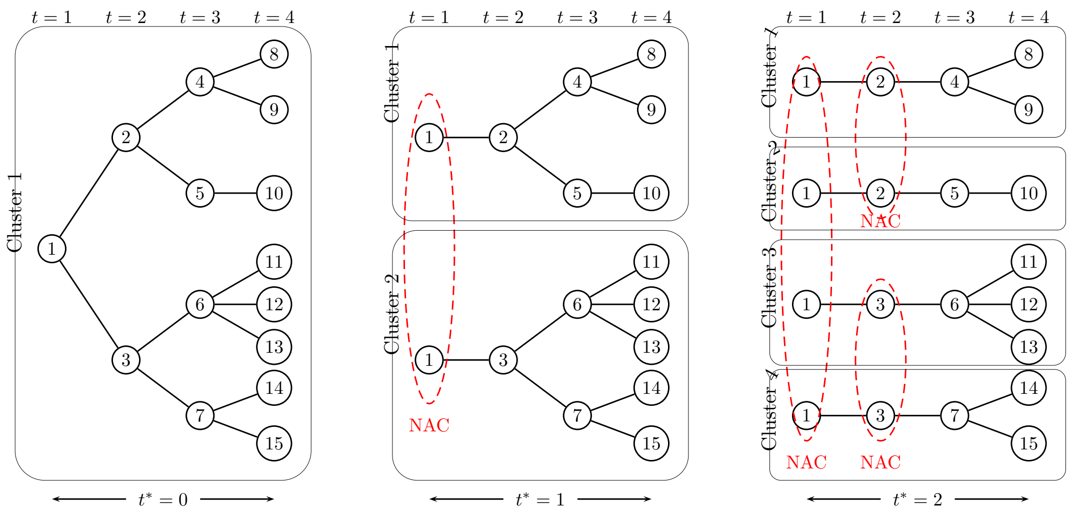

Figure 3.1: Scenario clustering for $t^* = 0$, $t^* = 1$ and $t^* = 2$

$$\begin{aligned} z^c_{t^*} &= \min \sum_{t\in\mathcal{T}} \mathbf{w}^c_t(\mathbf{a}^c_t\mathbf{x}^c_t + \mathbf{b}^c_t\mathbf{y}^c_t) \\ \text{s.t.} \quad & \text{cluster c constraint system (3.2)} \\ & \mathbf{x}^c_t \in \{0,1\}^{\mathbf{nx}^c_t}, \quad \mathbf{y}^c_t \in \mathbb{R}^{+\mathbf{ny}^c_t} \qquad \forall t \in \mathcal{T}. \end{aligned} \tag{3.1}$$

where the constraint system (3.2) is as follows:

$$\begin{aligned} \sum_{t'\in\mathcal{T}^t} \left(\mathbf{A}^{t,c}_{t'}\mathbf{x}^c_{t'} + +\mathbf{B}^{t,c}_{t'}\mathbf{y}^c_{t'}\right) &= \mathbf{h}^c_t \qquad \forall t \in \mathcal{T} : t \le t^* + 1 \\ \sum_{t'\in\mathcal{T}^t} \left([\mathbf{A}^t_{t'}]^c\mathbf{x}^c_{t'} + [\mathbf{B}^t_{t'}]^c\mathbf{y}^c_{t'}\right) &= \mathbf{h}^c_t \qquad \forall t \in \mathcal{T} : t > t^* + 1 \end{aligned} \tag{3.2}$$

See in Escudero *et al.* [2010b] the details for computing the weight parameter $\mathbf{w}^c_t$ for $t \le t^* + 1$ and the weight vector $\mathbf{w}^c_t$ for $t^* + 1 < t \le T$. Additionally, the first block of constraints matrices $(\mathbf{A}^{t,c}_{t'}, \mathbf{B}^{t,c}_{t'})$ is related to the pair stage t and cluster c, for $t \le t^* + 1$. The second block represents the constraints for stages from $t^* + 2$ until stage T, for $c \in \mathcal{C}$. The constraints matrices $[\mathbf{A}^t_{t'}]^c$ and $[\mathbf{B}^t_{t'}]^c$ can be split into the $|\mathcal{G}^c_{t-1}|$ and $|\mathcal{G}^c_t|$ submatrices related to the scenario groups in stage t for cluster c, respectively; see Escudero *et al.* [2012a] for the details on a slight particular case.

Notice that the nonanticipativity principle is implicitly taken into account for the stages from $t^* + 1$ until stage T, since the submodel for each cluster is formulated via a compact representation.

Let us split the set of stages $\mathcal{T}$ in two subsets, such that $\mathcal{T} = \mathcal{T}_1 \bigcup \mathcal{T}_2$, where $\mathcal{T}_1 = \{1, \dots, t^*\}$, and $\mathcal{T}_2 = \{t^* + 1, \dots, T\}$. For modeling the (explicit) NAC of the scenario clusters for the stages up to the break stage t^*, let the following definition of the NAC based cluster set.

Definition 3.3. *The* **NAC based cluster set***, say, $\mathcal{C}^g$ is the set of clusters that have the scenario group g in common, for $g \in \mathcal{G}_t$, $t \leq t^*$.*

Notice that clusters c and c' belong to set $\mathcal{C}^g$ iff $g \in \mathcal{G}_t^c \cap \mathcal{G}_t^{c'}$. The cluster submodels (3.1) are linked by the NAC to be formulated as follows,

$$\mathbf{x}_t^c - \mathbf{x}_t^{c'} = 0 \quad \forall c, c' \in \mathcal{C}^g, g \in \mathcal{G}_t, t \in \mathcal{T}_1 \tag{3.3}$$

$$\mathbf{y}_t^c - \mathbf{y}_t^{c'} = 0 \quad \forall c, c' \in \mathcal{C}^g, g \in \mathcal{G}_t, t \in \mathcal{T}_1. \tag{3.4}$$

The full DEM can be represented by a mixture of the splitting variable representation (for explicitly satisfying the NAC between the cluster submodels) and the compact representation (for implicitly satisfying the NAC of each cluster submodel, besides the other constraints in the submodel). So, the solution value $z_{t^*}^{DEM}$ of the full DEM can be obtained as the sum of the solution values of the scenario cluster submodels, $z_{t^*}^c$ (3.1) $\forall c \in \mathcal{C}$ plus the NAC (3.3)-(3.4) between the clusters.

By using the previous elements, the full DEM can be formulated in a *cluster splitting-compact representation* as follows,

$$\begin{array}{lll} & z_{t^*}^{DEM} = \min \sum\limits_{c \in \mathcal{C}} \sum\limits_{t \in \mathcal{T}} \mathbf{w}_t^c (\mathbf{a}_t^c \mathbf{x}_t^c + \mathbf{b}_t^c \mathbf{y}_t^c) & \\ \text{s.t.} & \text{cluster c constraint system (3.2)} & \forall c \in \mathcal{C} \\ & \mathbf{x}_t^c = \mathbf{x}_t^{c'} & \forall c, c' \in \mathcal{C}^g,\ g \in \mathcal{G}_t, t \in \mathcal{T}_1 \\ & \mathbf{y}_t^c = \mathbf{y}_t^{c'} & \forall c, c' \in \mathcal{C}^g,\ g \in \mathcal{G}_t, t \in \mathcal{T}_1 \\ & \mathbf{x}_t^c \in \{0,1\}^{\mathbf{nx}_t^c},\ \ \mathbf{y}_t^c \in \mathbb{R}^{+\mathbf{ny}_t^c} & \forall c \in \mathcal{C}, t \in \mathcal{T}. \end{array} \tag{3.5}$$

For the previous example corresponding to break stage $t^* = 2$ in Figure 1.2, $\mathcal{T}_1 = \{1, 2\}$ and $\mathcal{T}_2 = \{3, 4\}$. So, $\mathcal{C}^1 = \{1, 2, 3, 4, 5\} = \mathcal{C}$, $\mathcal{C}^2 = \{1, 2\}$, $\mathcal{C}^3 = \{3\}$ and $\mathcal{C}^4 = \{4, 5\}$.

Definition 3.4. *For a given $t^* \in \{0, 1, ..., T - 1\}$, let $C = C(t^*) = |\mathcal{G}_{t^*+1}|$ denote the number of scenario clusters, such that*

1. $z^0_{t^*} = \sum_{c=1}^{C(t^*)} z^c_{t^*}$ *gives the lower bound of the solution value of the original problem* (3.5) *based on the relaxation of the NAC for the stages* $t \in \mathcal{T}_1$*, i.e., the sum of the solution values* $z^c_{t^*}$ *for each scenario cluster model (3.1).*

2. $GAP^0_{t^*} = 100 \left| \frac{z^{DEM} - z^0_{t^*}}{z^0_{t^*}} \right|$ *defines the optimality gap (in %) of the solution value* $z^0_{t^*}$.

Proposition 1. *The following equalities and inequalities are satisfied in minimization problems (and therefore, the opposite inequalities in maximization problems):*

1. $z^0_0 \geq z^0_1 \geq z^0_2 \geq \ldots \geq z^0_{T-1}$
2. *For any* $t^* \in \{0, 1, 2, \ldots, T-1\}$*, it results* $z^{DEM} = z^{DEM}_{t^*}$, $z^{DEM} = z^0_0$ *and* $z^0_{t^*} \leq z^{DEM}_{t^*}$
3. *So,* $GAP^0_0 \leq GAP^0_1 \leq GAP^0_2 \leq \ldots \leq GAP^0_{T-1}$

Proof 1. *Notice that 1. is evident since two consecutive problems in the stages' chain only differ on the additional relaxation of the NAC corresponding to the later stage. 2. follows from definitions and the fact that the solution value* $z^{DEM}_{t^*}$ *of the full model* (3.5) *satisfies also the NAC for the stages in set* $\mathcal{T}_1$*. And 3. follows from the previous ones.*

So, the smaller the break stage t^* is, the stronger the $GAP^0_{t^*}$ is. However, the computational effort seems bigger for smaller t^*-decompositions. So, a computational analysis of GAPs and their efficiency is required for estimating the appropriate interval for the t^*-decomposition, see Section 3.7.2.

3.3 BFC decomposition algorithm

3.3.1 Basic definitions and auxiliary models

The main concepts of the BFC methodology were introduced in Alonso-Ayuso *et al.* [2003a,b] and subsequently refined mainly in Escudero *et al.* [2010b], particularly:

Definition 3.5. *A* **Branch-and-Fix tree** *($\mathcal{BFT}$) associated with any scenario cluster is the Branch-and-Bound (B&B) tree for optimizing the models* (3.1) *in a coordinated way with the models of the other clusters, such that the related full DEM* (3.5) *is solved up to optimality.*

As an additional notation, let $\mathcal{BFT}^c$ denote the *B*&*B* tree associated with scenario cluster c, $\mathcal{Q}^c$ be the set of active nodes in $\mathcal{BFT}^c$, $\mathcal{I}$ be the set of indices of the variables in vector $\mathbf{x}^c_t$, and x^c_{ti} be the ith variable in $\mathbf{x}^c_t$, for $c \in \mathcal{C}, t \in \mathcal{T}, i \in \mathcal{I}$.

Definition 3.6. *Two variables, say, x_{ti}^c and $x_{ti}^{c'}$ are said to be* **common variables** *for the scenario clusters c and c', if $c, c' \in \mathcal{C}^g | c \neq c', g \in \mathcal{G}_t, t \in \mathcal{T}, i \in \mathcal{I}$.*

Notice that the common variables have nonzero elements in the NAC related to a given scenario group.

Definition 3.7. *Any two nodes, say, $q \in \mathcal{Q}^c$ and $q' \in \mathcal{Q}^{c'}$ are said to be* **twin nodes** *if on the paths from the root node to each of them in their trees $\mathcal{BFT}^c$ and $\mathcal{BFT}^{c'}$, the common variables x_{ti}^c and $x_{ti}^{c'}$, if any, have been branched on / fixed at the same 0-1 value, for $c, c' \in \mathcal{C}^g | c \neq c', g \in \mathcal{G}_t, t \in \mathcal{T}, i \in \mathcal{I}$.*

Definition 3.8. *A* **Twin Node Family** *(TNF) $\mathcal{J}_f$ $\mathcal{J}_f$ is a set of nodes such that any node is a* twin *node to all the other node members in the family, for $f \in \mathcal{F}$, where $\mathcal{F}$ is the set of the families.*

Definition 3.9. *A* **candidate TNF** *is a TNF whose members have not yet branched on / fixed at all their common variables.*

Definition 3.10. *An* **integer TNF** *is a* TNF *where all x variables take integer values and the NAC* (3.3) *are satisfied.*

In each integer TNF the LP model (3.6) is solved to obtain a feasible solution for the original DEM (3.5). Notice that all the **x** variables for the stages from set $\mathcal{T}_1$ in the BFC algorithm are set up to 0-1 values in any integer TNF by either algorithmic branching, fixing as branching implications or taking 0-1 variables in the submodels (3.1). Additionally, model (3.6) defines the **y** variables for the stages in set $\mathcal{T}_1$ whose NAC (3.4) are to be explicitly satisfied. In fact, model (3.6) results from fixing the **x** variables for the all stages in set $\mathcal{T}$ at their 0-1 values in DEM (3.5). Let x denote the 0-1 vector for the values of vector **x**. The cluster splitting-compact representation of the model is as follows:

$$
\begin{array}{lll}
& z_{LP}^{TNF} = \min \sum_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} \mathbf{w}_t^c (\mathbf{a}_t^c \mathbf{x}_t^c + \mathbf{b}_t^c \mathbf{y}_t^c) & \\
\text{s.t.} & \text{cluster c constraint system (3.2)} & \forall c \in \mathcal{C} \\
& \mathbf{x}_t^c = x_t^c & \forall c \in \mathcal{C}, t \in \mathcal{T} \\
& \mathbf{y}_t^c = \mathbf{y}_t^{c'} & \forall c, c' \in \mathcal{C}^g,\ g \in \mathcal{G}_t, t \in \mathcal{T}_1 \\
& \mathbf{y}_t^c \in \mathbb{R}^{+\mathbf{n}y_t^c} & \forall c \in \mathcal{C}, t \in \mathcal{T}.
\end{array}
\tag{3.6}
$$

If the LP model (3.6) is feasible with solution (x, y^{TNF}), a new MIP auxiliary submodel can be defined by fixing the variables $\mathbf{x}_t^c$ and $\mathbf{y}_t^c$ to the values x_t^c and

$y_t^{c,TNF}$, respectively $\forall c \in \mathcal{C}, t \in \mathcal{T}_1$ in order to obtain a better solution value of the original DEM,

$$\begin{array}{lll} & z_f^{TNF} = \min \sum_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} \mathbf{w}_t^c (\mathbf{a}_t^c \mathbf{x}_t^c + \mathbf{b}_t^c \mathbf{y}_t^c) & \\ \text{s.t.} & \text{cluster c constraint system (3.2)} & \forall c \in \mathcal{C} \\ & \mathbf{x}_t^c = x_t^c & \forall c \in \mathcal{C}, t \in \mathcal{T}_1 \\ & \mathbf{y}_t^c = y_t^{c,TNF} & \forall c \in \mathcal{C}, t \in \mathcal{T}_1 \\ & \mathbf{x}_t^c \in \{0,1\}^{\mathbf{n}x_t^c} & \forall c \in \mathcal{C}, t \in \mathcal{T}_2 \\ & \mathbf{y}_t^c \in \mathbb{R}^{+\mathbf{n}y_t^c} & \forall c \in \mathcal{C}, t \in \mathcal{T}_2, \end{array} \tag{3.7}$$

from which the independent MIP submodels for scenario cluster $c \in \mathcal{C}$ can be expressed

$$\begin{array}{lll} & z_{fc}^{TNF} = \min \sum_{t \in \mathcal{T}_1} \mathbf{w}_t^c (\mathbf{a}_t^c x_t^c + \mathbf{b}_t^c y_t^{c,TNF}) + \sum_{t \in \mathcal{T}_2} \mathbf{w}_t^c (\mathbf{a}_t^c \mathbf{x}_t^c + \mathbf{b}_t^c \mathbf{y}_t^c) & \\ \text{s.t.} & \text{cluster c constraint system (3.2)} & \\ & \mathbf{x}_t^c = x_t^c & \forall t \in \mathcal{T}_1 \\ & \mathbf{y}_t^c = y_t^{c,TNF} & \forall t \in \mathcal{T}_1 \\ & \mathbf{x}_t^c \in \{0,1\}^{\mathbf{n}x_t^c} & \forall t \in \mathcal{T}_2 \\ & \mathbf{y}_t^c \in \mathbb{R}^{+\mathbf{n}y_t^c} & \forall t \in \mathcal{T}_2. \end{array} \tag{3.8}$$

Notice that the NAC for $\mathbf{x}$ and $\mathbf{y}$ variables for the stages in $\mathcal{T}_1$ are satisfied for the values x_t and $y_t{}^{TNF}$ obtained by solving model (3.6). So, z_f^{TNF} in model (3.7) can be expressed as follows:

$$z_f^{TNF} = \sum_{c \in \mathcal{C}} z_{fc}^{TNF}. \tag{3.9}$$

Let $x_{\overline{t}i}^c \, \forall c \in \mathcal{C}^g$ denote the last branched variable in the integer TNF under consideration, g is the related branching scenario group in set $\mathcal{G}_{\overline{t}}$ and $\overline{t}$ is the branching stage from set $\mathcal{T}_1$ (obviously, $1 \leq \overline{t} \leq t^*$) and $1 \leq i \leq nx_{\overline{t}}$. Additionally, define MIP model (3.10), so that for the presentation of the model, let $\mathcal{J}_1$ denote the set $\{j = 1, ..., nx_t^c, 1 \leq t < \overline{t} \text{ and } 1 \leq j \leq i \text{ for } t = \overline{t}\}$, and $\mathcal{J}_2$ is the set $\{i < j \leq nx_t^c \text{ for } t = \overline{t} \text{ and } j = 1, ..., nx_t^c, \overline{t} < t \leq T\}$.

$$\begin{array}{lll} & z_{MIP}^{TNF} = \min \sum_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} \mathbf{w}_t^c (\mathbf{a}_t^c \mathbf{x}_t^c + \mathbf{b}_t^c \mathbf{y}_t^c) & \\ \text{s.t.} & \text{cluster c constraint system (3.2)} & \forall c \in \mathcal{C} \\ & \mathbf{x}_{tj}^c = x_{tj}^c & \forall c \in \mathcal{C}, (tj) \in \mathcal{J}_1 \\ & \mathbf{y}_t^c = \mathbf{y}_t^{c'} & \forall c, c' \in \mathcal{C}^g, g \in \mathcal{G}_t, t \in \mathcal{T}_1 \\ & \mathbf{x}_{tj}^c \in \{0,1\}^{\mathbf{n}x_t^c} & \forall c \in \mathcal{C}, (tj) \in \mathcal{J}_2 \\ & \mathbf{y}_t^c \in \mathbb{R}^{+\mathbf{n}y_t^c} & \forall c \in \mathcal{C}, t \in \mathcal{T}, \end{array} \tag{3.10}$$

where the bound z_f^{TNF} is used as a cut-off to reduce the elapsed time of the solver.

It is straightforward to prove that there is no a better (in this case, smaller) solution value in any descendant TNF integer model (3.10) from the current integer TNF than the related value z_{MIP}^{TNF} obtained at that TNF integer set in the branching process presented below.

Proposition 2. *In any TNF and for any breakstage t^*, where $t^* + 1 \in \mathcal{T}$, the following inequalities are satisfied in minimization problems:*

$$z_{t^*}^{DEM} \leq z_{MIP}^{TNF} \leq z_f^{TNF} \leq z_{LP}^{TNF}$$

Proof 2. *The first inequality, $z_{t^*}^{DEM} \leq z_{MIP}^{TNF}$, holds because any feasible solution of model* (3.10) *is also a solution of model* (3.5)*, since the feasible region of the former problem has one set of constraints more than the latter problem, namely, the set where the $0-1$ variables from $\mathcal{J}_1$ have been fixed. The second inequality, $z_{MIP}^{TNF} \leq z_f^{TNF}$, holds because any feasible solution of model* (3.7) *is also a solution of model* (3.10)*, since the feasible region of the former problem has one set of constraints more than the latter problem, namely, the set where the continuous variables from $\mathcal{J}_1$ have been fixed. And the third inequality, $z_f^{TNF} \leq z_{LP}^{TNF}$, holds because any feasible solution of model* (3.6) *is also a solution of model* (3.7)*, since the feasible region of the former problem has one set of constraints more than the latter problem, namely, the set where the $0-1$ variables from $\mathcal{J}_2$ have been fixed.*

So, the (3.5)*,* (3.6) *and* (3.10) *models are upper bounds on the solution value of de DEM model* (3.5)*, as it is used in Step 7 of the Algorithm 3.1.*

Proposition 3. *Let us denote z_{f,t^*}^{TNF} the solution value of model* (3.7)*, where $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$ partitioning is related to break stage t^*. In any TNF, the following inequalities are satisfied in minimization problems:*

$$z_{f,1}^{TNF} \leq z_{f,2}^{TNF} \leq \ldots \leq z_{f,t^*}^{TNF} \leq \ldots \leq z_{f,T-1}^{TNF}$$

Proof 3. *$z_{f,t^*}^{TNF} \leq z_{f,t^*+1}^{TNF}$ follows because the feasible region of the former problem has two set of constraints more than the latter problem, namely, the set where the $0-1$ and continuous variables from $t^* + 1$ have been fixed.*

So, the earlier the breakstage t^*, the better the tightening of the upper bounds.

3.3.2 Procedure

The procedure of the serial Branch-and-Fix Coordination multistage (BFC) is presented in Algorithm 3.1. It considers the submodels (3.8) $\forall c \in \mathcal{C}$ as the main advantage over the serial version presented in Escudero *et al.* [2012a].

The initialization phase in Step 0 starts by solving the LP relaxation of the original DEM (3.5). If it is infeasible, therefore it is the given original model; on the contrary, if the integer constraints of the DEM are satisfied then the optimal solution has been found, in either case the execution stops. Otherwise the main variables are initialize and the BFC algorithm starts.

The root node of the branch-and-fix tree, $\mathcal{BFT}$, is analysed in Step 1. The MIP submodels (3.1) are solved to obtain $z^c \, \forall c \in \mathcal{C}$ and compute $z^0_{t^*} = \sum_{c \in \mathcal{C}} z^c$. If both x^c and y^c satisfy NAC, (3.3) and (3.4), then the optimal solution has been found and the execution ends. Otherwise, variable branching starts.

The forward branching phase is managed by Step 2 to Step 5, where the stage, the scenario group and the binary variable indexes are controlled.

Whenever a binary variable is branched, the corresponding candidate TNF is analysed in Step 6. As in Step 1, the MIP submodels (3.1) are solved in order to compute a bound of the original problem; if the bound does not improve the incumbent solution, then the branch is pruned in Step 9. Otherwise, if the x^c variables do not satisfy NAC (3.3), then the branching process continues. If x^c variables satisfy NAC (3.3) but the y^c variables do not satisfy NAC (3.4), then the integer TNF models are analysed. If both x^c and y^c satisfy NAC, (3.3) and (3.4), then a feasible solution has been found and the branch is pruned in Step 8.

The integer TNF models (3.6), (3.8) and (3.10) are solved in Step 7 in order to get a feasible solution of the original DEM (3.5) once the x^c variables satisfy NAC (3.3).

The backward branching and the branch pruning is managed is Step 8 to Step 11, where the stage, the scenario group and the binary variable indexes are controlled.

Algorithm 3.1: Serial BFC

```
Step 0: (Initializations)
        Solve the LP relaxation of the original DEM (3.5) to obtain $z_{LP}$.
        If it is infeasible, therefore it is the given original model, then STOP.
        If the integer constraints of the DEM are satisfied, then $z^{DEM}$ found, STOP.
        Set $z^{DEM} := \infty$, $\bar{t} := 0$.
Step 1: (Root node)
        Solve the MIP submodels (3.1) to obtain $z^c \, \forall c \in \mathcal{C}$ and compute $z^0_{t^*} = \sum_{c \in \mathcal{C}} z^c$
        If $x^c$ variables do not satisfy NAC (3.3), then go to Step 2.
        If $y^c$ variables do not satisfy NAC (3.4), then $i := 0$ and go to Step 7.
        The optimal solution of the DEM (3.5) is found, then $z^{DEM} := z^0_{t^*}$, STOP.
Step 2: (Next stage)
        Reset $\bar{t} := \bar{t} + 1$ and $i$:= 0. If $\bar{t} > t^*$, then go to Step 9.
Step 3: (Next scenario group)
        Select $g \in \mathcal{G}_{\bar{t}}$. If all of its groups have been branched, then go to Step 2.
        Reset i:=0.
Step 4: (Next node)
        Reset $i := i + 1$. If $i > nx_{\bar{t}}$, then go to Step 3.
Step 5: (Branching)
        Set $x^c_{\bar{t}i} := 0 \, \forall c \in \mathcal{C}^g$.
Step 6: (Candidate TNF)
        Solve the MIP submodels (3.1) to obtain $z^c \, \forall c \in \mathcal{C}^g$. Compute $z_{t^*} = \sum_{c \in \mathcal{C}^g} z^c$.
        If $z_{t^*} \geq z^{DEM}$, then go to Step 8.
        If any $\mathbf{x}^c_t$ for $\forall t \in \mathcal{T}_1$ does not satisfy NAC (3.3) $\forall c \in \mathcal{C}$, then go to Step 4.
        If all $\mathbf{y}^c_t$ satisfy NAC (3.4) $\forall t \in \mathcal{T}_1, c \in \mathcal{C}$, then $z^{DEM} := z_{t^*}$. Go to Step 8.
Step 7: (Integer TNF)
        Solve LP model (3.6).
        If it is feasible, update $z^{DEM} := \min\{z^{TNF}_{LP}, z^{DEM}\}$.
        Solve in parallel the submodels (3.8) to obtain $z^{TNF}_{fc} \, \forall c \in \mathcal{C}^g$.
        If it is feasible, then $z^{TNF}_f = \sum_{c \in \mathcal{C}^g} z^{TNF}_{fc}$ and $z^{DEM} := \min\{z^{TNF}_f, z^{DEM}\}$.
        Solve MIP model (3.10) to obtain $z^{TNF}_{MIP}$.
        If it is feasible, update $z^{DEM} := \min\{z^{TNF}_{MIP}, z^{DEM}\}$.
Step 8: (Branch pruning)
        If $x^c_{\bar{t}i}$ has been branched to 0 for any $c \in \mathcal{C}^g$, then go to Step 11.
Step 9: (Backward to previous node)
        Reset $i := i - 1$.
        If $i = 0$ and $\bar{t} \leq 1$, then the solution value $z^{DEM}$ has been found, STOP.
        If $i = 0$ and all groups in $\mathcal{G}_{\bar{t}}$ have already been branched then $\bar{t} := \bar{t} - 1$ and
            select the last group $g \in \mathcal{G}_{\bar{t}}$.
        else if not all nodes have been branch, then select the previous group $g \in \mathcal{G}_{\bar{t}}$.
Step 10: (Prune checking)
        If $x^c_{\bar{t}i} = 1$ for any $c \in \mathcal{C}^g$, then go to Step 9.
Step 11: (Opposite branching)
        Reset $x^c_{\bar{t}i} := 1 \, \forall c \in \mathcal{C}^g$ and go to Step 6.
```

3.4 Inner parallelization of the BFC algorithm

In this section the main ideas of the parallelization of the submodels (3.1) in Step 1 and Step 6 and the submodels (3.8) in Step 7 are presented. If a set of threads is considered in a computing node, the execution of the $|C|$ scenario cluster submodels (3.1) can be parallelized by using different strategies depending on the number of task threads to be used in distributed memory (using MPI) and the number of auxiliary threads to be used in shared memory (by the MIP solver of choice).

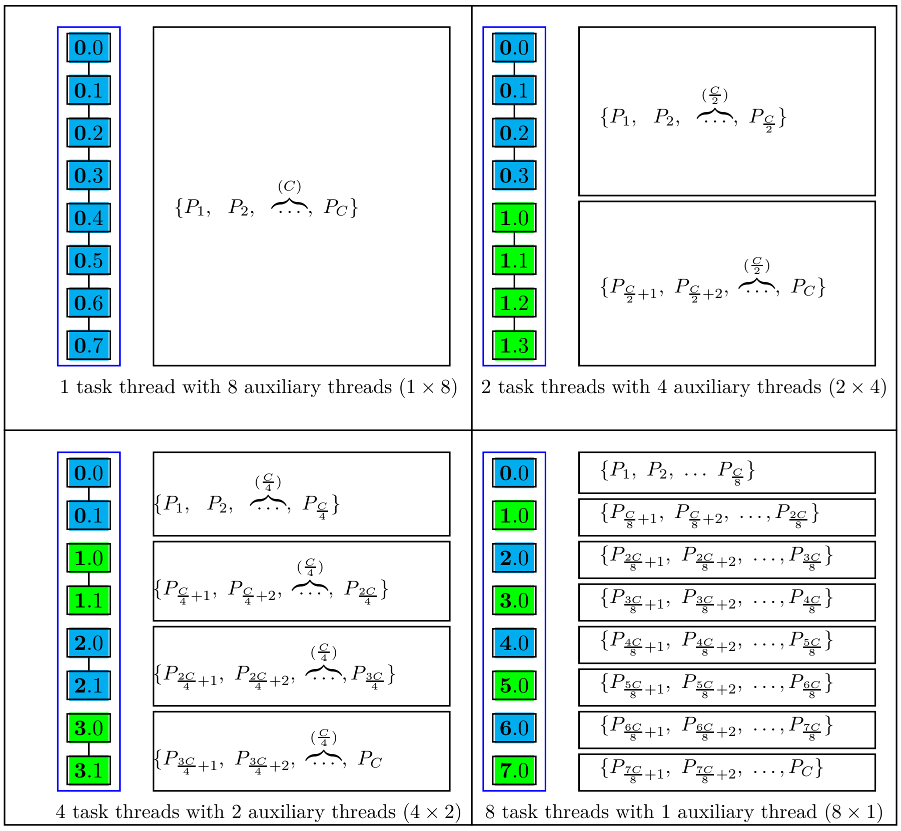
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
$\{P_1, \ P_2, \ \overset{(C)}{\frown\cdots}, \ P_C\}$
1 task thread with 8 auxiliary threads (1 × 8)
0.0
0.1
0.2
0.3
1.0
1.1
1.2
1.3
$\{P_1, \ P_2, \ \overset{(\frac{C}{2})}{\frown\cdots}, \ P_{\frac{C}{2}}\}$
$\{P_{\frac{C}{2}+1}, \ P_{\frac{C}{2}+2}, \ \overset{(\frac{C}{2})}{\frown\cdots}, \ P_C\}$
2 task threads with 4 auxiliary threads (2 × 4)
0.0
0.1
1.0
1.1
2.0
2.1
3.0
3.1
$\{P_1, \ P_2, \ \overset{(\frac{C}{4})}{\frown\cdots}, \ P_{\frac{C}{4}}\}$
$\{P_{\frac{C}{4}+1}, \ P_{\frac{C}{4}+2}, \ \overset{(\frac{C}{4})}{\frown\cdots}, \ P_{\frac{2C}{4}}\}$
$\{P_{\frac{2C}{4}+1}, \ P_{\frac{2C}{4}+2}, \ \overset{(\frac{C}{4})}{\frown\cdots}, P_{\frac{3C}{4}}\}$
$\{P_{\frac{3C}{4}+1}, \ P_{\frac{3C}{4}+2}, \ \overset{(\frac{C}{4})}{\frown\cdots}, \ P_C$
4 task threads with 2 auxiliary threads (4 × 2)
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
$\{P_1, \ P_2, \ \ldots \ P_{\frac{C}{8}}\}$
$\{P_{\frac{C}{8}+1}, \ P_{\frac{C}{8}+2}, \ \ldots, P_{\frac{2C}{8}}\}$
$\{P_{\frac{2C}{8}+1}, \ P_{\frac{2C}{8}+2}, \ \ldots, P_{\frac{3C}{8}}\}$
$\{P_{\frac{3C}{8}+1}, \ P_{\frac{3C}{8}+2}, \ \ldots, P_{\frac{4C}{8}}\}$
$\{P_{\frac{4C}{8}+1}, \ P_{\frac{4C}{8}+2}, \ \ldots, P_{\frac{5C}{8}}\}$
$\{P_{\frac{5C}{8}+1}, \ P_{\frac{5C}{8}+2}, \ \ldots, P_{\frac{6C}{8}}\}$
$\{P_{\frac{6C}{8}+1}, \ P_{\frac{6C}{8}+2}, \ \ldots, P_{\frac{7C}{8}}\}$
$\{P_{\frac{7C}{8}+1}, \ P_{\frac{7C}{8}+2}, \ \ldots, P_C\}$
8 task threads with 1 auxiliary thread (8 × 1)

Figure 3.2: Parallel computing strategies: task threads with auxiliary threads

As an example, assume that 8 threads are available for parallelizing the optimization of the $|C|$ independent submodels. Some strategy options are as follows: (1) 8 task threads are used for solving the submodels (one per each thread) and running the MIP solver in the same task thread (i.e., the MIP solver is not allowed to use internal functions in parallel); (2) 4 task threads are used with 2 auxiliary threads each; (3) 2 task threads and 4 auxiliary threads and (4) 1 task thread and 8 auxiliary threads, see Figure 3.2.

Additionally, let main thread be the thread where the non-parallelized calculations are executed; so, it is the one that gathers the distributed information. It can be denoted as thread **0**, and correlatively the other task threads can be denoted as **1**, **2**, ... Similarly, the auxiliary threads for each primary one can be denoted as 0, 1 , 2, ..., such that **i**.j denotes the jth auxiliary thread for the **i**th task one. Notice that each auxiliary thread is linked to a specific task thread.

The general scheme of the inner Parallel Branch-and Fix-Coordination (Inner P-BFC) is presented in Algorithm 3.2 and illustrated in Figure 3.3 by using an illustrative platform where there are 2 task threads and 4 auxiliary threads each.

First, the optimization variables related to the DEM (3.5) problem are declared, as introduced in Escudero *et al.* [2012a]. Then the MPI variables are declared by the task threads, see Pacheco [1996]; Snir *et al.* [1995] for the most frequent MPI variables and Section A.1 in Appendix A.

Then, in order to have a point-to-point communication environment it is essential to identify each task thread with a *thread rank* number, so that the sender and the receiver can be easily specified. Additionally, the rank enables a Single Program Multiple Data (SPMD) paradigm to be worked on, which implies that different threads execute different tasks in the same program. Moreover, the tasks to be executed are divided between thread groups.

When independent cluster MIP submodels (3.1) are solved, each task thread starts a shared memory parallel programming environment with a chosen number of auxiliary threads. Once finished the main thread gathers the solution and the solution value of all submodels by using message-passing communication. In order for the message to be successfully communicated, see Pacheco [1996], the system must append some information to the data that the application program wishes to transmit. The MPI library provides a rich variety of functions for message-passing tasks. For further information see Snir *et al.* [1995] and Appendix A.

Finally, when the optimal solution is found the results are reported and the MPI environment finishes; execution ends in all task threads.

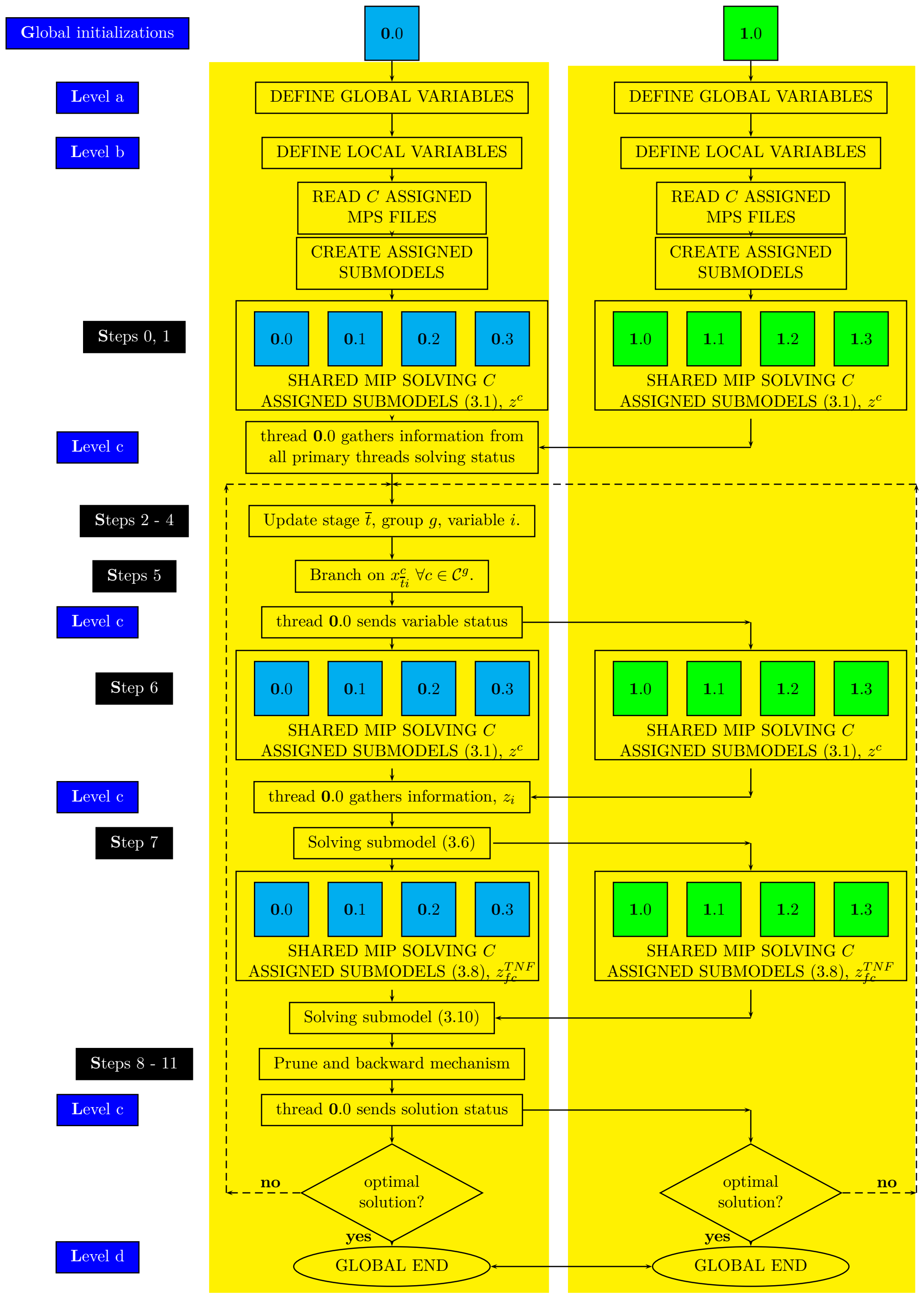
Global initializations
0.0
1.0
Level a
DEFINE GLOBAL VARIABLES
Level b
DEFINE LOCAL VARIABLES
READ C ASSIGNED MPS FILES
CREATE ASSIGNED SUBMODELS
Steps 0, 1
0.0 0.1 0.2 0.3
1.0 1.1 1.2 1.3
SHARED MIP SOLVING C ASSIGNED SUBMODELS (3.1), z^c
Level c
thread **0.0** gathers information from all primary threads solving status
Steps 2 - 4
Update stage $\bar{t}$, group g, variable i.
Steps 5
Branch on $x^c_{\bar{t}i}$ $\forall c \in \mathcal{C}^g$.
Level c
thread **0.0** sends variable status
Step 6
Level c
thread **0.0** gathers information, z_i
Step 7
Solving submodel (3.6)
SHARED MIP SOLVING C ASSIGNED SUBMODELS (3.8), z^{TNF}_{fc}
Solving submodel (3.10)
Steps 8 - 11
Prune and backward mechanism
Level c
thread **0.0** sends solution status
optimal solution?
no
yes
Level d
GLOBAL END

Figure 3.3: Illustrative one-path (1 × 2 × 4) Inner P-BFC diagram

Algorithm 3.2: Inner P-BFC

```
Global initialization:  [All task and auxiliary threads]

Level a: Declaring optimization and MPI variables  [All task threads]
        The optimization variables are declared, see details in Section A.1
        in Apendix A

Level b: Definition of the global environment.  [All primary threads]
        Identify the number of task threads and assign a thread rank number
        to each one.
Step 0: Solve the LP relaxation of DEM (3.5).  [Main thread]

Step 1: Solve the C submodels (3.1).  [All task and auxiliary threads]
        Each task thread solves the associated optimization models with a
        chosen number of auxiliary threads.

Level c: Global MPI communication.  [All task threads]
        If the solution value of the original DEM (3.5), computed as
        $z^{DEM} = \sum_{c \in \mathcal{C}} z^c$, satisfies the NAC (3.3) and (3.4), the optimal solution
        is found and, then, go to Level d.

Steps 2 to 5: Next stage $\bar{t}$, next scenario group $g \in \mathcal{G}_{\bar{t}}$, next node i
        and branching.  [Main thread]
        Follow the instructions given in Algorithm 3.1 to branch the next variable.

Step 6: Solve assigned C submodels (3.1).  [All task and auxiliary threads]
        Each auxiliary thread, with a chosen number of auxiliary threads,
        solve the $|\mathcal{C}^g|$ independent cluster MIP submodels (3.1) with the appropriate
        $x$- fixations in the branching phases of the coordinated cluster trees.

Level c: Global MPI communication.  [All task threads]
        The main thread gathers the solution and the solution value
        of all submodels, such that $z = \sum_{c \in \mathcal{C}^g} z^c$ is computed.

Steps 7: Solve TNF models.  [Main thread]
        The LP submodel (3.6), the parallelized MIP submodels (3.8) $\forall c \in \mathcal{C}$
        and the MIP submodel (3.10) are solved. If the optimal solution is found
        Level c: Global MPI communication.  [All task threads]
            Report results and go to Level d.

Steps 9 to 11: Pruning and Backward mechanism.  [Main thread]
        Follow Algorithm 3.1 to prune and backward branching.

Level d: Execution ends in all threads.  [All task threads]
```

3.5 Outer parallelization of the BFC algorithm

The Outer P-BFC is an algorithm whose efficiency is based on parallel interconnected executions of the BFC algorithm. The outer level of parallelization is managed by the so-called paths. At the beginning, the procedure defines a set of k 0-1 variables, $(x_1, x_2, \ldots, x_k)$, the combinations of whose 0-1 values allows to initiate 2^k paths. Each path is implemented by a main thread and associated to a dynamically reassigned subproblem of the unvisited Branch-and-Fix tree defined by a path Branch-and-Fix tree, $\mathcal{BFT}_{path}$, and its corresponding root node $\overline{\mathcal{N}}_{path}= (\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_k)$, the BFC algorithm, a set of task and auxiliary threads and a MPI communication environment with other main threads.

According with the thread assignation given in Section 1.3, an Outer P-BFC execution is denoted as $(a \times 1 \times h)$. When 32 threads are available, for instance the $(4 \times 1 \times 8)$ thread assignation means that 4 main threads are defined (they are associated to a four-path outer scheme), a single task threads will solve clusters in parallel (the main thread itself), by calling 8 CPLEX optimizer auxiliary threads.

A synchronization phase based on communication increases global efficiency in two ways: main threads associated to paths exchange feasible solution values, allowing an earlier prunning of branches; and the main threads associated to paths with no more Branch-and-Fix nodes to visit reassign the path subproblem by redefining the root node, $\overline{\mathcal{N}}_{path}$, and the path Branch-and-Fix tree, $\mathcal{BFT}_{path}$, obtained by splitting the tree of an unfinished $\mathcal{BFT}$ part. The parallelization of the set of paths reduces the number of Branch-and-Fix nodes to be visited and therefore the execution time.

The Outer algorithm is illustrated in the Figure 3.4 for four paths. Therefore, it starts by fixing two 0-1 variables, (x_1, x_2) to $\overline{\mathcal{N}}_1 = (0, 0)$, $\overline{\mathcal{N}}_2 = (0, 1)$, $\overline{\mathcal{N}}_3 = (1, 0)$ and $\overline{\mathcal{N}}_4 = (1, 1)$, respectively, such that these combinations define the initializations for 2^2 paths in simultaneous and interconnected executions, so, the algorithm starts in 4 main threads simultaneously. When each path has obtained a new feasible solution such that its values are not smaller than the global incumbent solution z^{DEM} or the $\mathcal{BFT}_{path}$ has been fully visited, then the synchronization phase begins.

The independent execution part corresponds to Steps 1 to Step 11 in Algorithm 3.3 when synchronization phase is not called and is illustrated with yellow boxes in Figure 3.4. Each path follows the serial BFC algorithm steps described previously in Algorithm 3.1 with two variations.

First, the initialization, step where the algorithm starts with a root node $\overline{\mathcal{N}}_{path} = (\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_k)$ with k 0-1 variables fixed and its associated Branch-and-

Algorithm 3.3: Outer P-BFC

```
Level a:  (Declaring optimization and MPI variables) [All main threads].
          GLOBAL START.
Level b:  (Definition of the global environment) [All main threads].
Step 0:   (Initializations)
Step 1:   (Root node)
Step 2:   (Next stage)
Step 3:   (Next scenario group)
Step 4:   (Next node)
Step 5:   (Branching)
Step 6:   (Candidate TNF)
          If a feasible solution of DEM is obtained, then update $z_{path}^{DEM}$.
          Go to Synchronization phase.
Step 7:   (Integer TNF)
          At the end of Step 7, update $z_{path}^{DEM}$, if all nodes have been visited
          at the $\mathcal{BFT}_{path}$, then set $dead_{path} = 1$ (dead), else set $dead_{path} = 0$ (active).
          Go to Synchronization phase.
Step 8:   (Branch pruning)
Step 9:   (Backward to previous node)
          If all the nodes have been visited at the $\mathcal{BFT}_{path}$, then set $dead_{path} = 1$ (dead).
          Go to Synchronization phase.
Step 10:  (Prune checking)
Step 11:  (Opposite branching)
```

```
Synchronization phase
          Level c:  (All paths gather $z_{path}^{DEM}$ and $dead_{path}$) [All main threads].
          Set $z^{DEM} = min(z^{DEM}, min_{path}\{z_{path}^{DEM}\})$.
          If all $dead_{path} = 1$ (dead) then
              Level d:  (Finish MPI environmment) [All main threads].
              GLOBAL END.
          else
              Level c:  (Variable branching exchange) [All main threads].
              Dead paths are reassigned by splitting active path $\mathcal{BFT}_{path}$.
              All paths update root node $\overline{\mathcal{N}}_{path}$.
          Paths where $dead_{path} = 0$ continue branching, go to Step 8.
          Paths where $dead_{path} = 1$ restart algorithm, go to Step 1.
```

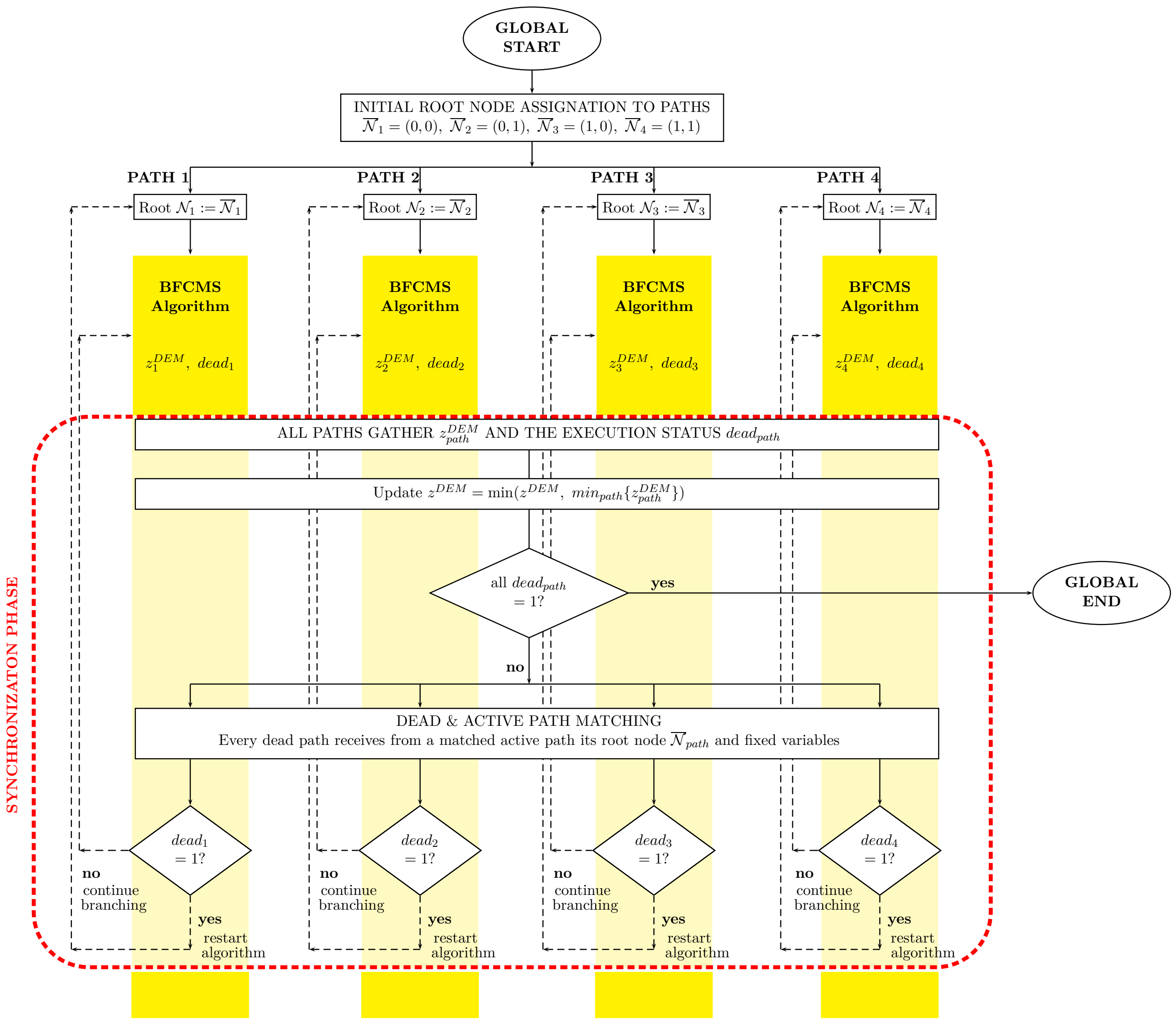
GLOBAL START
INITIAL ROOT NODE ASSIGNATION TO PATHS
$\overline{\mathcal{N}}_1 = (0,0)$, $\overline{\mathcal{N}}_2 = (0,1)$, $\overline{\mathcal{N}}_3 = (1,0)$, $\overline{\mathcal{N}}_4 = (1,1)$
PATH 1
PATH 2
PATH 3
PATH 4
Root $\mathcal{N}_1 := \overline{\mathcal{N}}_1$
Root $\mathcal{N}_2 := \overline{\mathcal{N}}_2$
Root $\mathcal{N}_3 := \overline{\mathcal{N}}_3$
Root $\mathcal{N}_4 := \overline{\mathcal{N}}_4$
BFCMS Algorithm
z_1^{DEM}, $dead_1$
z_2^{DEM}, $dead_2$
z_3^{DEM}, $dead_3$
z_4^{DEM}, $dead_4$
SYNCHRONIZATON PHASE
ALL PATHS GATHER z_{path}^{DEM} AND THE EXECUTION STATUS $dead_{path}$
Update $z^{DEM} = \min(z^{DEM},\ min_{path}\{z_{path}^{DEM}\})$
all $dead_{path}$ = 1?
yes
GLOBAL END
no
DEAD & ACTIVE PATH MATCHING
Every dead path receives from a matched active path its root node $\overline{\mathcal{N}}_{path}$ and fixed variables
$dead_1$ = 1?
$dead_2$ = 1?
$dead_3$ = 1?
$dead_4$ = 1?
no continue branching
yes restart algorithm

Figure 3.4: Illustrative four-path (4 × 1 × 1) Outer P-BFC diagram

Fix tree $\mathcal{BFT}_{path}$ for each path. And second, the path execution moves to an interconnected synchronization phase when its branching situation can benefit other paths, that is, whenever a new feasible solution has been found whose z_{path}^{DEM} (in Step 6 or Step 7) is smaller than the global incumbent value z^{DEM}, or all the nodes of the $\mathcal{BFT}_{path}$ have already been visited (checked in Step 9).

For the description of the synchronization phase, let us define the active path and the dead path, where initially all paths are free (non-active and non-dead):

- An **active** path, the BFC algorithm for the path has still nodes of the $\mathcal{BFT}_{path}$ that have not yet been visited. Set $dead_{path} = 0$.
- A **dead** path, all nodes of the $\mathcal{BFT}_{path}$ have been visited or have been pruned. Set $dead_{path} = 1$.

The interconnected synchronization phase, the bottom block of Algorithm 3.3 and illustrated with a dashed red box in Figure 3.4, starts; first, with all paths gathering the last feasible solution obtained by each path, z_{path}^{DEM}, and updating the incumbent global value, z^{DEM}, of the Deterministic Equivalent Model (DEM). Second, dead/active path analysis starts and one of the following three situations can happen:

- All paths are dead. So, all nodes of the $\mathcal{BFT}_{path}$ have been visited and, therefore, the original Branch-and-Fix tree $\mathcal{BFT}$ have been fully visited and, then, the execution ends in all paths, so, the Outer P-BFC algorithm stops, and the optimal solution has been found (global end).
- All paths are active. So, each one will continue branching and fixing in its own $\mathcal{BFT}_{path}$, with the new z^{DEM} updated.
- Otherwise, each dead path will match with an active path, let us denote $path_1$ and $path_2$, respectively. The next fixed variable x_{k+1} in the $\mathcal{BFT}_{path_2}$ will be descended with root node $\overline{\mathcal{N}}_{path_2} = (\overline{x}_1, \ldots, \overline{x}_k)$, the initial root node for $path_2$ is updated in its $(k+1)$th index, for example, to $\overline{\mathcal{N}}_{path_2} = (\overline{x}_1, \ldots, \overline{x}_k, 0)$ and the dead path $path_1$ will restart the BFC algorithm with the initial root node $\overline{\mathcal{N}}_{path_1} = (\overline{x}_1, \ldots, \overline{x}_k, 1)$ and its associated new $\mathcal{BFT}_{path_1}$, while the active path $path_2$ will continue branching in its updated $\mathcal{BFT}_{path_2}$.

Figure 3.5 and Table 3.1 show an illustrative example of the Outer P-BFC solving instance P3 using two paths, where nodes of Path 1 and Path 2 are in blue and green, respectively. When two paths are available only one 0-1 variable can be fixed for defining the path, let it x_1; so, Path 1 starts the BFC algorithm with root node $\overline{\mathcal{N}}_1 : (x_1) = (0)$ and Path 2 with $\overline{\mathcal{N}}_2 : (x_1) = (1)$. The synchronization phase is reached four times during the global solving, the process is as follows.

The first synchronization phase, denoted *Sync.*1 in Figure 3.5, takes place when Path 1 obtains a feasible solution (for the original problem, that is, all x-variables

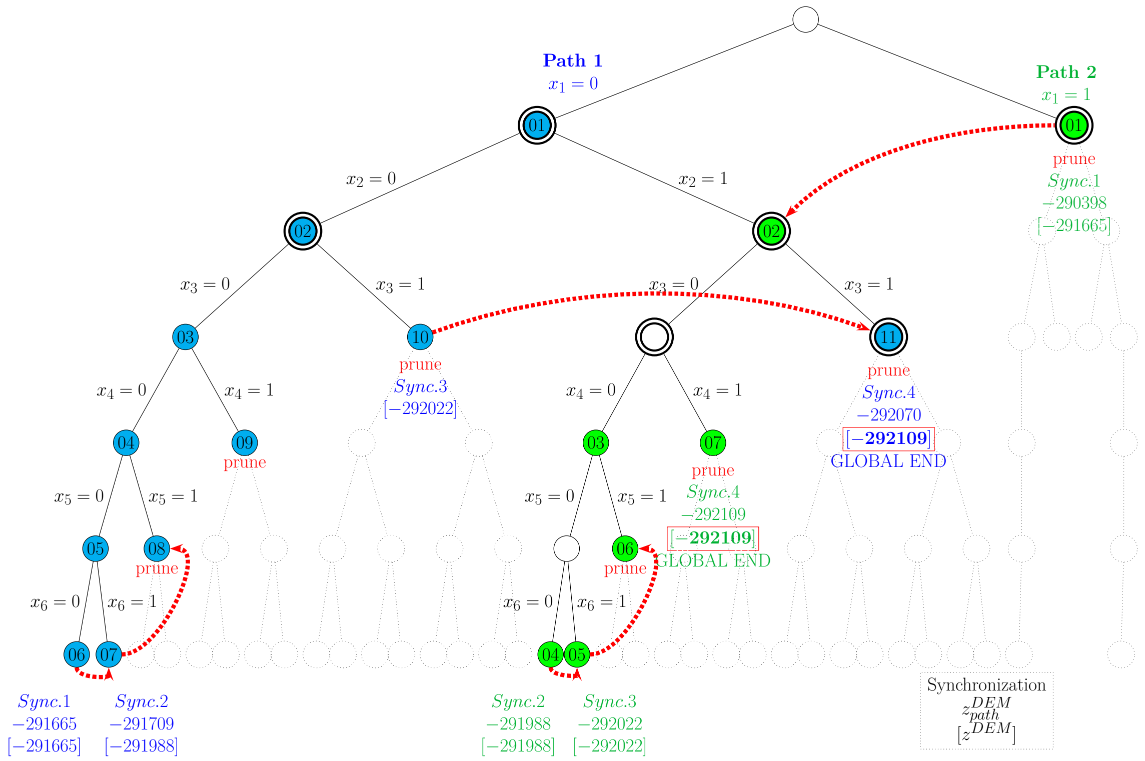

Figure 3.5: Two-path Outer P-BFC performance for instance P3

take 0-1 values), at node 06, where $z_1^{DEM} = -291665$, and as it is still active, so $dead_1 := 0$; while Path 2 obtains a feasible solution at node 01, $z_2^{DEM} = -290398$, and therefore the branch is pruned and, then, $\mathcal{BFT}_2$ is fully visited, so, $dead_2 := 1$. After the path solutions comparison, as $z_1^{DEM} < z_2^{DEM}$ and $z_1^{DEM} < z^{DEM}$, then $z^{DEM} := z_1^{DEM}$. As Path 2 is dead, let us descend to fix one binary variable, i.e., x_2 by sharing Path 1 Branch-and-Fix tree. The root node for Path 1 is updated to $\overline{\mathcal{N}}_1 : \ (x_1, x_2) = (0, 0)$ and will continue branching; while Path 2 will give up the previous tree and be engaged to the root node $\overline{\mathcal{N}}_2 : \ (x_1, x_2) = (0, 1)$ and will restart the BFC to solve the new path Branch-and-Fix tree $\mathcal{BFT}_2$. Note that root nodes are indicated with doublelined circles in Figure 3.5.

The second synchronization phase, denoted *Sync.*2, takes place when Path 1 obtains a feasible solution in node 07 (whose solution value is smaller than the global incumbent value z^{DEM}) being $z_1^{DEM} = -291709$ and it is still active. On the other hand, Path 2 branches in nodes 02 to 04, such that a feasible solution is obtained, being $z_2^{DEM} = -291988$ (smaller than the global incumbent value) and it is still

Path 1			**Path 2**		
Root/Node	z_1^c	z_1^{DEM}	Root/Node	z_2^c	z_2^{DEM}
Root (0)			Root (1)		
Node 01	-292117		Node 01	-290404	-290398
Node 02	-291839				
Node 03	-291826				
Node 04	-291759				
Node 05	-291741				
Node 06	-291678	-291665			
Step 7 solved, $dead_1 = 0$			Step 7 solved and all nodes branched, $dead_2 = 1$		
Synchronization phase 1: $\mathbf{z^{DEM}} = -\mathbf{291665}$					
goto Step 8 (continue branching)			goto Step 1 (restart algorithm)		
Root $(0,0)$			Root $(0,1)$		
Node 07	-291718	-291709	Node 02	-292116	
			Node 03	-292061	
			Node 04	-291996	-291988
Step 7 solved, $dead_1 = 0$			Step 7 solved, $dead_2 = 0$		
Synchronization phase 2: $\mathbf{z^{DEM}} = -\mathbf{291988}$					
goto Step 8 (continue branching)			goto Step 8 (continue branching)		
Root $(0,0)$			Root $(0,1)$		
Node 08	-291652		Node 05	-292027	-292022
Node 09	-291826				
Node 10	-291803	∞			
all nodes branched, $dead_1 = 1$			Step 7 solved, $dead_2 = 0$		
Synchronization phase 3: $\mathbf{z^{DEM}} = -\mathbf{292022}$					
goto Step 1 (restart algorithm)			goto Step 8 (continue branching)		
Root $(0,1,1)$			Root $(0,1,0)$		
Node 11	-292075	-292070	Node 06	-291733	
			Node 07	-292114	-292109
Step 7 solved and all nodes branched, $dead_1 = 1$			Step 7 solved and all nodes branched, $dead_2 = 1$		
Synchronization phase 4: $\mathbf{z^{DEM}} = -\mathbf{292109}$					
GLOBAL END					

Table 3.1: Two-path Outer P-BFC performance for instance P3

active. After the path solutions comparison, as $z_1^{DEM} > z_2^{DEM}$ and $z_2^{DEM} < z^{DEM}$, then $z^{DEM} := z_2^{DEM}$, nodes 07 and 04 are pruned in Path 1 and Path 2, respectively. As $dead_1 := 0$ and $dead_2 := 0$, root nodes do not need to be updated and both paths continue branching.

The third synchronization phase, denoted $Sync.3$ takes place when Path 1 branches on nodes 08 to 10 and stops because it has finished branching its own tree rooted at $(0, 0)$. On the other hand, Path 2 branches on node 05 where a feasible solution is obtained, being $z_2^{DEM} = -292022$ (smaller than the global incumbent value). After the path solutions comparison, as $z_1^{DEM} > z_2^{DEM}$ and $z_2^{DEM} < z^{DEM}$, then $z^{DEM} := z_2^{DEM}$. Node 10 of Path 1 is pruned and the path is dead, $dead_1 := 1$, since Path 2 is still active, $dead_2 := 0$. Path 1 has finished its own tree, therefore, let us descend to fix one binary variable, i.e., x_3. Path 1 will give up the previous tree and be engaged to root node $\overline{\mathcal{N}}_1: (x_1, x_2, x_3) = (0, 1, 1)$ and the root node for Path 2 is updated to $\overline{\mathcal{N}}_2: (x_1, x_2, x_3) = (0, 1, 0)$.

Finally, the last synchronization phase, denoted *Sync.*4 takes place when Path 1 branches on node 11, a feasible solution is obtained, being $z_1^{DEM} = -292070$ (smaller than the global incumbent solution). On the other hand, Path 2 branches on nodes 06 and 07, so that Step 7 is executed obtaining a feasible solution, being $z_2^{DEM} = -292109$ (smaller than the global incumbent value). After the path solutions comparison, as $z_1^{DEM} > z_2^{DEM}$ and $z_2^{DEM} < z^{DEM}$, then $z^{DEM} := z_2^{DEM}$. Node 11 of Path 1 is pruned. And additionally, both paths are dead, $dead_1 := 1$ and $dead_2 := 1$, since all nodes at the path trees $\mathcal{BFT}_1$ and $\mathcal{BFT}_2$ have been branched. Therefore, Outer P-BFC algorithm has finished, and the incumbent solution found is $z^{DEM} = -292109$.

3.6 Outer-Inner parallelization of the BFC algorithm

Observe that the independent BFC execution of each path in Algorithm 3.3 can be internally optimized by performing the Inner P-BFC described in Algorithm 3.2. The hybrid Outer-Inner BFC allows to use the computational resources in the approach where the marginal effect on the elapsed time is bigger. The inner approach should be reinforced when the scenario cluster submodels (3.1) represent a significant part of the total elapsed time and/or the number of binary variables to be branched is small. On the other hand, the outer approach should be reinforced when the integer TNF model solving, (3.6), (3.8) and (3.10) represent a significant part of the total elapsed time and/or the number of binary variables to be branched is large.

According to the thread assignation given in Section 1.3, a joint Outer-Inner P-BFC execution is denoted as $(a \times b \times h)$. When 64 threads are available, for instance the $(2 \times 4 \times 8)$ thread assignation means that 2 main threads are defined (they are associated to a two-path outer scheme), 4 task threads will solve clusters in parallel (they are used in inner scheme inside each path), by calling 8 CPLEX optimizer auxiliary threads.

The procedure of the hybrid Outer-Inner BFC is described in Algorithm 3.4. The main structure corresponds to Algorithm 3.3, where inner parallelization of Algorithm 3.2 has been added. Therefore, after solving the scenario cluster submodels (3.1) in Step 1 and Step 6, a secondary MPI communication is performed within the task group, so that each main thread gathers the solution and solution values of the associated task threads. Notice that the declaring of MPI variables and the finish of the MPI environment is performed by all task threads.

Algorithm 3.4: Outer-Inner P-BFC

```
Level a:  (Declaring optimization and MPI variables) [All task threads].
          GLOBAL START.
Level b:  (Definition of the global environment) [All main threads].
Step 0:  (Initializations)
Step 1:  (Root node)
          Level c: Secondary MPI communication.  [All task threads]
          The main thread gathers the solution and the solution value
          of all submodels.
Step 2:  (Next stage)
Step 3:  (Next scenario group)
Step 4:  (Next node)
Step 5:  (Branching)
Step 6:  (Candidate TNF)
          Level c: Secondary MPI communication.  [All task threads]
          The main thread gathers the solution and the solution value
          of all submodels.
          If a feasible solution of DEM is obtained, then update $z_{path}^{DEM}$.
          Go to Synchronization phase.
Step 7:  (Integer TNF)
          At the end of Step 7, update $z_{path}^{DEM}$, if all nodes have been visited
          at the $\mathcal{BFT}_{path}$, then set $dead_{path} = 1$ (dead), else set $dead_{path} = 0$ (active).
          Go to Synchronization phase.
Step 8:  (Branch pruning)
Step 9:  (Backward to previous node)
          If all the nodes have been visited at the $\mathcal{BFT}_{path}$, then set $dead_{path} = 1$ (dead).
          Go to Synchronization phase.
Step 10:  (Prune checking)
Step 11:  (Opposite branching)

Synchronization phase
          Level c:  (All paths gather $z_{path}^{DEM}$ and $dead_{path}$) [All main threads].
          Set $z^{DEM} = min(z^{DEM}, min_{path}\{z_{path}^{DEM}\})$.
          If all $dead_{path} = 1$ (dead) then
              Level d:  (Finish MPI environmment) [All task threads].
              GLOBAL END.
          else
              Level c:  (Variable branching exchange) [All main threads].
              Dead paths are reassigned by splitting active path $\mathcal{BFT}_{path}$.
              All paths update root node $\overline{\mathcal{N}}_{path}$.
          Paths where $dead_{path} = 0$ continue branching, go to Step 8.
          Paths where $dead_{path} = 1$ restart algorithm, go to Step 1.
```

3.7 Computational experience

The computational experiments were conducted in the ARINA computational cluster at SGI/IZO-SGIker from UPV/EHU (see Section 1.5). For this computational experiment, 16 Intel Xeon type computing nodes have been used, consisting of 8 cores with 48Gb of RAM. The P-BFC algorithm has been implemented in a `C++` experimental code which uses the state-of-the-art optimization LP/MIP solver CPLEX V12.2 (called from COIN-OR V1.3.1). The optimizer is used by the algorithm to solve the LP relaxation of the original DEM (3.5), the MIP submodels (3.1) for the set of scenario clusters $\mathcal{C}$ in different steps, LP submodel (3.6), the MIP submodels (3.8) for the set of scenario clusters $\mathcal{C}$ and MIP model (3.10).

The computational experience is reported as follows: Section 3.7.1 presents the dimensions of the testbed we have experimented with. Section 3.7.2 gives the solution value of the original DEM (3.5) and strong lower bounds via scenario clustering. Section 3.7.3 introduces the analysis of the parallel solving of the testbed root nodes. Sections 3.7.4 and 3.7.5 report the results of the comparison performed with different inner and outer P-BFC strategies versus plain use of CPLEX, respectively. Finally, Section 3.7.6 reports the results of the comparison performed with different Outer-Inner P-BFC strategies versus plain use of CPLEX.

3.7.1 Testbed dimensions

The instances P1-P14, denoted Testbed 2, of the computational experimentation are taken from Escudero *et al.* [2012a]. The dimensions of the original DEM (3.5) and the set Q of numbers $\mathcal{C}$ of scenario cluster submodels (3.1) for each potential break stage $t^* = 0, 1, \ldots, T-1$ are given in Table 3.2. The other headings of the table are as follows: m, number of constraints; nx, number of 0-1 variables; ny, number of continuous variables; nel, number of nonzero coefficients in the constraint matrix; $dens$, constraint matrix density (in %); $|\Omega|$, number of scenarios; $|\mathcal{G}|$, number of scenario groups; and T, number of stages.

3.7.2 Scenario clustering solution value and related GAPs

Table 3.3 reports the performance of the break stages $t^* = 1$ and $t^* = 2$ for obtaining strong lower bounds of the solution value of the original DEM (3.5), by comparing the LP optimality gap $GAP_{LP}{=}\ \frac{z^{DEM} - z_{LP}}{z_{LP}}(\%)$ and the gap based on the break stages defined as $GAP^0_{t^*} = \frac{z^{DEM} - z^0_{t^*}}{z^0_{t^*}}(\%)$. The other headings are as follows: z^{DEM} and z_{LP}, solution values of the original DEM and its LP relaxation, respectively; C, number of clusters that have been considered; $z^0_{t^*} = \sum\limits_{c \in \mathcal{C}} z^c$, where z^c is the solution value of the MIP model (3.1) for scenario cluster c.

Table 3.2: Dimensions of DEM, compact representation. Testbed 2.

Instance	m	nx	ny	nel	$dens$	$\|\Omega\|$	$\|\mathcal{G}\|$	T	Q
P1	2114	453	1359	44040	1.15	113	151	4	$\{1,8,29,113\}$
P2	2544	530	1590	88465	1.64	74	106	4	$\{1,7,24,74\}$
P3	7072	1632	4896	295577	0.64	217	272	4	$\{1,10,44,217\}$
P4	9248	2176	6528	515768	0.64	217	272	4	$\{1,10,44,217\}$
P5	2002	429	1287	41861	1.22	80	143	5	$\{1,6,16,40,80\}$
P6	12766	2946	8838	534563	0.35	340	491	5	$\{1,8,29,113,340\}$
P7	14400	3456	10368	1206875	0.60	182	288	5	$\{1,7,24,74,182\}$
P8	17380	3950	11850	606173	0.22	574	790	5	$\{1,9,39,167,574\}$
P9	24552	5580	16740	856121	0.16	844	1116	5	$\{1,10,44,217,844\}$
P10	2010	402	1206	28064	0.87	104	201	6	$\{1,5,13,26,52,104\}$
P11	2814	603	1809	59256	0.87	104	201	6	$\{1,5,13,26,52,104\}$
P12	4545	909	2727	93918	0.57	160	303	6	$\{1,6,16,40,80,160\}$
P13	11824	2217	6651	249021	0.24	451	739	6	$\{1,7,24,74,182,451\}$
P14	45216	8478	25434	951759	0.06	2036	2826	6	$\{1,9,39,167,574,2036\}$

Table 3.3: Stochastic solution values and GAPs. Testbed 2.

		DEM		$t^*=1$			$t^*=2$		
Instance	z_{LP}	z^{DEM}	GAP_{LP}	C	$z^0_{t^*}$	$GAP^0_{t^*}$	C	$z^0_{t^*}$	$GAP^0_{t^*}$
P1	-157109	-156324	0.5	8	-156324	0.0	29	-157060	0.5
P2	-6483.55	-6146.04	5.2	7	-6175.99	0.5	74	-6254.76	1.7
P3	-293677	-292109	0.5	10	-292118	0.0	44	-293086	0.3
P4	-285706	-283938	0.6	10	-284151	0.1	44	-285441	0.5
P5	-6309.89	-6067.51	3.8	6	-6067.51	0.0	16	-6166.96	1.6
P6	-36302.8	-35959.9	0.9	8	-35960.3	0.0	29	-36008.3	0.1
P7	-270222	-269441	0.3	7	-269447	0.0	24	-269880	0.2
P8	-155077	-154814	0.2	9	-154814	0.0	39	-154902	0.1
P9	-226377	-225754	0.3	10	-225754	0.0	44	-225863	0.0
P10	37343.6	38156.6	2.2	5	38156.6	0.0	13	38118.3	0.1
P11	39364	39805.7	1.1	5	39797.5	0.0	13	39773.9	0.1
P12	40892	41502.3	1.5	6	41475.5	0.1	16	41445.8	0.1
P13	40522.3	41337.4	2.0	7	41337	0.0	24	41302.6	0.1
P14	41254.1	41783.5	1.3	9	41744.6	0.1	39	41701.7	0.2

In general, both GAPs are small, but $GAP^0_{t^*}$ for break stage $t^* = 1$ is the smallest one for the testbed, as it is zero in almost all the instances. This does not necessarily mean that the NAC are satisfied by the solution whose value is $z^0_{t^*}$. The $GAP^0_{t^*}$ for break stage $t^* = 2$ is also very good for many instances in Testbed 2.

3.7.3 Preliminary analysis: Parallel solving of root node

Let us begin the parallel computation experience by analysing the elapsed time of solving the root node of the instances in Tested 2. The root node solution $z^0_{t^*}$ is obtained by solving the cluster subproblems related to the chosen break stage; as shown in Table 3.3, the smaller the break stage, t^*, the better bound. When we decompose the original problem with an early break stage, the number of cluster subproblems is low but their scale is large; alternatively a later break stage creates more, but smaller scaled, subproblems. In this section the effect of the break stage on the best parallel strategy (task threads and auxiliary threads) is analysed.

Table 3.4 for instances P1-P4, Table 3.5 for instances P5-P9 and Table 3.6 for instances P10-P14 show the elapsed time (in seconds) required for solving up to optimality the root node cluster suproblems for different combinations of break stages and the 16 (task and auxiliary) threads available in our computational experimentation, based on the fact that CPLEX cannot use more than 8 (auxiliary threads) in the academic version.

Let us describe how these C submodels can be solved in parallel in order to obtain the solution values (of the optimal solutions) of each cluster submodel c called $z^1_{t^*}, z^2_{t^*}, \dots, z^C_{t^*}$, and let us present the study of the real time vs ($task \times auxiliary$) threads strategies. Notice that a single main thread is considered.

Table 3.4 for P1-P4, Table 3.5 for P5-P9 and Table 3.6 for P10-P14 show in bold the ($task \times auxiliary$) threads combinations with the smallest elapsed time as well as the characteristics of the related parallel computing strategies that have been used for obtaining those elapsed times. The identifiers of each strategy for a given set of threads means are as follows: *Pure distributed* (PD), the greater number of task threads, the smaller elapsed time; *Sharing* (SH), the greater number of task threads, the larger elapsed time; *Quasidistributed* (QD) the greater number of task threads, the smaller elapsed time except for a single case; and, *Undefined* (UD) the number of task threads versus auxiliary threads does not have a clear impact on the greater/smaller elapsed time.

We can observe in Table (3.4) that the pure distributed strategy is the best one for the instances P1-P4. For $t^* = 1$ there is not a clear best ($task \times auxiliary$) combination, being all of them very good ones for P1-P3. It is worth to point out that for the biggest instance of this serie, i.e., P4, the best combination is (2×8) being pure shared the best strategy. On the other hand, the best combination is (16×1) pure distributed for $t^* = 2$ and $t^* = 3$.

We can observe in Tables (3.5) and Table (3.6) (larger scale instances) the same pattern observed in Tables (3.4), that is the combination (16×1) for

Table 3.4: Elapsed time for solving the root node of instances P1-P4

Case	task × auxiliary	$t^* = 1$	$t^* = 2$	$t^* = 3$
P1	2×8	0.549	1.225	3.556
	4×4	0.246	0.305	0.888
	8×2	**0.168**	0.160	0.381
	16×1	0.177	**0.109**	**0.185**
	best case	**QD**	**PD**	**PD**
P2	2×8	2.417	1.408	2.459
	4×4	**0.584**	0.375	0.743
	8×2	0.588	**0.204**	0.385
	16×1	0.832	0.210	**0.234**
	best case	**UD**	**QD**	**PD**
P3	2×8	7.928	3.258	12.0678
	4×4	**7.916**	1.099	3.487
	8×2	10.676	0.662	1.404
	16×1	10.883	**0.444**	**0.789**
	best case	**UD**	**PD**	**PD**
P4	2×8	**117.773**	4.370	13.891
	4×4	185.580	1.826	4.657
	8×2	256.956	1.129	2.227
	16×1	594.424	**0.943**	**1.340**
	best case	**SH**	**PD**	**PD**

(*task* × *auxiliary*) is almost always the best one; being the pure distributed the best strategy for $t^* > 1$. On the contrary, the best (*task* × *auxiliary*) combination as well as the best strategy for $t^* = 1$ are not clear. In any case, it seems clear that the best elapsed time usually is obtained for the combination (16×1), the best strategy is the pure distributed one and the best break stage is $t^* > 1$.

Tables 3.7 and 3.8 show the elapsed time (in seconds) that is required for obtaining the lower bounds of the solution value of DEM (3.5) for break stages t^* =1 and t^* =2, respectively, for different strategies on using 16 (task and auxiliary) threads available for our computational experimentation. Notice that CPLEX cannot use more than 8 (auxiliary threads) in its academic version. The smallest elapsed time of the four strategies for each instance in any of the two tables is expressed in bold numbers.

Observe in Table 3.7 that there is not a clear best strategy (*task* × *auxiliary*) for break stage $t^* = 1$. It is worth to point out that $(1 \times 2 \times 8)$ is the best strategy for one of the hardest instance of our testbed, i.e. P4. Additionally, notice that for this instance (where some MIP cluster submodels are very hard to solve), it is useful the assignment of the 8 threads to CPLEX.

On the other hand, observe in Table 3.8 that $(16\times\ 1)$ is the best strategy for

Table 3.5: Elapsed time for solving the root node of instances P5-P9

Instance	task × auxiliary	$t^* = 1$	$t^* = 2$	$t^* = 3$	$t^* = 4$
P5	2×8	0.687	0.986	1.803	2.746
	4×4	0.679	0.385	0.426	0.766
	8×2	**0.490**	0.163	0.184	0.360
	16×1	0.681	**0.130**	**0.132**	**0.220**
	best case	**QD**	**PD**	**PD**	**PD**
P6	2×8	24.823	4.463	7.079	20.633
	4×4	18.075	2.118	2.845	5.939
	8×2	**17.720**	1.240	1.417	2.633
	16×1	20.717	**0.992**	**0.855**	**1.452**
	best case	**QD**	**PD**	**PD**	**PD**
P7	2×8	14.216	9.043	11.825	22.437
	4×4	11.070	4.745	5.984	10.378
	8×2	**8.135**	**2.523**	3.081	5.542
	16×1	8.136	2.932	**2.445**	**3.792**
	best case	**QD**	**QD**	**PD**	**PD**
P8	2×8	12.248	4.757	7.408	20.633
	4×4	**8.628**	2.435	2.611	5.939
	8×2	14.472	1.853	1.295	2.633
	16×1	12.530	**1.502**	**0.769**	**1.452**
	best case	**UD**	**PD**	**PD**	**PD**
P9	2×8	102.835	7.776	10.830	36.961
	4×4	**41.4875**	4.102	3.616	11.850
	8×2	42.998	**3.175**	1.635	4.543
	16×1	54.801	3.325	**1.040**	**2.566**
	best case	**UD**	**QD**	**PD**	**PD**

$t^* = 2$. Notice the remarkable small elapsed time that is required for obtaining the lower bounds in all instances, mainly when they are compared with the times reported in Table 3.7, (where $t^* = 1$). On the other hand the best strategy is (16×1) for $t^* > 2$; the elapsed times are not reported since they are very similar compared with these other ones.

3.7.4 Inner P-BFC strategies versus plain use of CPLEX

Table 3.9 reports the elapsed time for one-path and several task threads (i.e., inner parallelization) that are obtained by P-BFC with $t^* = 1$ versus plain use of CPLEX, considering 8 auxiliary threads in both cases. The algorithm has also been executed for break stages $t^* \geq 2$, but no advantage was obtained (except for instances P6 and P12), probably, due to the fact that the instances were not large enough to consider more cluster submodels and more 0-1 variables to branch even in a parallelized scheme. The additional headings are as follows: n^n, number of twin nodes explored in the $\mathcal{BFT}$ phase of the algorithm; n^{TNF}, number of integer TNF encountered by the algorithm whose associated submodels are solved in Step 7; and t^{DEM}, total

Table 3.6: Elapsed time for solving the root node of instances P10-P14

Instance	task $\times$ auxiliary	$t^* = 1$	$t^* = 2$	$t^* = 3$	$t^* = 4$	$t^* = 5$
P10	2×8	1.098	0.788	0.636	1.121	2.236
	4×4	0.415	0.300	0.366	0.504	0.814
	8×2	**0.317**	0.172	0.192	0.266	0.396
	16×1	0.433	**0.132**	**0.156**	**0.152**	**0.273**
	best case	**QD**	**PD**	**PD**	**PD**	**PD**
P11	2×8	1.199	1.056	1.501	2.104	3.617
	4×4	0.652	0.423	0.453	0.730	1.021
	8×2	**0.618**	0.320	0.269	0.336	0.553
	16×1	0.813	**0.195**	**0.213**	**0.230**	**0.364**
	best case	**QD**	**PD**	**PD**	**PD**	**PD**
P12	2×8	7.102	1.787	1.591	2.579	4.417
	4×4	**4.627**	0.885	0.712	0.995	1.830
	8×2	5.055	0.474	0.379	0.497	0.810
	16×1	5.617	**0.279**	**0.304**	**0.303**	**0.426**
	best case	**UD**	**PD**	**PD**	**PD**	**PD**
P13	2×8	4.405	3.207	3.537	5.103	11.222
	4×4	2.894	1.710	1.533	2.263	4.907
	8×2	**2.677**	0.890	0.904	1.208	2.461
	16×1	3.694	**0.582**	**0.485**	**0.726**	**1.269**
	best case	**QD**	**PD**	**PD**	**PD**	**PD**
P14	2×8	12.647	6.637	9.646	25.780	79.719
	4×4	8.733	3.686	3.962	7.719	21.759
	8×2	**7.150**	2.136	2.121	3.324	9.492
	16×1	7.807	**1.482**	**1.139**	**1.791**	**6.336**
	best case	**QD**	**PD**	**PD**	**PD**	**PD**

Table 3.7: Elapsed time for solving the root node in Testbed 2. Summary for $t^* = 1$

Instance	C	$(1 \times 16 \times 1)$	$(1 \times 8 \times 2)$	$(1 \times 4 \times 4)$	$(1 \times 2 \times 8)$
P1	8	0.177	**0.168**	0.246	0.549
P2	7	0.832	0.588	**0.584**	2.417
P3	10	10.883	10.676	**7.916**	7.928
P4	10	594.424	256.956	185.580	**117.773**
P5	6	0.681	**0.490**	0.679	0.687
P6	8	20.717	**17.720**	18.0875	24.823
P7	7	8.136	**8.135**	11.070	14.216
P8	9	12.530	14.472	**8.628**	12.248
P9	10	54.801	42.998	**41.487**	102.835
P10	5	0.433	**0.317**	0.415	1.098
P11	5	0.813	**0.618**	0.652	1.199
P12	6	5.617	5.055	**4.627**	7.102
P13	7	3.694	**2.677**	2.894	4.405
P14	9	7.807	**7.150**	8.733	12.647

elapsed time (in seconds) for solving DEM (3.5) using different strategies; the elapsed time for strategy $(1 \times 1 \times 8)$ is also reported for plain use of CPLEX.

Table 3.8: Elapsed time for solving the root node in Testbed 2. Summary for $t^* = 2$

Instance	C	$(1 \times 16 \times 1)$	$(1 \times 8 \times 2)$	$(1 \times 4 \times 4)$	$(1 \times 2 \times 8)$
P1	29	**0.109**	0.160	0.305	1.225
P2	24	0.210	**0.204**	0.375	1.408
P3	44	**0.444**	0.662	1.099	3.258
P4	44	**0.943**	1.129	1.826	4.370
P5	16	**0.130**	0.163	0.385	0.986
P6	29	**0.992**	1.240	2.118	4.463
P7	24	2.932	**2.523**	4.745	9.043
P8	39	**1.502**	1.853	2.435	4.757
P9	44	3.325	**3.175**	4.102	7.776
P10	13	**0.132**	0.172	0.300	0.788
P11	13	**0.195**	0.320	0.423	1.056
P12	16	**0.279**	0.474	0.885	1.787
P13	24	**0.582**	0.890	1.710	3.207
P14	39	**1.482**	2.136	3.686	6.637

Notice that the combination $(1 \times 1 \times 8)$ is the serial version. The performance of P-BFC is remarkable. Observe also that plain use of CPLEX cannot provide a solution for instances P3, P4, P5, P8, P9 and P12. By contrast, P-BFC requires a very small elapsed time, even for the hardest instance P4 in the testbed.

Table 3.9: Inner P-BFC for $t^* = 1$ versus plain use of CPLEX. Testbed 2.

	Nonsymmetric P-BFC with CPLEX from COIN-OR (main × task × auxiliary) threads							Plain Use of CPLEX
			t^{DEM}					t^{DEM}
Instance	n^n	n^{TNF}	$(1 \times 1 \times 8)$	$(1 \times 2 \times 8)$	$(1 \times 4 \times 8)$	$(1 \times 8 \times 8)$	$(1 \times 16 \times 8)$	$(1 \times 1 \times 8)$
P1	1	0	1	2	2	3	5	23
P2	3	1	45	45	37	37	42	3049
P3	24	6	332	239	228	216	214	—
P4	12	6	1727	1628	1515	1457	1438	—
P5	1	0	1	1	1	2	3	—
P6	3	2	88	69	56	50	44	4239
P7	3	2	115	74	68	60	60	461
P8	1	0	26	20	29	23	21	—
P9	1	0	109	68	54	41	44	—
P10	1	0	1	1	1	2	2	25
P11	3	2	10	6	5	5	5	530
P12	3	2	47	36	25	24	24	—
P13	3	2	40	25	24	24	23	1187
P14	3	2	119	97	97	92	92	509

—: Out of memory (48Gb) or time limit exceeded (6h)

3.7.5 Outer P-BFC strategies versus plain use of CPLEX

Table 3.10 reports the elapsed time required for solving the hardest instances in the testbed, i.e., P3 and P4, using P-BFC, alternatively, with several-paths and one task thread (i.e., outer parallelization) versus one-path and several task threads (i.e., inner parallelization) for $t^* = 1$.

A constant number of 8 threads is considered for the optimizer (in both cases, inner and outer versions of the algorithm). The additional headings are S_{th}, speedup, defined as $S_{th} = \frac{t^{DEM}_{serial}}{t^{DEM}_{inner}}$ and $E_{th}\%$, efficiency, defined as $E_{th}\% = 100 \cdot \frac{S_{th}}{th}$, where th is the total number of computing nodes (8 threads each) used by the corresponding P-BFC. It can be observed that the greater number of paths, the smaller elapsed time to obtain the optimal solution. The savings in elapsed time obtained by using 16 paths rather than a single one in P3 and P4 are 71.99% and 61.49%, respectively. This is a remarkable time reduction. Additionally, notice that plain use of CPLEX cannot solve the instances, see Table 3.9, due to either running out of memory (48Gb) or exceeding the 6 hours time limit.

Table 3.10: Inner and Outer P-BFC scalability for $t^* = 1$ in P3 and P4

Inner P-BFC						
Strategy	P3			P4		
$(1 \times b \times 8)$	t^{DEM}	S_b	$E_b\%$	t^{DEM}	S_b	$E_b\%$
$(1 \times 1 \times 8)$	332	1.00	100	1727	1.00	100
$(1 \times 2 \times 8)$	239	1.39	69.46	1628	1.06	53.04
$(1 \times 4 \times 8)$	228	1.46	36.40	1515	1.14	28.50
$(1 \times 8 \times 8)$	216	1.54	19.21	1457	1.19	14.82
$(1 \times 16 \times 8)$	214	1.55	9.70	1438	1.20	7.51

Outer P-BFC						
Strategy	P3			P4		
$(a \times 1 \times 8)$	t^{DEM}	S_a	$E_a\%$	t^{DEM}	S_a	$E_a\%$
$(1 \times 1 \times 8)$	332	1.00	100	1727	1.00	100
$(2 \times 1 \times 8)$	193	1.72	86.01	1168	1.48	73.93
$(4 \times 1 \times 8)$	131	2.53	63.36	945	1.83	45.69
$(8 \times 1 \times 8)$	99	3.35	41.91	702	2.46	30.75
$(16 \times 1 \times 8)$	93	3.57	22.30	665	2.60	16.23

Figure 3.6 depicts the comparison in *speed up* and *efficiency* for instance P3 of the strategies of the algorithm with one-path and different task threads $(1 \times b \times 8)$ (i.e., inner parallelization alone), and the strategies with several paths and one task thread $(a \times 1 \times 8)$ (i.e., outer parallelization alone), where $a, b = 1, 2, 4, 8, 16$.

Table 3.11 reports the elapsed time for all the instances in the testbed for $t^* = 1$ using several paths in P-BFC. The time reduction is remarkable. The computational experimentation with $t^* = 2$ does not show any improvement over $t^* = 1$, except for P6 and P12.

The inner and outer parallelization strategies perform better than plain use of CPLEX even on the hardest instances, since CPLEX cannot solve them. Notice that the parallelization strategies obtain the solution value for all the instances in

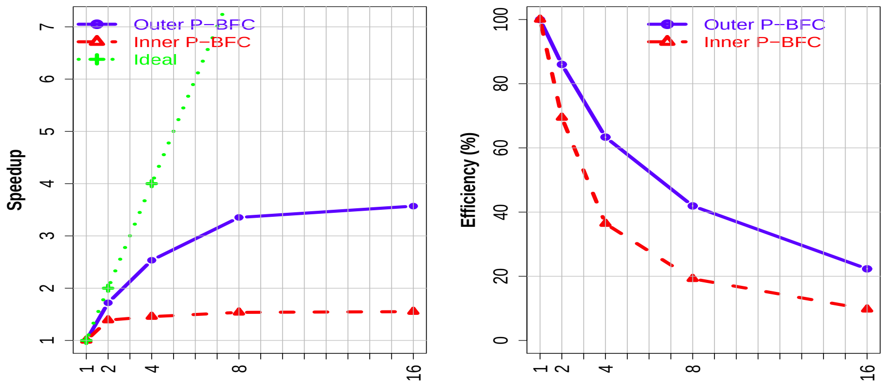

Figure 3.6: Scalability of instance P3. Outer P-BFC versus Inner P-BFC

Table 3.11: Outer P-BFC for $t^* = 1$ versus plain use of CPLEX. Testbed 2.

	Nonsymmetric P-BFC with COIN-OR/CPLEX (main × task × auxiliary) thread										Plain Use of CPLEX
	$(1\times1\times8)$		$(2\times1\times8)$		$(4\times1\times8)$		$(8\times1\times8)$		$(16\times1\times8)$		$(1\times1\times8)$
Instance	n^n	t^{DEM}	n^n	t^{DEM}	n^n	t^{DEM}	n^n	t^{DEM}	n^n	t^{DEM}	t^{DEM}
P1	1	1	1	1	1	3	1	6	1	14	23
P2	3	45	1	47	1	14	1	21	1	26	3049
P3	24	332	11	193	9	131	5	99	4	93	—
P4	12	1727	7	1168	5	945	4	702	4	665	—
P5	1	1	1	2	1	2	1	6	1	9	—
P6	3	88	1	36	1	48	1	46	1	75	4239
P7	3	115	1	61	1	75	1	96	1	64	461
P8	1	26	1	35	1	50	1	51	1	62	—
P9	1	109	1	135	1	123	1	128	1	258	—
P10	1	1	1	2	1	4	1	33	1	36	25
P11	3	10	1	4	1	4	1	8	1	14	530
P12	3	47	1	20	1	23	1	27	1	34	—
P13	3	40	1	25	1	71	1	85	1	100	1187
P14	3	119	1	82	1	106	1	84	1	81	509

—: Out of memory (48Gb) or time limit exceeded (6h).

the testbed. However, outer parallelization generally requires less elapsed time than inner parallelization, see the results reported in Table 3.9 and Table 3.11. This is mainly because the great advantage that the synchronization phase of the outer parallelization strategy has in reducing the number of nodes visited. It is due to the bound updating and the dynamic allocation of the active paths, thus, assuring better use of computer resources.

3.7.6 Outer-Inner P-BFC strategies versus plain use of CPLEX

The computational results of the instances of the testbed for the joint outer-inner parallelization strategy are reported in Table 3.12 in line with the best elapsed time obtained by outer parallelization alone. The best ones are obtained with break stage $t^* = 1$ for all instances, but P6 and P12 whose results are obtained with $t^* = 2$. Five blocks of thread assignments are considered in the table.

Table 3.12: Outer-Inner P-BFC versus plain use of CPLEX. Testbed 2.

		Nonsymmetric P-BFC with CPLEX from COIN-OR (main × task × auxiliary) threads										Plain Use of CPLEX
		8 threads		16 threads		32 threads		64 threads		128 threads		8 threads
Instance	t^*	n^n	t^{DEM}	n^n	t^{DEM}	n^n	t^{DEM}	n^n	t^{DEM}	n^n	t^{DEM}	t^{DEM}
Strategy		$(1 \times 1 \times 8)$		$(1 \times 2 \times 8)$		$(1 \times 4 \times 8)$		$(1 \times 8 \times 8)$		$(1 \times 16 \times 8)$		$(1 \times 1 \times 8)$
P1	1	1	1	1	2	1	2	1	3	1	5	23
P5	1	1	1	1	1	1	1	1	2	1	3	—
P8	1	1	26	1	20	1	29	1	23	1	21	—
P9	1	1	109	1	68	1	54	1	41	1	44	—
P10	1	1	1	1	1	1	1	1	2	1	2	25
Strategy		$(1 \times 1 \times 8)$		$(2 \times 1 \times 8)$		$(2 \times 2 \times 8)$		$(2 \times 4 \times 8)$		$(2 \times 8 \times 8)$		$(1 \times 1 \times 8)$
P2	1	3	45	1	47	1	41	1	49	1	52	3049
P7	1	3	115	1	61	1	49	1	65	1	64	461
P11	1	3	10	1	4	1	4	1	8	1	8	530
P13	1	3	40	1	25	1	20	1	18	1	35	1187
P14	1	3	119	1	82	1	75	1	69	1	90	509
Strategy		$(1 \times 1 \times 8)$		$(2 \times 1 \times 8)$		$(4 \times 1 \times 8)$		$(8 \times 1 \times 8)$		$(8 \times 2 \times 8)$		$(1 \times 1 \times 8)$
P3	1	24	332	11	193	9	131	5	99	5	114	—
Strategy		$(1 \times 1 \times 8)$		$(2 \times 1 \times 8)$		$(4 \times 1 \times 8)$		$(4 \times 2 \times 8)$		$(4 \times 4 \times 8)$		$(1 \times 1 \times 8)$
P4	1	12	1727	7	1168	5	945	5	879	5	789	—
Strategy		$(1 \times 1 \times 8)$		$(2 \times 1 \times 8)$		$(2 \times 2 \times 8)$		$(2 \times 4 \times 8)$		$(2 \times 8 \times 8)$		$(1 \times 1 \times 8)$
P6	2	3	38	1	21	1	26	1	22	1	50	4239
P12	2	3	15	14	24	14	19	14	21	14	29	—

—: Out of memory (48Gb) or time limit exceeded (6h).

Table 3.12 shows the behaviour of the joint Outer-Inner P-BFC strategy. The first block comprises instances P1, P5, P8, P9 and P10 whose optimization is obtained in one node, only one-path and an increased number of task threads is considered. Observe that the marginal effect of the Inner P-BFC is always higher than when using the Outer P-BFC, that is, the most efficient parallel strategy corresponds to pure Inner P-BFC. Since just one node is visited on the serial BFC no gain is obtained by adding paths. The second block comprises instances P2, P7, P11, P13 and P14, where initially the marginal effect of the Outer P-BFC is higher reducing the visited nodes to one for strategy $(2\times\ 1 \times 8)$. It is again more efficient to increase threads in inner paradigm for strategies with two paths. More interestingly, instances P3, P4, P6 and P12 show that depending on the ratio of number/scale of nodes, inner or outer can present a higher marginal effect, taking into account that the maximum number of available threads is 128. The third block comprises only instance P3, where the number of paths is up to 8 and the number of task threads is small. The fourth block comprises only instance P4, with up to 4 paths, where there is a mixture of paths and task threads. The fifth block comprises instances P6 and P12. Up to two paths are considered for P6, since it is the outer strategy that provides the best results for the instance. The best strategy for P12 does not require outer parallelization. Every time a new group of 8 threads is added the related higher marginal effect can balance from one paradigm to other. Consequently, we find parallel executions (with outer, inner and outer-inner strategies) to be the most efficient ones for these instances.

In brief, the outer paradigm reduces the number of visited twin nodes, whereas the inner paradigm reduces the computing time of each node (remember that model (3.10) is currently not subdivided and therefore serially executed). Additionally, the break stage selection has a major impact on the marginal effect balance since a higher break stage implies more visited but smaller scaled nodes.

Table 3.13: Best P-BFC strategies summary. Testbed 2.

	Nonsymmetric P-BFC					CPLEX
Instance	t^*	*best strategy*	n^n	t^{DEM}	*sratio* (%)	t^{DEM}
P1	1	$(1 \times 1 \times 8)$	1	1	0	23
P2	1	$(4 \times 1 \times 8)$	1	14	68.89	3049
P3	1	$(16 \times 1 \times 8)$	4	93	71.99	—
P4	1	$(16 \times 1 \times 8)$	4	665	61.49	—
P5	1	$(1 \times 1 \times 8)$	1	1	0	—
P6	2	$(2 \times 1 \times 8)$	2	21	76.14	4239
P7	1	$(2 \times 2 \times 8)$	1	49	57.39	461
P8	1	$(1 \times 4 \times 4)$	1	16	38.46	—
P9	1	$(1 \times 8 \times 8)$	1	41	62.39	—
P10	1	$(1 \times 1 \times 8)$	1	1	0	25
P11	1	$(2 \times 1 \times 8)$	1	4	60	530
P12	2	$(1 \times 1 \times 8)$	3	15	68.09	—
P13	1	$(2 \times 4 \times 8)$	1	18	55	1187
P14	1	$(2 \times 4 \times 8)$	1	69	42.02	509

—: Out of memory (48Gb) or time limit exceeded (6h).

Observe that the instances P3 and P4 are the hardest ones in the testbed, but they are solved efficiently. It is worth pointing out the performance of the outer-inner parallelization versus the serial one, since the elapsed time of the latter is 332 (secs) for P3 and 1727 (secs) for P4, while the elapsed time for the former is 114 and 789, respectively.

Figure 3.7 and Table 3.13 summarize the best strategy for each instance in the testbed. The additional headings are as follows: *best strategy*, $(a \times b \times h)$ with the smallest elapsed time; and *sratio* (%), saving ratio of the elapsed time of the best strategy and the time of the serial one $(1 \times 1 \times 8)$. It can be observed that the elapsed time of the parallelization strategy is substantially smaller than that of the serial one in 11 of the 14 instances of the testbed. In all cases, both types of versions solve all the instances in a time that is more than one order of magnitude smaller than the time required by plain use of CPLEX. Based on those results, the break stage of choice is $t^* = 1$ if the computer resources available allows it. Otherwise, the value should be increased.

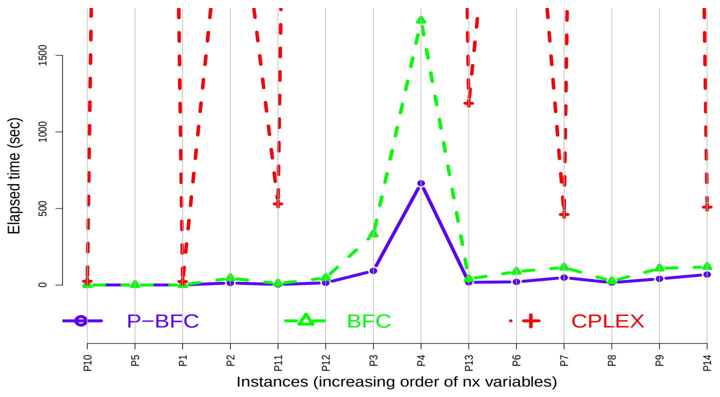

Figure 3.7: Elapsed time comparison for P-BFC, serial BFC and CPLEX

3.8 Conclusions

The elapsed time for obtaining the optimal solution is much more smaller in the serial version of the decomposition algorithm BFC than in the plain use of a state-of-the-art MIP solver, being always reliable, something that in the plain use of the MIP solver is frequently not.

Several parallelization strategies have been introduced into the BFC methodology for obtaining faster risk neutral optimal solutions to medium-scale and large-scale multistage stochastic mixed 0-1 problems, where (exogenous) uncertainty is represented by nonsymmetric scenario trees. The parallel computing version of the algorithm is referred to as P-BFC.

Parallelization is performed in two levels. The inner level parallelizes the optimization of MIP submodels attached to the set of scenario clusters created by the modeler-defined break stage, say t^*. Based on this stage the model is represented by a mixture of the splitting variable and compact representations, such that the non-anticipativity constraints (NAC) up to stage t^* are explicitly stated. The scenario clusters MIP submodels from stage $t^* + 1$ (one for each scenario group in the stage) implicitly satisfy the NAC for the scenario groups that belong to any stage $t > t^* + 1$. Since the cluster submodels are independent, they are optimized in parallel in Step 1 of the algorithm to obtain a strong lower bound of the solution value of the original stochastic problem, in Step 6 while iterating the Branch-and-Fix Coordination approach of the algorithm until the break stage and, finally, in Step 7 to be executed at each integer TNF to satisfy the explicit NAC. Based on an extensive computational experience to assess the validity of the proposed approach, the results of the inner parallelization are remarkable in the broad testbed used for the experiment. Compared to the serial version even in the case where the CPLEX solver is allocated the maximum number of threads (8, in the academic version), the results basically depend on the number of task threads that are allocated for parallel optimization of the cluster submodels. Notice that the elapsed time usually is one order of magnitude smaller than the time required by plain use of CPLEX. Additionally, the optimizer alone very frequently fails to solve the problem. The elapsed time can be improved still further if the parallelization of the execution of model (3.10) in Step 7 of the Inner P-BFC is performed, by decomposing it via the iterative dualization of its NAC of the y- variables. For this purpose, the Clustering Lagrangian Decomposition approach can be used, see Escudero *et al.* [2013a] for the two-stage case. Its adaptation to P-BFC provides a direction for future work.

The outer level of parallelization is based on the path concept. In this chapter a path is defined by a main thread managing a serial Branch-and-Fix Coordination algorithm and the combinations of a set of 0-1 variables as initial condition, such that each one can itself be internally optimized with the inner parallelization scheme (multiple task threads) or without it (single task thread). The results of using the outer parallelization alone are very good, so that the larger the number of paths, the smaller the elapsed time to obtain the optimal solution.

However, the full parallelization of the algorithm which consists of using inner parallelization on the paths resulting from outer parallelization provides the optimal solution to the original stochastic problem in an astonishingly small elapsed time. That time can be several orders of magnitude smaller than the time required by the serial version of the algorithm, specially for break stage $t^* = 1$ if the computer resources allow it. Otherwise, the break stage value should be increased, but the results are still remarkable.

Chapter **4**

Parallel Stochastic Dynamic Programming

Arrange whatever pieces come your way.
Virginia Woolf

4.1 Introduction

The main objective of this chapter consists of presenting inner and outer parallelization versions of the Stochastic Dynamic Programming (SDP) algorithm, referred to as P-SDP, so that the solution quality improvement and elapsed time reduction in problem solving are analyzed. The inner version parallelizes the optimization of the MIP subproblems attached to the set of subtrees that have been created by the modeler-defined stages. A strategy is presented for analyzing the performance when using parallel computing based on a message-passing scheme for solving the stage subtrees based MIP subproblems versus the serial version of the SDP algorithm. The outer version of P-SDP optimizes the problem solving from paths, which are solved in parallel at the threads and where each path execution depends on the results obtained by the other ones along the algorithm. The main results of a broad computational experience are reported to assess whether the performance of the parallel computing approach compares favorably to the serial one. The elapsed time required by the inner parallelization is up to one order of magnitude smaller than that of the serial version of the algorithm, where efficiency is up to 90% when using 12 threads, and the performance depends on computer resources availability. Therefore, the larger the number of task threads, the smaller the elapsed time required for problem solving. Additionally, the Outer P-SDP can improve the solution quality versus the serial one, even for instances where the state-of-the-art engine of choice, CPLEX, cannot even provide a feasible solution. As an illustrative example of the computational performance of the new approach for solving a realistic production planning problem under uncertainty, for a large-scale instance whose dimensions are 5.56 million constraints, 1.41 million 0-1 variables and 3.49 million continuous variables, Inner P-SDP and SDP give a solution value with a 0.16% gap versus plain use of CPLEX V.12.5 requiring an elapsed time of 978 seconds for Inner P-SDP and 7220 for SDP, while CPLEX is stopped since it was running out of memory (35Gb) after 8274 seconds whose solution value has a 0.78% quasi-optimality gap at that time instant. On the other hand, plain use of CPLEX was stopped running out of memory after 3003 seconds while solving the LP relaxation of the largest instance in the testbed that we have experimented with, whose dimensions are 57.8 million constraints, 15.4 million 0-1 variables and 38.5 million continuous variables. For that instance, Inner P-SDP and SDP required 2446 and 26180 seconds, respectively, to provide a solution whose value is the same but, obviously, its quality could not be assessed, see Aldasoro *et al.* [2014].

The rest of the chapter is organized as follows: Section 4.2 presents the serial version of the SDP algorithm to be parallelized. Section 4.3 introduces the Parallel Stochastic Dynamic Programming (P-SDP) schemes. Section 4.4 reports the main results of a broad computational experience to assess the validity of the parallel versions of the SDP algorithm versus its serial one and plain use of CPLEX. Section 4.5 summarizes the main conclusions.

4.2 SDP decomposition algorithm

The section presents the SDP algorithm for solving multiperiod stochastic mixed 0-1 programs whose versions for particular problems have been introduced in Cristobal *et al.* [2009]; Escudero *et al.* [2013b]. Section 4.2.1 decomposes the time horizon into stages. The algorithmic framework is presented in Section 4.2.2. The crucial ingredient of the algorithm, the so-named Expected Future Value (EFV) curve, is explained in detail in Section 4.2.3.

4.2.1 Breaking the time horizon into stages

The SDP algorithm combines consecutive time periods into stages, as shown in Figure 4.1. The following additional notation for the stages is used throughout the chapter:

$\mathcal{P}$**,** set of periods along the time horizon.

$a(g)$**,** immediate ancestor node of node g, for $g \in \mathcal{G}$.

$t(g)$**,** stage to which scenario group g belongs, therefore, $g \in \mathcal{G}_{t(g)}$.

$\mathcal{R}^t$, set of nodes (i.e, scenario groups) in the earliest period of stage t, for $t \in \mathcal{T}$, therefore, $\mathcal{R}^t$ is the set of root nodes of the subtrees in the scenario tree related to stage t, for $t \in \mathcal{T}$.

$\mathcal{M}_r \subseteq \mathcal{G}_t$, set of nodes in the subtree rooted at node $r \in \mathcal{R}^t$ included in stage t, for $t \in \mathcal{T}$.

$\mathcal{L}_r$, set of leaf nodes from $\mathcal{M}_r$ in the subtree whose root node is r, for $r \in \mathcal{R}^t$, $t \in \mathcal{T}$.

$\mathcal{S}_\ell$, set of immediate successor scenario groups to scenario group ℓ, for $\ell \in \mathcal{L}_r$, $r \in \mathcal{R}^t$, $t \in \mathcal{T}$. Note that $\mathcal{S}_\ell = \emptyset$ for $\ell \in \mathcal{G}_T$. Also note that $\bigcup_{r' \in \mathcal{S}_\ell} \mathcal{M}_{r'}$ is the node set in the scenario subtrees related to those immediate successor subproblems of node ℓ.

$\tilde{\mathcal{A}}_\ell$, set consisting of leaf node $\ell \in \mathcal{L}_r$ and its ancestor nodes in $\mathcal{A}_\ell$, such that their variables have nonzero elements in constraints associated with the nodes in the immediate successor subproblems to node ℓ, for $\ell \in \mathcal{L}_r, r \in \mathcal{R}^t$, $t \in \mathcal{T} \backslash \{|\mathcal{T}|\}$.

The splitting of time horizon $\mathcal{P}$ into the set of stages $\mathcal{T}$ and, then, the decision of the modeler defined composition should take into account the dimensions of the constraint systems for the sets $\mathcal{M}_r \, \forall r \in \mathcal{R}^t$ as well as the cardinality of set $\mathcal{A}_\ell$ for $\ell \in \mathcal{L}_r$.

Once the time horizon has been split into stages, the DEM can be divided into subproblems, which are connected by the linking variables. For each $r \in \mathcal{R}^t$, $t \in \mathcal{T}$, let us associate a subproblem with the subtree defined by node set $\mathcal{M}_r$, being r its root node. In Figure 4.1, $\mathcal{M}_5 = \{5, 8, 9\}$ defines a subtree (to whom a subproblem is attached) in stage $t = 2$ with node 5 as its root. In this example, there are in total eleven subtrees/subproblems (they are marked by dashed boxes).

The subproblem defined by node set $\mathcal{M}_{r'}$, $r' \in \mathcal{R}^{t-1}$, $t \in \mathcal{T} \setminus \{1\}$ can be written as follows,

$$\begin{aligned}
F'_{r'}(\overline{V}_{a(r')}) := \min & \sum_{\ell \in \mathcal{L}_{r'}} w_\ell \Big(\sum_{g \in \mathcal{A}_\ell} (a_g x_g + b_g y_g) + \lambda_\ell(\cdot) \Big) \\
\text{s.t.} & \sum_{q \in \mathcal{A}_g} (A^g_q x_q + B^g_q y_q) = h_g && \forall g \in \mathcal{M}_{r'} \\
& x_g \in \{0,1\}^{nx(g)},\ 0 \le y_g \le \hat{y}_g && \forall g \in \mathcal{A}_\ell,\ \ell \in \mathcal{L}_{r'} \\
& x_g = \overline{x}_g,\ y_g = \overline{y}_g && \forall g \in \mathcal{A}_{a(r')},
\end{aligned} \tag{4.1}$$

where $\overline{V}_g = (\overline{x}_g,\ \overline{y}_g)$, such that $\overline{x}_g$ and $\overline{y}_g$ are the solution of the variables in vectors x_g and y_g, respectively, $V_{a(r')}$ is the vector of x- and y-variables of

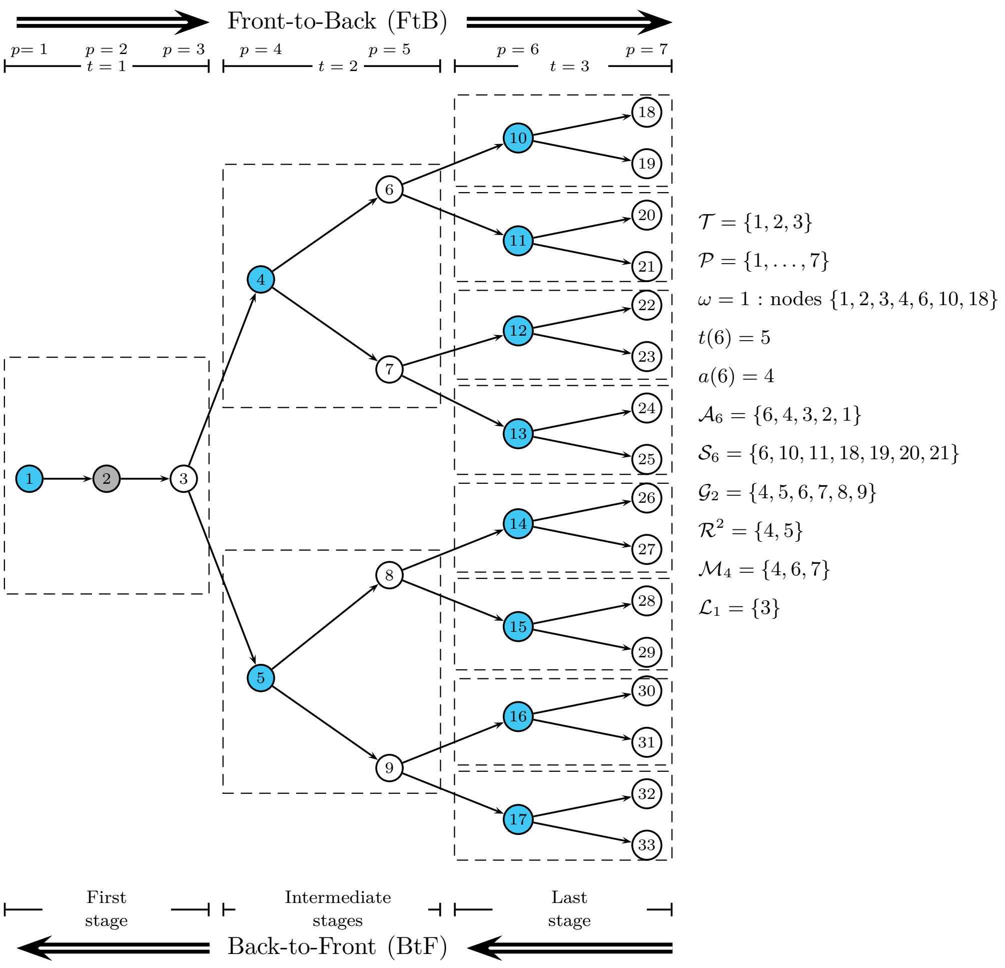
Front-to-Back (FtB)
p= 1
$p = 2$
$t = 1$
$p = 3$
$p = 4$
$t = 2$
$p = 5$
$p = 6$
$t = 3$
$p = 7$
$\mathcal{T} = \{1, 2, 3\}$
$\mathcal{P} = \{1, \dots, 7\}$
$\omega = 1$: nodes $\{1, 2, 3, 4, 6, 10, 18\}$
$t(6) = 5$
$a(6) = 4$
$\mathcal{A}_6 = \{6, 4, 3, 2, 1\}$
$\mathcal{S}_6 = \{6, 10, 11, 18, 19, 20, 21\}$
$\mathcal{G}_2 = \{4, 5, 6, 7, 8, 9\}$
$\mathcal{R}^2 = \{4, 5\}$
$\mathcal{M}_4 = \{4, 6, 7\}$
$\mathcal{L}_1 = \{3\}$
First stage
Intermediate stages
Last stage
Back-to-Front (BtF)

Figure 4.1: An example of multiperiod scenario tree

the ancestor scenario groups g to root node r' (without including itself) that have nonzero elements in the constraints of the scenario groups in set $\mathcal{M}_{r'}$ (i.e, $V_{a(r')} = \{x_g,\ y_g \forall g \in \tilde{\mathcal{A}}_{a(r')}\}$), and $\lambda_\ell(\cdot)$ $\lambda_\ell(\cdot)$ (assumed to be convex) gives the expected solution value of the immediate successor subproblems of leaf node ℓ. Note that function $\lambda_\ell(\cdot)$ depends upon the values of x- and y-variables whose vectors belong to set $\tilde{\mathcal{A}}_\ell$. Finally, $F'_{r'}(\overline{V}_{a(r')})$ is the solution value of model (4.1).

Figure 4.2 shows an example of function $\lambda_\ell(\cdot)$ and its approximation (EFV curve) for node ℓ. Two straightforward observations need to be made here. First, no decisions are taken prior to the root node $r' = 1$. And, second, no EFV curves are present in the subproblems in the last stage since the time horizon ends there.

Functions $\lambda_\ell(\cdot)$ for $\ell \in \mathcal{L}_{r'}$, $r' \in \mathcal{R}^{t-1}$, $t \in \mathcal{T} \setminus \{1\}$ and $F_r^{'}(\cdot)$ for $r \in \mathcal{S}_\ell$ are closely related. The immediate successor subproblems of node ℓ (i.e., successor subproblems in stage t) are given by the node sets $\mathcal{M}_r$, $\forall r \in \mathcal{S}_\ell$. We can express $\lambda_\ell(\cdot)$ as the weighted sum of the optimal objective function value of those subproblems:

$$\lambda_\ell(\cdot) = \sum_{r \in \mathcal{S}_\ell} w_r F_r^{'}(\cdot). \tag{4.2}$$

4.2.2 SDP decomposition algorithm

Since the $\lambda(\cdot)$ curves are generally difficult to compute, the SDP decomposition algorithm proposes to approximate them by piecewise linear convex functions. For that purpose, let a set of *reference levels* for subproblem r' such that the zzth reference level is included by parameter μ_ℓ^z and parameter vector π_ℓ^z to define the piecewise linear function. Figure 4.2 depicts an example of the $\lambda_\ell(\cdot)$ curve as well as its approximation EFV.

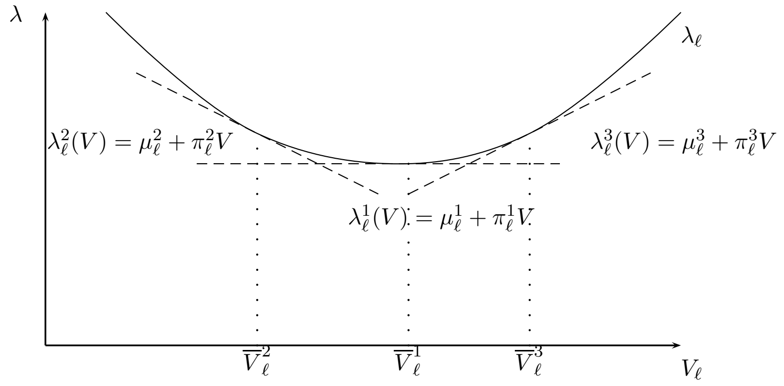

Figure 4.2: An example of $\lambda_\ell(\cdot)$ curve and its approximation EFV

The approximation of subproblem (4.1) can be expressed

$$\begin{aligned}
F_{r'}(\overline{V}_{a(r')}) := \min \; & \sum_{\ell \in \mathcal{L}_{r'}} w_\ell \Big(\sum_{g \in \mathcal{A}_\ell} (a_g x_g + b_g y_g) + \lambda_\ell \Big) & \\
\text{s.t.} \; & \sum_{q \in \mathcal{A}_g} (A_q^g x_q + B_q^g y_q) = h_g & \forall g \in \mathcal{M}_{r'} \\
& x_g = \overline{x}_g, \; y_g = \overline{y}_g & \forall g \in \mathcal{A}_{a(r')} \\
& x_g \in \{0,1\}^{nx(g)}, \; 0 \le y_g \le \hat{y}_g & \forall g \in \mathcal{A}_\ell, \; \ell \in \mathcal{L}_{r'} \\
& \lambda_\ell \ge \mu_\ell^z + \pi_\ell^z V_\ell & \forall z \in \mathcal{Z}_\ell, \; \ell \in \mathcal{L}_{r'}.
\end{aligned} \tag{4.3}$$

We will refer hereafter to the last constraints in (4.3) as the EFV curve constraints, where $\mathcal{Z}_\ell$ $\mathcal{Z}_\ell$ is the set of indices of currently active reference levels defining EFV curve constraints. It is trivial to see that a better approximation of function $\lambda_\ell(\cdot)$ is obtained increasing $|\mathcal{Z}_\ell|$, provided that any new reference level defines an active constraint.

The SDP algorithm adopts an approach where the EFV curves are refined iteratively using recursion. See other approaches in Pereira and Pinto [1991]; Ross [1995]; Shapiro *et al.* [2013]; Piazza and Pagnoncelli [2014]. Each iteration of the decomposition algorithm consists of a *Front-to-Back* (FtB) scheme followed by a *Back-to-Front* (BtF) scheme, see Figure 4.1. The FtB scheme is aimed at building a feasible solution ($x = \overline{x}$, $y = \overline{y}$) for the problem and checking whether it improves the incumbent solution, say (x^*, y^*). Subproblems in stage $t = 1$ to stage $t = |\mathcal{T}|$ are serially solved, passing the values of linking variables (vector $\overline{V}_\ell$, $\ell \in \mathcal{L}_{r'}, r' \in \mathcal{R}^{t-1}$) for stage $t > 1$ onto the immediate successor subproblems rooted at node r for all $r \in \mathcal{S}_\ell$ for refining the EFV curves of subproblem rooted at node r'. The refinement consists of creating a new reference level for each leaf node ℓ where $\ell \in \mathcal{L}_{r'}$, so that a new EFV point defining constraint is appended to the subproblem. Note: Non-active EFV curve defining constraints could be deleted at this point and the sets $\mathcal{Z}_\ell$ are updated at every, say, *nitek* iterations.

The BtF scheme is aimed at refining the EFV curves around the partial feasible solutions $\overline{V_\ell}$ (obtained for each subproblem in the FtB scheme executed at past iterations), by serially solving the subproblems r for $r \in \mathcal{R}^t$ in the last stage $t = |\mathcal{T}|$ to stage $t = 2$ and passing the refinement of the EFV curves onto the subproblems in the previous stages. The refinement consists of creating a new reference level for each active level in $\mathcal{Z}_\ell$ for each leaf node $\ell = a(r)$ where $\ell \in \mathcal{L}_{r'}$ at each subproblem $r' \in \mathcal{R}^{t-1}$, $t \in \mathcal{T} \setminus \{1\}$ once all subproblems rooted at nodes $r \in \mathcal{S}_\ell$ have been solved (see below). Note that $|\mathcal{Z}_\ell|$ is doubled at the end of the execution of the scheme for each iteration as it is the number of new points defining EFV curve constraints for each leaf node ℓ that are appended to the subproblems.

The stopping criteria for the algorithm at the end of the FtB scheme in any iteration, say *iter* is as follows:

1. The relative absolute difference between the solution value that has been obtained in the iteration, say $\overline{F} = \sum_{g \in \mathcal{G}} w_g(a_g\overline{x}_g + c_g\overline{y}_g)$, and the incumbent one of the original problem (1.4), say $F^* = \sum_{g \in \mathcal{G}} w_g(a_g x_g^* + c_g y_g^*)$, is below a given tolerance parameter, say $\epsilon_1 > 0$ *and* additionally the relative absolute difference between the solution values $\overline{F}$ at two consecutive iterations (i.e., *iter* -1 and *iter*) is below a given tolerance parameter, say $\epsilon_2 > 0$.

2. An upper bound on the number of iterations, say *miter*, is reached.
3. The allowed elapsed time is reached or the algorithm is running out of memory.

A detailed procedure of the serial SDP decomposition algorithm is presented in Algorithm 4.1, including the initialization and both the FtB and the BtF schemes in which each iteration is split. In the next section, the refinement of the EFV curves is discussed in detail, i.e., how to obtain the set of new reference levels included by the parameters giving the new linear piecewise functions to improve the EFV curves approximation of the $\lambda(\cdot)$ curves

4.2.3 $\lambda(\cdot)$ curves approximation

This section explains how the EFV approximation of $\lambda_\ell(\cdot)$ is refined at each iteration for $\ell \in \mathcal{L}_{r'}$, $r' \in \mathcal{R}^t$, $t \in \mathcal{T} \setminus \{1\}$.

Let $F_r(\overline{V}_\ell^z)$ denote the solution value of model (4.3) obtained for a given reference level numbered as z, where $\overline{V}_\ell^z$ is the fixed value of vector V_ℓ^z in the constraint system $x_g = \overline{x}_g,\ y_g = \overline{y}_g\,\forall g \in \tilde{\mathcal{A}}_\ell$.

Let z' be the identification of the new reference level to be created, such that $z' = z$ whenever it is created in the FtB scheme of the given iteration, and it is $z' = |\mathcal{Z}_\ell| + z$ if it is created in the BtF scheme, where the z-th reference level was created either in the FtB scheme of the same iteration or in any of both schemes in any previous iteration (see procedure SDP below).

The z'-th reference level is included by parameter $\mu_\ell^{z'}$ and parameter vector $\pi_\ell^{z'}$ that are obtained using an ad-hoc sensitivity analysis of solution value $F_r(\overline{V}_\ell^z)$ on a small perturbation, say $\xi_r^{z'}$ $\xi_r^{z'}$, performed on the linking constraint system $V_\ell^z = \overline{V}_\ell^z$ in the subproblems for all $r \in \mathcal{S}_\ell$, such that let $\pi_r^{z'}$ be the dual vector of those constraints. The i-th element of vector $\xi_r^{z'}$, say $(\xi_r^{z'})_i$ can be computed as follows:

$$(\xi_r^{z'})_i = ran^{z'}/f(iter),$$

such that $0 \leq (ran^{z'})_i \leq f$ for $(x_g)_i$-variable and $0 \leq (ran^{z'})_i \leq f \times (\hat{y}_g)_i$ for $(y_g)_i$-variable is the randomly generated i-th element of vector $ran^{z'}$, where $(x_g)_i$ and $(y_g)_i$ are the x- and y- variables that correspond to the i-th element of vector V_ℓ, f is a multiplicative factor such that $f \in (0,1]$ and $f(iter)$ is a monotonically non-decreasing function on the current iteration number $iter$. Therefore, it results

$$F_r(\overline{V}_\ell^{z'} + \xi_r^{z'}) \approx F_r(\overline{V}_\ell^{z'}) + \pi_r^{z'}\xi_r^{z'} = \mu_r^{z'} + \pi_r^{z'} \times (\overline{V}_\ell^{z'} + \xi_r^{z'})$$

where

$$\mu_r^{z'} = F_r(\overline{V}_\ell^{z'}) - \pi_r^{z'}\overline{V}_\ell^{z'}. \tag{4.4}$$

Note that due to the assumed convexity of function $\lambda_\ell(\cdot)$, it results

$$F_r'(\overline{V}_\ell^{z'} + \xi_r^{z'}) \leq F_r(\overline{V}_\ell^{z'} + \xi_r^{z'}).$$

The $\mu-$ and $\pi-$ expected values for scenario group ℓ over the related values of its immediate successor subproblems (and, therefore, rooted at nodes $r \in \mathcal{S}_\ell$ for the node sets $\mathcal{M}_r$) can be expressed

$$\mu_\ell^{z'} = \sum_{r \in \mathcal{S}_\ell} w_r \mu_r^{z'} \tag{4.5}$$

$$\pi_\ell^{z'} = \sum_{r \in \mathcal{S}_\ell} w_r \pi_r^{z'}. \tag{4.6}$$

The SDP algorithm for chosen set of stages $\mathcal{T}$ and maximum number of iterations *miter* is described in Algorithm 4.1.

4.3 On parallelizing the SDP decomposition algorithm

4.3.1 P-SDP parallelization strategies

The SDP algorithm can be parallelized in different ways, depending on the goal to be achieved, i.e., reducing the execution time or improving the incumbent solution value. The first paradigm consists of sharing, when possible, the subproblems solving steps among threads, see Section 4.3.2. The resulting parallel algorithm is so-named Inner P-SDP. On the other hand, the second paradigm performs simultaneous executions of the SDP algorithm, referred to as paths as above, so that a wider feasibility area is explored. Communication among paths is carried out several times per iteration targeting two effects: convergence when finding an incumbent solution and divergence towards new search directions. The corresponding algorithm is so-named Outer P-SDP, see Section 4.3.3. The parallel computing concepts that will be considered in the parallelization strategies are defined in Section 1.3. Notably, the task thread subgroup and coordinator task thread concepts will be used in order to allow an asynchronized execution.

4.3.2 Inner P-SDP parallelization strategy

The main parallelization activities of the Inner P-SDP algorithm are performed at:

- FtB scheme: Step 2 for solving the $|\mathcal{R}^t|$ subproblems for stage $t : 1 < t \leq |\mathcal{T}|$, and Step 3 for generating and appending the corresponding EFV-ℓ^z in defining constraints to the leaf nodes ℓ of each of the $|\mathcal{R}^{t-1}|$ subproblems for stage $t-1$,

Algorithm 4.1: Serial SDP

```
Step 0: (Initialization)
        Set $F^* := \infty$, $iter := 0$, $z := 1$.
        Set $\mathcal{Z}_\ell := \{1\}$, $Z'_\ell := \emptyset \, \forall \ell \in \mathcal{L}_r$, $r \in \mathcal{R}^t, t \in \mathcal{T} \setminus \{|\mathcal{T}|\}$.
Step 1: (FtB scheme: Solve subproblem (4.3) for stage $t = 1$)
        Set $iter := iter + 1$.
        Solve the subproblem for obtaining $F_1(\emptyset)$.
        Set $(\overline{x}^1_g, \overline{y}^1_g)_{g \in \mathcal{M}_1}$ to its optimal solution vector.
        Set $t := 2$.
Step 2: (Solve subproblem (4.3) rooted at node $r$, $\forall r \in \mathcal{R}^t$)
        Set $\ell := a(r)$.
        Solve the subproblem for obtaining $F_r(\overline{V}^z_{a(r)})$.
        Set $(\overline{x}^z_g, \overline{y}^z_g)_{g \in \mathcal{M}_r}$ to its optimal solution vector.
        Set $\pi^z_r$ as the dual vector of the linking constraint system $V^z_\ell = \overline{V}^z_\ell$.
        Compute $\mu^z_r$ (4.4).
        If $iter \equiv 0 \pmod{niterk}$ for $iter > 1$ then: Delete any non-active EFV-$\ell^z$
        defining constraint from subproblem (4.3) and reset $z = 0$ in set $\mathcal{Z}_\ell$ for $\ell \in \mathcal{L}_r$.
Step 3: (Generate and append the EFV-$\ell$ defining constraint to
        subproblem (4.3) rooted at node $r'$, $\forall \ell \in \mathcal{L}_{r'}$, $r' \in \mathcal{R}^{t-1}$)
        Compute $\mu^z_\ell$ (4.5) and $\pi^z_\ell$ (4.6).
        Append constraint $\lambda_\ell \geq \mu^z_\ell + \pi^z_\ell V_\ell$ to the subproblem.
Step 4: (Forward to next stage $t + 1$)
        If $t < |\mathcal{T}|$ then reset $t := t + 1$ and go to Step 2.
Step 5: (Compute solution value for original model (1.4) and
        check stopping criteria)
        $\overline{F}^{iter} = \sum_{g \in \mathcal{G}} w_g (a_g \overline{x}^z_g + c_g \overline{y}^z_g)$.
        If $\frac{|\overline{F}^{iter} - F^*|}{|F^*|} \leq \epsilon_1$ and $\frac{|\overline{F}^{iter} - \overline{F}^{iter-1}|}{|\overline{F}^{iter-1}|} \leq \epsilon_2$ then STOP.
        If $iter = 1$ or $\overline{F}^{iter} < F^*$ then $x^* := \overline{x}^z$, $y^* := \overline{y}^z$ and $F^* := \overline{F}^{iter}$.
        If $iter = miter$ then STOP.
Step 6: (BtF scheme: Compute dual vector of subproblem (4.3) rooted
        at node $r$ $\forall z \in \mathcal{Z}_{a(r)} : z > 0$, $r \in \mathcal{R}^t$, $t > 1$)
        Set $\ell := a(r)$.
        Solve the subproblem for obtaining $F_r(\overline{V}^z_\ell)$.
        Set $z' := |\mathcal{Z}_\ell| + z$ and enlarge $\mathcal{Z}'_\ell := \mathcal{Z}'_\ell \cup \{z'\}$.
        Set $\pi^{z'}_r$ as the dual vector of the linking constraint system $V^z_\ell = \overline{V}^z_\ell$.
        Compute $\mu^{z'}_r$ (4.4).
Step 7: (Generate and append the EFV-$\ell$ defining constraints in
        subproblem (4.3) rooted at node $r'$, $\forall \ell \in \mathcal{L}_{r'}$, $r' \in \mathcal{R}^{t-1}$)
        Compute $\mu^{z'}_\ell$ (4.5) and $\pi^{z'}_\ell$ (4.6) $\forall z' \in \mathcal{Z}'_\ell$.
        Append constraint $\lambda_\ell \geq \mu^{z'}_\ell + \pi^{z'}_\ell V_\ell \; \forall z' \in \mathcal{Z}'_\ell$ to the subproblem.
        Reset $\mathcal{Z}_\ell := \mathcal{Z}_\ell \cup \mathcal{Z}'_\ell$, $\mathcal{Z}'_\ell := 0$.
Step 8: (Backward to previous stage $t - 1$)
        Reset $t := t - 1$. If $t > 1$ then go to Step 6.
        Reset $z := |\mathcal{Z}_\ell| + 1$ and go to Step 1.
```

such that all subproblems rooted at node r for $r \in \mathcal{S}_\ell$ contribute to the EFV definition. It is worthy to point out that the serial algorithm SDP does not solve a subproblem of stage t for Step 2 (res. stage $t-1$ for Step 3) until all subproblems of the previous stage $t-1$ (res. successor stages) have been optimized, i.e. synchronized FtB execution. At the Inner P-SDP strategy there is no need to wait until all those subproblems in the previous stage are solved in Step 2 (res. successor stage in Step 3) to continue the execution of the FtB scheme, i.e. the execution can be asynchronized. In fact, the optimization of any subproblem must only wait until the optimization of its immediate ancestor subproblem is over for Step 2 (res. all successor ancestor subproblems for Step 3 are over) and the solutions are gathered.

- BtF scheme: Step 6 for solving the $|\mathcal{R}^t|$ subproblems for each of the $|\mathcal{Z}_{a(r)}|$ reference levels (if they are active) for stage $t : 1 < t \leq |\mathcal{T}|$, let r be the root node of any of those subproblems. And Step 7 for generating and appending the $|\mathcal{Z}_\ell^{'}|$ EFV-$\ell^{z'}$ curve defining constraints $\forall \ell \in \mathcal{L}_{r'}$ in the $|\mathcal{R}^{t-1}|$ subproblems, let r' be the root node of any of those subproblems, for the given stage t such that $t : 1 < t \leq |\mathcal{T}|$ and all subproblems rooted at node r for $r \in \mathcal{S}_\ell$ contribute to the EFV definition. Note that the BtF scheme can also be asynchronized for Step 6 and Step 7 since a thread can work on solving any subproblem, say, rooted at node r after solving all of its immediate successor subproblems. Then, Step 7 for generating and appending the EFV-λ_ℓ defining constraint in its immediate ancestor subproblem rooted by node, say r' (i.e., $\ell = a(r)$ for $\ell \in \mathcal{L}_{r'}$) waits until the subproblems rooted by all the successor nodes of node ℓ (i.e., nodes in set $\mathcal{S}_\ell$) are solved in Step 6. Once the EFV-λ_ℓ defining constraints for all leaf nodes ℓ (therefore, all nodes in set $\mathcal{L}_{r'}$) have been generated and appended to the subproblem rooted by node r', then it can be solved in Step 6.

- Communication and Synchronisation phase: Threads need to exchange information in order to reproduce the serial version behaviour. Firstly, $|\mathcal{R}^t|$ task thread subgroups will be declared, where $t = 2$, such that the scenario subtrees rooted at nodes r for $r \in \mathcal{R}^2$ will be assigned to the coordinator task threads; these threads manage the secondary communication environment. If more threads are available, they will work as subordinated task thread, i.e., each follows a coordinator task thread and splits the solving tasks. A tertiary communication layer handles the interaction between a coordinator task thread and its potential subordinated task threads. The frequency of the communication between the secondary and tertiary layers (i.e., the part of the algorithm that can be executed in an asynchronized way) depends on the model characteristics, i.e., the nature of the linking variables.

Note that once the stages are further and further forward in the FtB scheme, the number of usable threads is substantially increased until the last stage $|\mathcal{T}|$ where as many threads as the number of scenarios $|\Omega|$ can be used. Additionally, note that subproblems that belong to different stages and one is not ancestor of the other can also be optimized simultaneously in the asynchronized version of the Inner P-SDP algorithm.

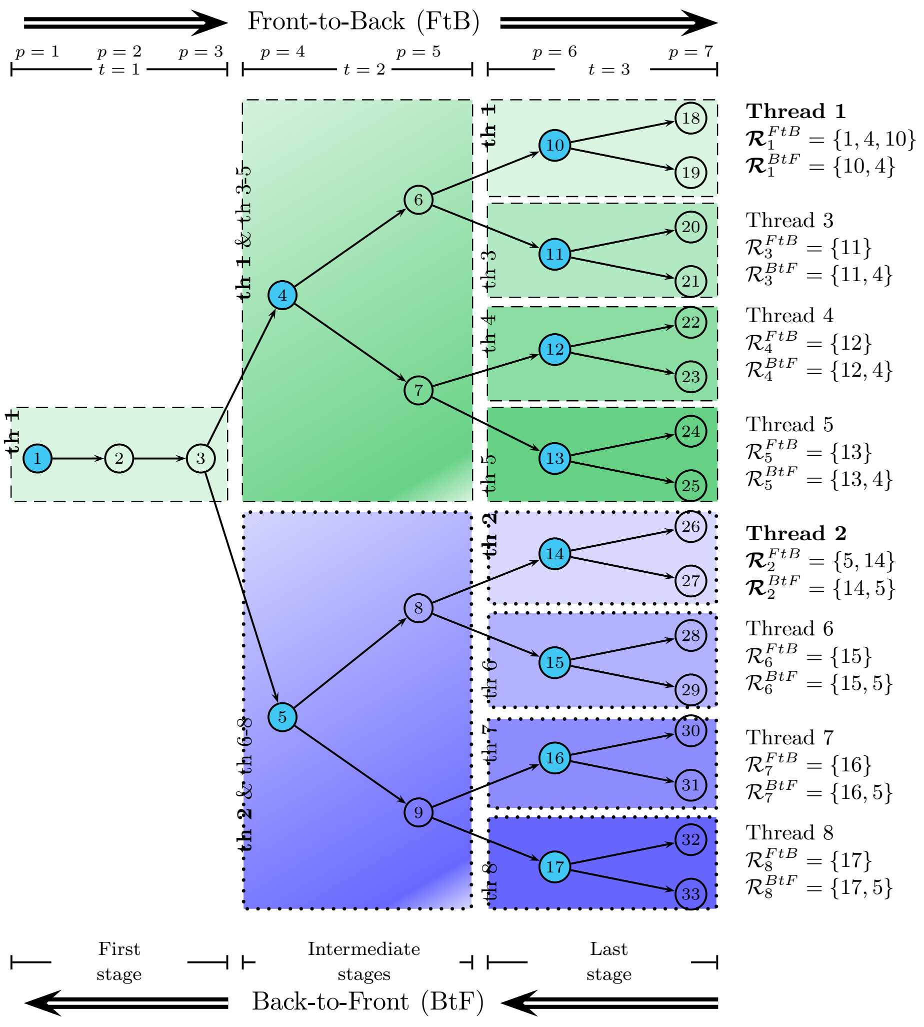
Front-to-Back (FtB)
$p=1$ $p=2$ $p=3$ $t=1$
$p=4$ $p=5$ $t=2$
$p=6$ $p=7$ $t=3$
th 1
th 1 & th 3-5
th 2 & th 6-8
th 1
th 2
th 3
th 4
th 5
th 6
th 7
th 8
Thread 1
$\mathcal{R}_1^{FtB} = \{1,4,10\}$
$\mathcal{R}_1^{BtF} = \{10,4\}$
Thread 3
$\mathcal{R}_3^{FtB} = \{11\}$
$\mathcal{R}_3^{BtF} = \{11,4\}$
Thread 4
$\mathcal{R}_4^{FtB} = \{12\}$
$\mathcal{R}_4^{BtF} = \{12,4\}$
Thread 5
$\mathcal{R}_5^{FtB} = \{13\}$
$\mathcal{R}_5^{BtF} = \{13,4\}$
Thread 2
$\mathcal{R}_2^{FtB} = \{5,14\}$
$\mathcal{R}_2^{BtF} = \{14,5\}$
Thread 6
$\mathcal{R}_6^{FtB} = \{15\}$
$\mathcal{R}_6^{BtF} = \{15,5\}$
Thread 7
$\mathcal{R}_7^{FtB} = \{16\}$
$\mathcal{R}_7^{BtF} = \{16,5\}$
Thread 8
$\mathcal{R}_8^{FtB} = \{17\}$
$\mathcal{R}_8^{BtF} = \{17,5\}$
First stage
Intermediate stages
Last stage
Back-to-Front (BtF)

Figure 4.3: Inner parallelization scheme with 8 task threads (2 coordinator and 6 subordinated)

In Figure 4.3, the inner parallelization strategy is illustrated for the scenario tree shown in Figure 4.1. Assume there are 8 available threads. As there are $|\mathcal{R}^2| = 2$

branches for the second stage, two coordinator task threads are considered (therefore two task thread subgroups), **th 1** in green and **th 2** in blue (threads 3 to 5 are subordinated task threads of coordinator **th 1** and threads 6 to 8 are subordinated task threads of coordinator **th 2**). The root nodes $\{r\}$ for the subproblems (4.3) that are solved in the FtB scheme are detailed in set $\mathcal{R}_{th}^{FtB}= \{r,\ r \in \mathcal{R}^t, t \in \mathcal{T}\}$ for coordinator task threads and set $\mathcal{R}_{th}^{BtF}= \{r,\ r \in \mathcal{R}^t,\ 2 < t \leq \mathcal{T}\}$ for subordinated task threads. The subproblems (4.3) solved in the BtF scheme are detailed in set $\mathcal{R}_{th}^{BtF} = \{r,\ r \in \mathcal{R}^t, t \in \mathcal{T} \setminus \{1\}\}$ for coordinator and subordinated task threads. Notice that the subproblem rooted at node $r = 1$ is iteratively solved only in the FtB scheme by the first coordinator task thread, **th 1**. For the FtB scheme in the example the coordinator task thread **th 1** solves the subproblems rooted at nodes 1, 4 and 10, while the subproblems rooted at nodes 11, 12 and 13 are solved by the subordinated task threads 3, 4 and 5, respectively. In the same way, the coordinator task thread **th 2** solves the subproblems rooted at nodes 5 and 14, while the subproblems rooted at nodes 15, 16 and 17 are solved by the subordinated task threads 6, 7 and 8, respectively. However, the subproblems from last stage $|\mathcal{T}|$ to stage 2 are distributed among all available task threads in the BtF scheme and observe that each subproblem is solved in intermediate stages $(2 \leq t \leq |\mathcal{T}| - 1)$ in an increasing number of computing time due to adding reference levels.

4.3.3 Outer P-SDP parallelization strategy

The Outer P-SDP paradigm is based on the path concept, a link between simultaneously SDP algorithm execution and main thread management. Notice that a single task thread is considered in each path. It uses the *solution pool* option offered by the state-of-the-art MIP engine for optimizing stage $t = 1$ subproblem alone, which allows not only one optimal solution of a problem but also multiple ones, referred as *alternative solutions*, to be stored. The scheme is as follows.

- Global solution pool: This phase aims to direct each path to a different starting search direction that consists of a solution drawn from the *solution pool.* It will be executed at the beginning of Step 1 for $iter = 1$. Every path executes the *solution pool* option and, thus, picks up an *alternative solution*, see below.
- Communication and synchronization: This phase is performed at the end of Step 5 in the FtB scheme, see Figure 4.4 and detail at Figure 4.5. The main threads gather the (feasible) solution obtained at each path and store the best among them as the global solution at the given iteration. If the stopping criteria is fulfilled then all main threads stop execution; otherwise it is checked if the global iteration solution value improves the global incumbent one, z^{DEM}. If it does, then z^{DEM} is updated and it is passed to all paths. Otherwise, each path checks if its current solution value improves its own incumbent solution

value, z_{path}^{DEM}. If it does not, a *warning* message of non-improved solution is stored. In any case, the execution will start the BtF scheme.

- Path perturbation seed at Step 6: In order to mantain the divergence between the different paths, each one has its proper seed for perturbing the current solution in order to generate reference levels.

- Alternative solution from path *pool*: It is executed at the beginning of Step 1 for $iter > 1$ and for the path where a *warning* message is issued at the previous iteration. A solution for stage $t = 1$ is chosen from the *pool* following a given selection criterion, which is problem depending, for instance, a solution whose value is the smallest one in the *pool*, etc.

4.4 Computational experience

The algorithm has been implemented in a `C` experimental code and uses the state-of-the-art CPLEX V12.5 optimization engine for solving the MIP subproblems (4.3). The computational experiments were conducted in the ARINA computational cluster at SGI/IZO-SGIker, Universidad del País Vasco, UPV/EHU, see Section 1.5. For the reported experiments, 8 Intel Xeon type computing nodes have been used, consisting of 12 cores with 48Gb of RAM, see Table 1.1.

4.4.1 Serial implementation overview

The SDP algorithm presented in Algorithm 4.1 has been implemented for solving the realistic production planning problem under uncertainty introduced in Cristobal *et al.* [2009]. As it frequently happens in tactical multistage planning problems, in our case only continuous variables in a period have nonzero coefficients in the constraints of the next one. Therefore the linking variables between stages only occur between the leaf nodes $\{\ell\}$ of the subproblems rooted at node, say, r' in a given stage $t-1$, and the root nodes r of the successor subproblems in the next stage, such that $r \in \mathcal{S}_\ell$, if any. (In our case, $V_\ell = \{y_g,\ g \in \tilde{\mathcal{A}}_\ell\}$ for $\ell \in \mathcal{L}_{r'},\ r' \in \mathcal{R}^{t-1}, t \in \mathcal{T}\backslash\{1\}$, such that $(y_g)_i$ denotes the stock of product i in set $\mathcal{I}$ related to scenario group g). Therefore, those continuous variables are the only ones to be iteratively perturbed. The perturbation has been performed as follows: $\xi = ran/f(iter)$, where $f(iter) = k$ being k a constant. Given the relatively small number of iterations performed and the large-scale of the instances, the algorithm cannot ensure the goodness of a unique searching direction, consequently, a constant factor k has been chosen in order to preserve a wide search. However, the value of parameter k changes depending on the value of the variable to be perturbed, such that small perturbations for small values and big perturbation for big values are generated, looking for a relative homogeneous change. Additionally, a single new reference level is generated at the

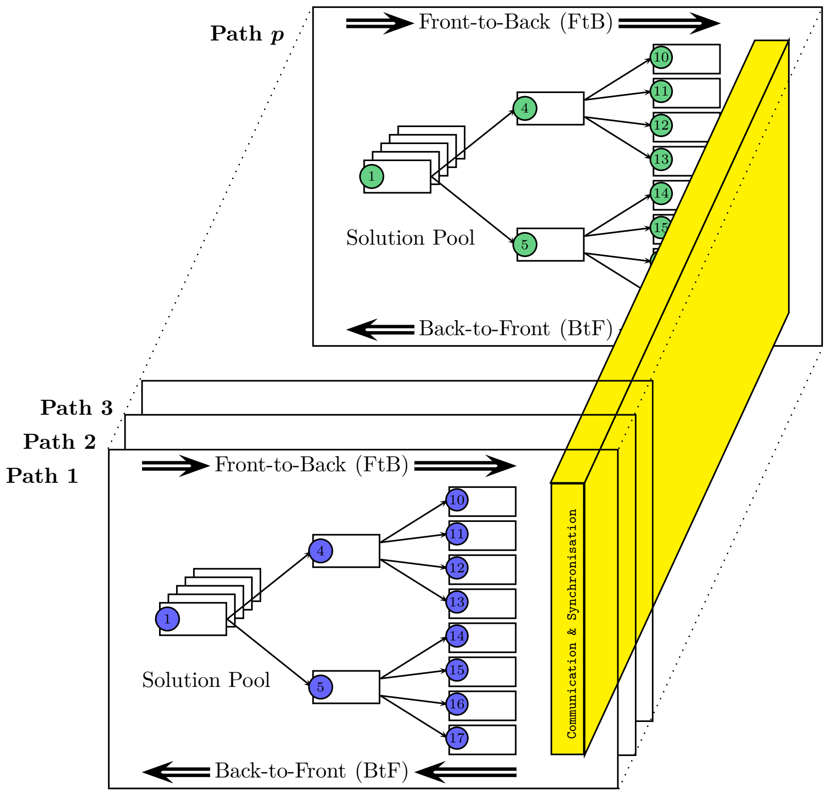

Figure 4.4: Outer P-SDP scheme for simultaneous execution of p paths

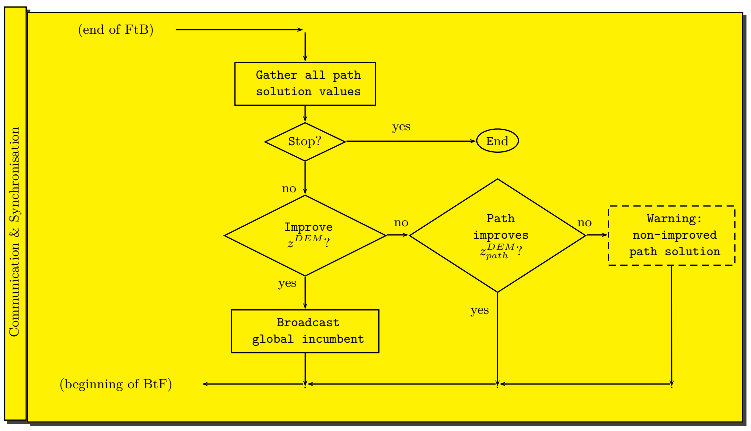
Communication & Synchronisation
(end of FtB)
Gather all path solution values
Stop?
yes
End
no
Improve z^{DEM}?
no
Path improves z^{DEM}_{path}?
no
Warning: non-improved path solution
yes
Broadcast global incumbent
yes
(beginning of BtF)

Figure 4.5: Detail of communication and synchronization phase for Outer P-SDP

BtF scheme. We observed in our computational experimentation with the type of problem under consideration that multiple perturbations of the same solution will lead to similar reference levels that, then, significantly increase the elapsed time. Finally, all reference levels are kept active.

Throughout the numerical experiments that are reported, the stopping criteria parameters are set up to $\epsilon_1 = 0.001$ and $\epsilon_2 = 0.0001$, $niterk = miter + 1$, $miter = 15$ iterations, time limit of 8 hours and 35 Gb of memory limit.

4.4.2 Inner parallel implementation overview

The Inner P-SDP maintains the algorithmic structure of the SDP algorithm; i.e. models, linking variables perturbation, reference level management and stopping criteria parameters. Note that if the time limit is reached at a SDP execution, the Inner P-SDP execution can be allowed continue to iterate and stop afterwards, see below.

As only the variables in a period have nonzero coefficients in the constraints of the following one, the implemented Inner P-SDP algorithm comprehends asynchronized execution parts. Thus, stage $t = 1$ is managed by $th1$, stage $t = 2$ by coordinator task threads and subsequent stages by both coordinator and subordinated task threads. Therefore, secondary communication is needed between stages 1 and 2; and tertiary communication will be performed between consecutive pair of stages from 2 to $|\mathcal{T}|$ whenever needed. Global synchronization is achieved when gathering the EFV curves at the end of FtB and BtF schemes. The procedure for Inner P-SDP mainly follows the structure of the SDP algorithm detailed in Algorithm 4.1 being Algorithm 4.2 its adaptation for the parallelization.

4.4.3 Outer parallel implementation overview

Let *inisize*, *inigap*, *nthread*, *pathsize* and *pathgap* be pilot case-driven parameters taken as input for the execution of the outer parallelization. The global solution pool generation phase stores up to *inisize* alternative solutions, chosing them from subproblem (4.3) solved at stage $t = 1$ for $iter = 1$ with the smallest optimality gap among those whose gap is lower than *inigap*. Then, the *nthread* most diverge solutions among them are chosen from the pool and assigned to threads. The selection criterion is as follows: The candidate solution in the pool with the highest euclidean distance of the vector of the values of the linking variables (at stage $t = 1$) with respect to the optimal solution is the first candidate; then the solution with the highest distance with respect to the optimal plus the distance with respect to the first candidate is the second candidate, and so on.

The alternative solution path selection phase (at stage $t = 1$ for $iter > 1$) stores

Algorithm 4.2: Inner P-SDP

```
Step 0: (Initialization)
Step 1: (FtB scheme: Solve subproblem (4.3) for stage $t = 1$)
        The subproblem is solved by coordinator task thread $th1$.
        Secondary communication: The coordinators gather the solution from
        $th1$. They then follow an asynchronized execution until the end of the
        FtB scheme, since no secondary communication is needed in-between.
        Set stage $t := 2$.
Step 2: (Solve subproblem (4.3) rooted at node $r$, $\forall r \in \mathcal{R}^t$)
        Each task thread solves its corresponding subproblems rooted at
        node $r \in \mathcal{R}_{th}^{FtB}$ and all task threads solve their own subproblems
        simultaneously. Note that for stage $t > 2$ all task threads
        are used, otherwise only the coordinator task threads are used.
Step 3: (Generate and append the EFV-$\ell$ defining constraint to
        subproblem (4.3) rooted at node $r'$, $\forall \ell \in \mathcal{L}_{r'}$, $r' \in \mathcal{R}^{t-1}$)
        Tertiary communication: Performed only for stage $t = 2$.
        Subordinated task threads gather the solution from their corresponding
        coordinator task thread so that they can follow an asynchronized
        execution until the end of the FtB scheme.
Step 4: (Forward to next stage $t + 1$)
        Synchronization: If stage $t = |\mathcal{T}|$ all task threads gather
        solution and EFV curves.
Step 5: (Compute solution value for original model (1.4) and check
        stopping criteria)
Step 6: (BtF scheme: Compute dual vector of subproblem (4.3) rooted
        at node $r$, $\forall r \in \mathcal{R}^t, t > 1$)
        The subproblem rooted at $r \in \mathcal{R}_{th}^{BtF}$ is solved by its corresponding thread.
        Finally, if stage $t = 2$ then the results of solving each
        subproblem rooted at $r \in \mathcal{R}_{th}^{BtF}$ by a coordinator task thread
        are shared with its corresponding subordinated task threads.
        Tertiary communication: Each coordinator and its subordinated task
        threads gather solution and follow a synchronized execution among
        them but their execution is asynchronized with respect to other
        task thread groups.
Step 7: (Generate and append the EFV-$\ell$ defining constraints in
        subproblem (4.3) rooted at node $r'$, $\forall \ell \in \mathcal{L}_{r'}$, $r' \in \mathcal{R}^{t-1}$)
        Synchronization: In case stage $t = 1$ all task threads gather
        solution and EFV curves.
Step 8: (Backward to previous stage $t - 1$)
```

up to *pathsize* candidates with an optimality gap lower than *pathgap* and takes the one with the highest above euclidean distance from the optimal solution for stage $t = 1$.

The stopping criteria parameters are set up to the same values used at the SDP and Inner P-SDP. The procedure for Outer P-SDP is described in Algorithm 4.3

Algorithm 4.3: Outer P-SDP

```
Step 0: (Initialization)
Step 1: (FtB scheme: Solve subproblem (4.3) for $t = 1$)
        Generate the global solution pool for iteration $iter = 1$ from the set
        of optimal and quasi-optimal solutions from solving the subproblem
        as described above.
        For $iter > 1$ and those paths that have issued a warning message on
        “non-improved path solution” at the end of the FtB scheme (Step 5)
        of the previous iteration, an alternative solution is picked up from
        the pool, according to the criterion presented above.
Step 2: (Solve subproblem (4.3) rooted at node $r$, $\forall r \in \mathcal{R}^t,\ t \geq 2$)
Step 3: (Generate and append the EFV-$\ell$ defining constraint to
        subproblem (4.3) rooted at node $r'$, $\forall \ell \in \mathcal{L}_{r'}$, $r' \in \mathcal{R}^{t-1}$)
Step 4: (Forward to next stage $t + 1$)
Step 5: (Compute solution value for original model (1.4) and check
        stopping criteria)
        The communication and synchronization phase is executed, see
        Subsection 4.3.3 and Figure 4.5. Compute global iteration solution
        and check stopping criteria. If global incumbent solution has been
        updated, all task threads gather the corresponding solution; if global
        incumbent solution and path incumbent solution have not been
        updated, the warning message “non-improved path solution” is
        issued for the appropiate path.
Step 6: (BtF scheme: Compute dual vector of subproblem (4.3)
        rooted at node $r$, $r \in \mathcal{R}^t, t > 1$)
        Path reference levels: Generate new reference levels using the path
        criteria for random values.
Step 7: (Generate and append the EFV-$\ell$ defining constraints in
        subproblem (4.3) rooted at node $r'$, $\forall \ell \in \mathcal{L}_{r'}$, $r' \in \mathcal{R}^{t-1}$)
Step 8: (Backward to previous stage $t - 1$)
```

4.4.4 Testbed problem: Production planning under uncertainty

The problem consists of deciding how much production and, how much loss in product demand can be expected at each period along a time horizon. Production capacity constraints, product stock limitations, some logistic constraints related to the production lot sizing, and product demand requirements should be satisfied at a minimum cost. There is a vast amount of literature on the deterministic version of the problem. See the seminal paper of Wagner and Whitin [1958] for considering only continuous variables. See Belvaux and Wolsey [2001]; Dillenberger *et al.* [1994]; Krarup and Bilde [1977]; Miller *et al.* [2000]; Pochet and Wolsey [1991]; Shapiro [1993]; Sousa and Wolsey [1992]; Wolsey [1997]; Zipkin [1986], among others, for considering lot sizing limitations and other logical constraints (and, then, considering 0-1 variables).

However, very frequently the production decisions must be made in the presence of uncertainty in several important parameters, such as production cost, product demand and resource availability along a multi-period time horizon, see Ahmed *et al.* [2003].

Production planning stochastic model

We present below a model for production planning taken from Alonso-Ayuso *et al.* [2007], where the uncertainty is treated via a scenario tree based scheme, such that the occurrence of the events is represented by a multi-period scenario tree. The following notation for the sets and parameters is used in the tactical production planning model.

Sets:

$\mathscr{J}$, set of products.

$\mathscr{R}$, set of resources.

Deterministic parameters:

$\hat{N}$, maximum number of products to be produced in a single time period.

$\underline{X}_{jt}$,$\overline{X}_j$, conditional minimum and maximum volume of product j that can be produced at time period t, respectively, if any, for $j \in \mathscr{J}$, $t \in \mathscr{T}$.

$\overline{S}_j$, maximum volume of product j that can be in stock at any time period, for $j \in \mathscr{J}$.

o_{rj}, unit capacity consumption of resource r by product j, for $r \in \mathscr{R}$, $j \in \mathscr{J}$.

h_j, unit holding cost of product j at any time period, for $j \in \mathscr{J}$.

p_j, unit lost demand penalty for product j, for $j \in \mathscr{J}$.

f_j, fixed cost to be incurred for producing product j at any time period, for $j \in \mathscr{J}$.

Uncertain parameters under scenario group g, for $g \in \mathscr{G}$:

O_r^g, available capacity of resource r at time period $t(g)$, for $r \in \mathscr{R}$.

D_j^g, demand of product j, for $j \in \mathscr{J}$.

c_j^g, unit processing cost of product, for $j \in \mathscr{J}$.

Variables under scenario group g for product j, for $g \in \mathscr{G}$, $j \in \mathscr{J}$:

δ_j^g, 0-1 variable such that its value is 1 if product j is produced under scenario group g, and 0 otherwise.

x_j^g, production volume of product j under scenario group g.

s_j^g, stock volume of product j under scenario group g.

y_j^g, lost demand of product j at time period $t(g)$ under scenario group g.

Determine the production and stock management policy to minimize the expected production and stock cost and the lost demand penalty plus the production fixed cost over the scenarios along the time horizon, subject to constraints (4.8)-(4.14).

$$\min \sum_{g \in \mathscr{G}} w^g \sum_{j \in \mathscr{J}} \left[c_j^g x_j^g + h_j s_j^g + p_j y_j^g + f_j \delta_j^g \right] \tag{4.7}$$

subject to

$$\sum_{j\in\mathscr{J}} o_{rj}x_j^g \le O_r^g \qquad \forall r\in\mathscr{R}, g\in\mathscr{G} \tag{4.8}$$

$$\underline{X}_{j,t(g)}\delta_j^g \le x_j^g \le \overline{X}_j\delta_j^g \qquad \forall j\in\mathscr{J}, g\in\mathscr{G} \tag{4.9}$$

$$\sum_{j\in\mathscr{J}} \delta_j^g \le \hat{N} \qquad \forall g\in\mathscr{G} \tag{4.10}$$

$$s_j^{a(g)} + x_j^g = D_j^g + s_j^g - y_j^g \qquad \forall j\in\mathscr{J}, g\in\mathscr{G} \tag{4.11}$$

$$0 \le s_j^g \le \overline{S}_j \qquad \forall j\in\mathscr{J}, g\in\mathscr{G} \tag{4.12}$$

$$y_j^g \ge 0 \qquad \forall j\in\mathscr{J}, g\in\mathscr{G} \tag{4.13}$$

$$\delta_j^g \in \{0,1\} \qquad \forall j\in\mathscr{J}, g\in\mathscr{G} \tag{4.14}$$

The knapsack constraints (4.8) ensure that the consumption of the resources does not exceed the capacity. Constraints (4.9) define the semi-continuous character of the production volume. The cover induced constraints (4.10) do not allow to produce more products in a single time period than the maximum allowed. Constraints (4.11) define the demand balance equations, such that the demand deficit is lost. Bounds (4.12) give the upper bounds of the product stock. For example, see in Nemhauser and Wolsey [1988] the deterministic version of the problem.

4.4.5 Testbed dimensions

Table 4.1 and Table 4.5 give the dimensions of the instances and the related models included in Testbed 3 and Testbed 4 that we here experiment with, respectively, for the realistic stochastic production planning problem under consideration. Testbed 3 includes the most difficult problems in Cristobal *et al.* [2009]. Testbed 4 includes new instances which are between two and ten times larger-scale than those from Testbed 3. The headings are as follows: *Case*, instance identification; $|J|$, number of products; $|R|$, number of resources; $|\mathcal{P}|$, number of periods; $|\mathcal{G}|$, number of nodes (i.e., scenario groups); and $|\Omega|$, number of scenarios; m, number of constraints; nx, number of 0-1 variables: ny, number of continuous variables; nel, number of nonzero elements in the constraint matrix; and $dens\%$, constraint matrix density, defined as $dens\% = 100\cdot\frac{nel}{n\times m}$, where n is the number of variables, $n = nx + ny$.

Tables 4.2 and 4.6 give the scenario tree structuring and the dimensions of the smallest and largest subproblems solved by the SDP algorithm for Testbed 3 and Testbed 4, respectively. The *Scenario tree* is a predefined structure of the multistage symmetric scenario tree, where each node is related to a period, such that the structure has the form $A_1^{B_1}A_2^{B_2}\dots A_{|\mathcal{T}|}^{B_{|\mathcal{T}|}}$, where, to start with, $\sum_{t\in\mathcal{T}} B_e$ is the number of periods in the time horizon, B_e denotes the number of consecutive periods in stage t and where the number of immediate successor nodes is the same at each node, and A_e is the number of these successor nodes for each node in $B_e, t \in \mathcal{T}$. For instance, the structure $1^6 2^3 3^2$ means the tree has 11 periods distributed in three stages (6, 3, and 2 periods for stage 1, 2 and 3, respectively), such that the first 6 periods have only one node each, the next three periods have two successor nodes for each node, and the last two periods have three successor nodes for each node. The headings are as follows: $|\mathcal{T}|$, smallest number of stages for obtaining a feasible solution in allowed time (8h) for SDP; *nprob*, number of subproblems in which the DEM is decomposed in total for the $|\mathcal{T}|$ stages; $\{|\mathcal{R}^t|\}_{t \in \mathcal{T}}$, set of number of subproblems for set of stages t; and for the smallest and largest subproblems: m, number of constraints; nx, number of 0-1 variables: ny, number of continuous variables.

Table 4.1: Problem dimensions. Testbed 3

Case	$\lvert J\rvert$	$\lvert R\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{G}\rvert$	$\lvert\Omega\rvert$	m	nx	ny	nel	*dens%*
c43	50	15	13	855	432	163080	42750	106650	892975	0.0037
c44	50	20	13	855	432	167355	42750	106650	1072525	0.0043
c45	100	20	13	660	432	234660	66000	154800	1635440	0.0032
c46	100	30	13	660	432	241260	66000	154800	2142320	0.0040
c47	100	20	13	855	432	316755	85500	213300	2134790	0.0023
c48	100	30	13	855	432	325305	85500	213300	2798270	0.0029
c49	10	2	14	1956	1296	71148	19560	45720	200966	0.0043
c50	10	4	14	1956	1296	75060	19560	45720	226394	0.0046
c51	10	2	14	2556	1296	96948	25560	63720	262898	0.0030
c52	10	4	14	2556	1296	102060	25560	63720	311462	0.0034
c53	10	2	16	10332	5184	392436	103320	258120	1115486	0.0008
c54	10	4	16	10332	5184	413100	103320	258120	1239470	0.0008
c55	10	4	16	11684	7776	448020	116840	272760	1340022	0.0008
c56	10	2	16	11684	7776	424652	116840	272760	1199814	0.0007
c57	100	20	14	1956	1296	693876	195600	457200	4775444	0.0011
c58	100	40	14	1956	1296	732996	195600	457200	7903088	0.0017
c59	100	20	14	2556	1296	946476	255600	637200	6411860	0.0008
c60	100	40	14	2556	1296	997596	255600	637200	10555136	0.0012
c61	100	20	16	10332	5184	3831372	1033200	2581200	26001944	0.0002
c62	100	30	16	10332	5184	3934692	1033200	2581200	34277876	0.0002
c63	100	20	16	11684	7776	4141364	1168400	2727600	28507632	0.0002
c64	100	30	16	11684	7776	4258204	1168400	2727600	38497452	0.0002

Tables 4.3 and 4.7 show the results obtained by plain use of CPLEX in the optimization of DEM (1.4) for Testbed 3 and Testbed 4, respectively. The headings are as follows: z_{LP} and t_{LP}, solution value of the LP relaxation of DEM and elapsed time (secs) for obtaining it; z^{DEM} and t^{DEM}, solution value of DEM and related elapsed time (secs); and $OG\%$, optimality gap achieved for the MIP problem. Note: CPLEX uses up to 8 threads, maximum of the academic version.

The CPLEX results reported in Cristobal *et al.* [2009] for Testbed 3 whose instances have the dimensions shown in Table 4.1 are such that the optimal solution is only obtained for instances $c49$ and $c50$. Table 4.3 shows that the optimal

Table 4.2: Smallest and largest subproblem dimensions. Testbed 3

	Scenario				Smallest subproblem			Largest subproblem		
Case	*tree*	$\|\mathcal{T}\|$	*nprob*	$\{\|\mathcal{R}^t\|\}_{t\in\mathcal{T}}$	*m*	*nx*	*ny*	*m*	*nx*	*ny*
c43	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	1296	300	900	2840	750	1850
c44	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	1326	300	900	2915	750	1850
c45	$1^6 2^4 3^3$	3	51	$\{1,2,48\}$	2526	600	1800	6315	1500	4500
c46	$1^6 2^4 3^3$	3	51	$\{1,2,48\}$	2586	600	1800	6465	1500	4500
c47	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	2526	600	1800	5515	1500	3700
c48	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	2586	600	1800	5665	1500	3700
c49	$1^6 2^4 3^4$	3	51	$\{1,2,48\}$	258	60	180	1450	400	930
c50	$1^6 2^4 3^4$	3	51	$\{1,2,48\}$	270	60	180	1530	400	930
c51	$1^6 3^4 2^4$	3	166	$\{1,3,162\}$	258	60	180	1720	400	1200
c52	$1^6 3^4 2^4$	3	166	$\{1,3,162\}$	270	60	180	1800	400	1200
c53	$1^6 3^4 2^6$	3	166	$\{1,3,162\}$	258	60	180	2389	630	1570
c54	$1^6 3^4 2^6$	3	166	$\{1,3,162\}$	270	60	180	2515	630	1570
c55	$1^6 2^5 3^5$	3	99	$\{1,2,96\}$	270	60	180	4635	1210	2820
c56	$1^6 2^5 3^5$	3	99	$\{1,2,96\}$	258	60	180	4393	1210	2820
c57	$1^6 2^4 3^4$	3	51	$\{1,2,48\}$	2526	600	1800	14140	4000	9300
c58	$1^6 2^4 3^4$	3	51	$\{1,2,48\}$	2646	600	1800	14940	4000	9300
c59	$1^6 3^4 2^4$	3	166	$\{1,3,162\}$	2526	600	1800	16840	4000	12000
c60	$1^6 3^4 2^4$	3	166	$\{1,3,162\}$	2646	600	1800	17640	4000	12000
c61	$1^6 3^4 2^6$	3	166	$\{1,3,162\}$	2526	600	1800	23323	6300	15700
c62	$1^6 3^4 2^6$	3	166	$\{1,3,162\}$	2586	600	1800	23953	6300	15700
c63	$1^6 2^5 3^5$	3	99	$\{1,2,96\}$	2526	600	1800	42841	12100	28200
c64	$1^6 2^5 3^5$	3	99	$\{1,2,96\}$	2586	600	1800	44051	12100	28200

solution has now also been obtained for instances *c*45, *c*46, *c*51, *c*52 and *c*56. This improvement is mainly due to better software/hardware platform (CPLEX version was updated from V9.1 to V12.5, among others). For the other instances, CPLEX stops in Cristobal *et al.* [2009] due to running out of memory. For the last four instances, *c*61 to *c*64 CPLEX V9.1 did not even find the LP optimal solution. Now, CPLEX V12.5 solves the LP problem for the four instances and their optimality gap for the original DEM (1.4) varies between 0.19% and 1.80%. It is worthy to point out the difficulty of the instances in Testbed 3; the big difference between the solution values z_{LP} and z^{DEM} gives an indication of the weakness of the MIP model.

4.4.6 Serial SDP and parallel SDP

Tables 4.4 and 4.8 show the results for the SDP and Inner P-SDP algorithms for Testbed 3 and Testbed 4. The headings for SDP and Inner P-SDP are as follows: z^{DEM}, solution value of DEM; $GG\%$, goodness gap of the incumbent solution, defined as $GG\% = 100 \cdot \frac{z^{DEM}_{serial} - z^{DEM}_{cpx}}{z^{DEM}_{cpx}}$, where cpx means plain use of CPLEX and $serial$ means SDP; $niter$, number of full iterations; nz, number of generated reference levels; and $nprob$, number of solved MIP subproblems. The aditional headings are as follows: t^{DEM}_{serial} and t^{DEM}_{inner}, related elapsed time (secs) required by SDP and Inner P-SDP for $niter$ iterations, respectively; the scalability parameters: S_{th}, speedup, defined as $S_{th} = \frac{t^{DEM}_{serial}}{t^{DEM}_{inner}}$ and $E_{th}\%$, efficiency, defined as $E_{th}\% = 100 \cdot \frac{S_{th}}{th}$, where th

Table 4.3: Plain use of CPLEX. Solution value and elapsed time. Testbed 3

Case	z_{LP}	t_{LP}	z^{DEM}	t^{DEM}	*OG%*
c43	93258	2	3498249[b]	16212	0.06
c44	87761	3	4211366[b]	24252	0.03
c45	196912	4	8036004	37	$*$
c46	187742	4	8087808	375	$*$
c47	192924	4	7151251[b]	9421	0.09
c48	176461	5	6594167[b]	9007	0.12
c49	14094	1	993334	1	$*$
c50	16666	1	1005119	2	$*$
c51	18402	1	772567	16	$*$
c52	20511	1	862754	20	$*$
c53	19224	4	670234[b]	15091	0.32
c54	21403	3	769236[b]	16938	0.19
c55	21940	3	1163290[b]	8948	0.07
c56	23775	3	1126270	43	$*$
c57	190641	10	7174215[b]	10064	0.06
c58	179749	15	8753936[b]	23991	0.02
c59	183144	14	8200795[b]	15974	0.08
c60	186600	20	8582624[b]	17757	0.13
c61	219479	25	9407495[b]	3432	0.19
c62	216297	29	8685992[b]	3422	1.61
c63	241836	24	7913950[b]	6555	1.11
c64	235090	77	7514260[b]	5926	1.80

Note: solved with CPLEX v12.5 using 8 threads
[b] Stop. Out of memory (35Gb)
$*$ Default optimality gap achieved: 10^{-4}%

is the total number of threads used by Inner P-SDP.

Note that *GG%* can be negative indicating that the related decomposition algorithm obtains a better solution value than plain use of CPLEX in DEM within the time allowed. It is worthy to point out the limit on the elapsed time and memory availability. The results reported for Testbed 3 and Testbed 4 by using SDP and P-SDP consider a single reference level per subproblem and iteration, as stated above. The addition of more reference levels, see Step 6 in the general SDP scheme presented in Algorithm 4.1, implies higher elapsed time and the improvement is not remarkable for those instances of the production planning problem under uncertainty which we have experimented with. Only one auxiliary thread per task thread has been used in CPLEX to optimize the MIP subproblems (4.3) whose results have been reported in the tables. The elapsed time is smaller than the one for the platform where several auxiliary threads are used in almost all instances of the testbeds, due to the small dimensions of the subproblems.

The heading *niter* shows the number of full iterations at which SDP satisfies any of the stopping criteria (observe that symbol [a] means the stopping has been due to time limit exceeded). The results of Inner P-SDP correspond to the same number of iterations *niter* as SDP (and, then, both strategies obtain the same solution value of the original problem (4.7)-(4.14)). In this way, the elapsed time for both strategies can be fairly compared in Tables 4.4 and 4.8. It is also worth to point out that the interpretation of the scalability parameters S_{12} and $E_{12}\%$ is more accurate. See results in Table 4.8 in blue for the instances where Inner P-SDP is stopped by its own criteria, independently of the stopping reasons for SDP.

However, the aim of the outer parallelization consists of (exclusively) improving the solution value over the one obtained by the serial or inner parallelization versions. Notice that Inner P-SDP requires smaller elapsed time than SDP. On the contrary, since Outer P-SDP is looking for a better solution value it may require more elapsed time than any of the other two strategies, since it may perform more iterations or greater excess time may be required for handling the pool, see below.

Table 4.4: SDP and Inner P-SDP. Solution value and elapsed time. Testbed 3

		SDP *Algorithmic behaviour*					SDP	P-SDP	*Scalability*	
Case	*Scenario tree*	z^{DEM}	*GG%*	*niter*	*nz*	*nprob*	t^{DEM}_{serial}	t^{DEM}_{inner}	S_{12}	$E_{12}\%$
c43	$1^6 3^3 2^4$	3539594	1.18	5	9	566	33	5	6.60	55.00
c44	$1^6 3^3 2^4$	4249979	0.92	15	29	2256	145	19	7.63	63.60
c45	$1^6 2^4 3^3$	8123167	1.08	4	7	372	28	5	5.60	46.67
c46	$1^6 2^4 3^3$	8139108	0.63	5	9	487	51	8	6.38	53.13
c47	$1^6 3^3 2^4$	7227178	1.06	15	29	2256	433	58	7.47	62.21
c48	$1^6 3^3 2^4$	6660629	1.01	6	11	708	108	16	6.75	56.25
c49	$1^6 2^4 3^4$	1004692	1.14	15	29	1857	23	3	7.67	63.89
c50	$1^6 2^4 3^4$	1006477	0.14	3	5	261	4	1	6.67	55.56
c51	$1^6 3^4 2^4$	775933	0.44	4	7	1186	14	3	4.67	38.89
c52	$1^6 3^4 2^4$	870090	0.85	5	9	1538	18	3	6.00	50.00
c53	$1^6 3^4 2^6$	685613	2.29	4	7	1186	59	7	8.43	70.24
c54	$1^6 3^4 2^6$	776389	0.93	15	29	5806	486	51	9.53	79.41
c55	$1^6 2^5 3^5$	1165132	0.16	5	9	919	73	9	8.11	67.59
c56	$1^6 2^5 3^5$	1128968	0.24	3	5	501	20	3	8.33	69.44
c57	$1^6 2^4 3^4$	7256183	1.14	4	7	372	91	13	7.00	58.33
c58	$1^6 2^4 3^4$	8803020	0.56	7	13	729	200	26	7.69	64.10
c59	$1^6 3^4 2^4$	8251017	0.61	3	5	840	171	30	5.70	47.50
c60	$1^6 3^4 2^4$	8640233	0.67	4	7	1186	473	106	4.46	37.19
c61	$1^6 3^4 2^6$	9442635	0.37	4	7	1186	1220	224	5.45	45.39
c62	$1^6 3^4 2^6$	8630797[a]	-0.64	14	27	4976	27006	4505	5.99	49.96
c63	$1^6 2^5 3^5$	7938162	0.31	5	9	919	2237	309	7.24	60.33
c64	$1^6 2^5 3^5$	7484868	-0.39	15	29	3249	22167	3442	6.44	53.67

P-SDP under 12 threads. Subproblems solved by CPLEX V12.5 with 1 thread in both SDP and P-SDP
[a] Time limit exceeded (8h)

It is worthy to point out that the solution values reported in Table 4.4 are smaller (remember that our problem is a minimization one) than those reported in Cristobal *et al.* [2009] for all instances in Testbed 3 but c43, c47, c54, c55 and c56, due to some refinements in common features introduced in SDP and the differences in the random numbers for obtaining reference levels. Moreover, the largest instances c49 to c64 already solved in Cristobal *et al.* [2009] with five stages, now can be solved with three stages and, it happens that the solution values are smaller. The goodness gap varies between -0.64% and 2.29% with an average of 0.67%, in spite of the large-scale instances in the testbed whose dimensions are up to $m = 4.2$ M constraints, $nx = 1.1$ M 0-1 variables and $ny = 2.7$ M continuous variables. On the other hand, SDP also obtains the incumbent solution in relatively small elapsed time confirming the validity of the SDP approaches.

Observe in Table 4.4 that Inner P-SDP obtains the same solution value with intentionally the same number of iterations as SDP (as we pointed out above) but faster. Additionally, notice that the execution of all instances finishes before reaching the time limit, except for instance c62. In any case, its performance can also be measured by the scalability (speedup and efficiency) parameters; observe that for Inner P-SDP the speedup average is 6.81 and varies between 4.46 an 9.53 (therefore, elapsed time is between 4 and 10 times smaller than for the serial version, i.e., up to one order of magnitude time reduction) and efficiency varies between 37.19% and 79.41% with average is 56.74%. Note: One computing node with 12 threads has been used for P-SDP.

Table 4.5: Problem dimensions. Testbed 4

Case	$\vert J\vert$	$\vert R\vert$	$\vert\mathcal{P}\vert$	$\vert\mathcal{G}\vert$	$\vert\Omega\vert$	m	nx	ny	nel	*dens%*
c65	200	40	13	855	432	632655	171000	426600	7039780	0.0019
c66	200	100	13	660	432	508260	132000	309600	11756500	0.0052
c67	300	80	13	855	432	965655	256500	639900	18789045	0.0022
c68	300	140	14	1956	1296	2234196	586800	1371600	70936884	0.0016
c69	400	180	13	855	432	1349955	342000	853200	52443410	0.0033
c70	400	250	13	660	432	1048860	264000	619200	54941540	0.0059
c71	500	200	13	855	432	1665855	427500	1066500	72404095	0.0029
c72	500	300	14	1316	864	2596116	658000	1542000	163900016	0.0029
c73	600	200	14	2556	1296	5870556	1533600	3823200	258504420	0.0008
c74	600	400	13	660	432	1589460	396000	928800	130359000	0.0062
c75	700	250	13	855	432	2306205	598500	1493100	125338565	0.0026
c76	700	350	14	1316	864	3541916	921200	2158800	265882428	0.0024
c77	800	300	13	855	432	2647755	684000	1706400	170273800	0.0027
c78	800	450	16	11684	7776	36437484	9347200	21820800	3443801640	0.0003
c79	900	400	13	855	432	3032055	769500	1919700	253365120	0.0031
c80	900	500	13	660	432	2317860	594000	1393200	242906700	0.0053
c81	1000	300	13	855	432	3245355	855000	2133000	213003845	0.0022
c82	1000	400	13	1827	972	7068627	1827000	4509000	600774992	0.0013
c83	1000	550	14	1956	1296	7605756	1956000	4572000	875756624	0.0018
c84	1000	350	14	2556	1296	9825156	2556000	6372000	738204812	0.0008
c85	1000	250	16	15435	7776	57838185	15435000	38529000	3228837695	0.0001
c86	1000	600	16	11684	7776	45982084	11684000	27276000	1417735964	0.0001
c87	250	100	13	2280	1152	2222280	570000	1422000	50472950	0.0011
c88	450	200	11	1452	972	2468052	653400	1522800	110580270	0.0021
c89	650	300	11	2178	1152	5569578	1415700	3498300	351951946	0.0013
c90	900	600	10	1890	1296	6773490	1701000	3936600	830320920	0.0022

The dimensions of the instances and their models for Testbed 4 are much larger than those for Testbed 3; compare Tables 4.5 and 4.1, and 4.6 and 4.2. Note the very

Table 4.6: Smallest and largest subproblem dimensions. Testbed 4

	Scenario				Smallest subproblem			Largest subproblem		
Case	*tree*	$\|\mathcal{T}\|$	*nprob*	$\{\|\mathcal{R}^t\|\}_{t\in\mathcal{T}}$	m	nx	ny	m	nx	ny
c65	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	5046	1200	3600	10815	3000	7200
c66	$1^6 2^4 3^3$	3	51	$\{1,2,48\}$	5406	1200	3600	13515	3000	9000
c67	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	7686	1800	5400	16815	4500	11100
c68	$1^6 2^4 3^4$	3	51	$\{1,2,48\}$	8046	1800	5400	45540	12000	27900
c69	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	10686	2400	7200	23515	6000	14800
c70	$1^6 2^4 3^3$	3	51	$\{1,2,48\}$	11106	2400	7200	27765	6000	18000
c71	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	13206	3000	9000	29015	7500	18500
c72	$1^6 2^5 3^3$	3	99	$\{1,2,96\}$	13806	3000	9000	71331	15500	46500
c73	$1^6 3^4 2^4$	3	166	$\{1,3,162\}$	15606	3600	10800	104040	24000	72000
c74	$1^6 2^4 3^3$	3	51	$\{1,2,48\}$	16806	3600	10800	42015	9000	27000
c75	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	18306	4200	12600	40165	10500	25900
c76	$1^6 2^5 3^3$	3	99	$\{1,2,96\}$	18906	4200	12600	97681	21700	65100
c77	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	21006	4800	14400	46115	12000	29600
c78	$1^6 2^2 2^3 3^2 3^3$	5	971	$\{1,2,8,96,864\}$	10953	2400	7200	40263	10400	24000
c79	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	24006	5400	16200	52815	13500	33300
c80	$1^6 2^4 3^3$	3	51	$\{1,2,48\}$	24606	5400	16200	61515	13500	40500
c81	$1^6 3^3 2^4$	3	58	$\{1,3,54\}$	25806	6000	18000	55515	15000	36000
c82	$1^6 3^2 3^3 2^2$	4	517	$\{1,3,27,486\}$	11203	3000	7000	57213	13000	39000
c83	$1^6 2^4 3^4$	3	51	$\{1,2,48\}$	27306	6000	18000	155040	40000	93000
c84	$1^6 3^2 3^2 2^4$	4	193	$\{1,3,27,162\}$	17404	4000	12000	62265	15000	42000
c85	$1^6 3^2 3^3 2^5$	4	517	$\{1,3,27,486\}$	17004	4000	12000	128781	31000	90000
c86	$1^6 2^2 2^3 3^2 3^3$	5	971	$\{1,2,8,96,864\}$	13803	3000	9000	57813	13000	37000
c87	$1^6 6^2 2^5$	3	79	$\{1,6,72\}$	6606	1500	4500	30131	7750	19250
c88	$1^6 6^2 3^3$	3	115	$\{1,6,108\}$	12006	2700	8100	21963	5850	13500
c89	$1^6 12^2 2^3$	3	301	$\{1,12,288\}$	17406	3900	11700	37713	8450	25350
c90	$1^6 12^2 3^2$	3	445	$\{1,12,432\}$	14104	3600	8100	54613	11700	35100

large dimensions of the instances, the largest one has $m = 57.8$ million constraints, $nx = 15.4$ million 0-1 variables and $ny = 38.5$ million continuous variables.

Testbed 4 has been experimented with for assessing the performance of the decomposition algorithms SDP and P-SDP for instances where CPLEX cannot provide/prove the optimal solution due to either reaching time allowed (8h) or running out of memory (35Gb), see Table 4.7. Plain use of CPLEX V12.5 cannot even obtain the LP solution value in 9 out of the 13 last (and largest) instances in the testbed. The DEMs of the whole testbed are very weak, see in Table 4.7 the big difference between z_{LP} and z^{DEM} for those instances where plain use of CPLEX can provide a solution; it shows the high difficulty of problem solving.

Observe in Table 4.8 that the elapsed time limit (8h) is sometimes reached before the other stopping criteria are satisfied but no memory restriction occurs for the SDP and Inner P-SDP algorithms. Remember that $niter = 1$ means that only one feasible solution has been obtained. In comparison with plain use of CPLEX, the instances $c79$, $c80$, $c81$ and $c83$ can now be solved by the decomposition algorithms, and the elapsed time for the Inner P-SDP is remarkable. For five of the largest instances, namely, c78, c82, c84, c85 and c86, three stages are not enough for obtaining a

Table 4.7: Plain use of CPLEX. Solution value and elapsed time. Testbed 4

Case	z_{LP}	t_{LP}	z^{DEM}	t^{DEM}	$OG\%$
c65	346937	13	14663331[b]	13721	0.10
c66	363024	19	15219357[a]	8h	0.00
c67	595663	34	18610023[b]	14740	0.15
c68	626680	137	22958504[b]	20487	0.07
c69	757870	101	28992651[b]	16139	0.16
c70	766174	102	28295702[b]	14223	0.02
c71	950456	166	35163637[b]	16322	0.15
c72	1008738	172	34942146[b]	22726	0.07
c73	1254355	885	44974691[b]	4537	1.00
c74	1130106	302	43052882[b]	15307	0.05
c75	1318378	584	51159243[b]	13039	0.15
c76	1428993	538	54165785[b]	20446	0.11
c77	4333228	8295	56382236[b]	15777	0.80
c78	–[b]	15968	–[b]	15968	–
c79	–[a]	8h	–[a]	8h	–
c80	15213652	26160	64781037[a]	8h	0.08
c81	–[a]	8h	–[a]	8h	–
c82	–[b]	2626	–[b]	2626	–
c83	–[b]	1601	–[b]	1601	–
c84	–[b]	2227	–[b]	2227	–
c85	–[b]	3003	–[b]	3003	–
c86	–[b]	4546	–[b]	4546	–
c87	506035	95	16695042[b]	9849	1.16
c88	742359	226	27472791[a]	8h	0.12
c89	1033449	1446	38932000[b]	8274	0.78
c90	–[a]	8h	–[a]	8h	–

Note: solved with CPLEX v12.5 using 8 threads
[a] Time limit exceeded (8h)
[b] Stop. Out of memory (35Gb)

solution value by using SDP, due to exceeding the time allowed. Four stages are needed for instances $c82$, $c84$ and $c85$. Using five stages, all the instances can be solved (i.e., the running of the algorithm have only been stopped by the satisfaction of any of the algorithmic stopping criteria), including the largest ones $c78$ and $c86$. Note that the smaller the number of stages, the better feasible solution that can be obtained. The goodness gap with respect to plain use of CPLEX varies between 0.23% and 1.27% with average 0.79% (and improving the CPLEX solution value of instance c87 in $GG\% = -0.19$). The scalability parameters are that the speedup varies between 2.27 and 10.70 (average is 5.89) and the efficiency varies between 18.93% and 89.19% (average is 49.10%). Observe that SDP usually requires smaller elapsed time than plain use of CPLEX (for the instances where it can provide a solution).

Table 4.8 shows in blue the results obtained by Inner P-SDP for some of the instances, in particular the largest ones c85 and c86 in Testbed 4, where more iterations are performed than with SDP, which was stopped because of time

Table 4.8: SDP and Inner P-SDP. Solution value and elapsed time. Testbed 4

		SDP *Algorithmic behaviour*					SDP	P-SDP	*scalability*	
Case	*Scenario tree*	z^{DEM}	*GG%*	*niter*	*nz*	*nprob*	t^{DEM}_{serial}	t^{DEM}_{inner}	S_{12}	$E_{12}\%$
c65	$1^6 3^3 2^4$	14743592	0.55	5	9	566	199	28	7.11	59.23
c66	$1^6 2^4 3^3$	15314501	0.63	6	11	606	271	44	6.16	51.33
c67	$1^6 3^3 2^4$	18794916	0.99	5	9	566	589	97	6.07	50.60
c68	$1^6 2^4 3^4$	23158260	0.87	3	5	261	713	111	6.42	53.53
c69	$1^6 3^3 2^4$	29231009	0.82	4	7	430	735	122	6.02	50.20
c70	$1^6 2^4 3^3$	28580107	1.01	5	9	487	1490	278	5.36	44.66
c71	$1^6 3^3 2^4$	35500155	0.96	5	9	566	2280	370	6.16	51.35
c72	$1^6 2^5 3^3$	35386874	1.27	3	5	501	7740	3051	2.54	21.14
c73	$1^6 3^4 2^4$	45151032[a]	0.39	1	1	166	480	55	8.73	72.73
c74	$1^6 2^4 3^3$	43412879	0.84	5	9	487	3706	634	5.85	48.71
c75	$1^6 3^3 2^4$	51689710	1.04	5	9	566	3612	595	6.07	50.59
c76	$1^6 2^5 3^3$	54634660[a]	0.87	2	3	298	16891	7436	2.27	18.93
c77	$1^6 3^3 2^4$	56511745	0.23	3	5	300	2364	415	5.70	47.47
c78	$1^6 2^2 2^3 3^2 3^3$	68678909[a]	–	2	3	3018	26437	3388	7.8	65.03
c79	$1^6 3^3 2^4$	61360087	–	6	11	708	14473	2850	5.08	42.32
c80	$1^6 2^4 3^3$	65211203	0.66	5	9	487	10442	1994	5.24	43.64
c81	$1^6 3^3 2^4$	68722477	–	4	7	430	9351	2081	4.49	37.45
c82	$1^6 3^2 3^3 2^2$	69093005[a]	–	4	7	3886	22837	3777	6.05	50.39
c83	$1^6 2^4 3^4$	72865478[a]	–	3	5	261	21988	8181	2.69	22.40
c84	$1^6 3^2 3^2 2^4$	73706161	–	4	7	1618	10645	1787	5.96	49.64
c85	$1^6 3^2 3^2 2^5$	83305369[a]	–	1	1	517	26180	2446	10.7	89.19
		82942217[a]	–	3	5	2703	–	16283	($IG^{inner}\%$ 0.44)	
c86	$1^6 2^2 2^3 3^2 3^3$	87573333[a]	–	1	1	971	18201	2537	7.17	59.79
		87015342[a]	–	6	11	13326	–	26170	($IG^{inner}\%$ 0.64)	
c87	$1^6 6^2 2^5$	16663432	-0.19	5	9	803	1151	129	8.92	74.35
c88	$1^6 6^2 3^3$	27685724	0.78	5	9	1127	2078	259	8.02	66.86
c89	$1^6 12^2 2^3$	38995640	0.16	5	9	2897	7220	978	7.38	61.52
c90	$1^6 12^2 3^2$	52246689	–	5	9	4193	16574	2266	7.31	60.95

P-SDP under 12 threads. Subproblems solved by CPLEX v12.5 with 1 thread in both SDP and P-SDP
[a] Time limit exceeded (8h)

limitation. Therefore, the execution of Inner P-SDP for obtaining the new results is stopped when any of its own stopping criteria is satisfied, in this case also time limitation. The table then shows the comparison of the solution values obtained by using both strategies SDP and Inner P-SDP. The new heading is as follows: $IG^{inner}\%$, improvement gap of the solution value obtained by Inner P-SDP with respect to SDP, defined as $IG^{inner}\% = 100 \cdot \frac{z^{DEM}_{serial} - z^{DEM}_{inner}}{z^{DEM}_{inner}}$. It can be observed that the parallel algorithm obtains a better feasible solution than the serial one for those two instances. Therefore, the inner parallelization can obtain not only faster but also better solutions than the serial version.

Table 4.9 shows the results of the Outer P-SDP versus the SDP for the instances for which the improvement gap is greater than 0.01%. The additional headings are: *npool*, number of iterations in which one path, at least, picks up a solution for subproblem in stage $t = 1$ from the solution pool for Outer P-SDP; $IG^{outer}\%$, improvement gap of the solution value obtained by Outer P-SDP with respect to

SDP, defined as $IG^{outer}\% = 100 \cdot \frac{z^{DEM}_{serial} - z^{DEM}_{outer}}{z^{DEM}_{outer}}$; and Δz^{DEM}, absolute difference between the incumbent values obtained by Outer P-SDP and SDP.

Table 4.9: SDP and Outer P-SDP. Comparing solution value and elapsed time

	SDP			Outer P-SDP								
Case	z_{DEM}	t^{DEM}	*niter*	z^{DEM}	t^{DEM}	*niter*	*npool*	*nz*	*nprob*	*GG*%	$IG^{outer}\%$	Δz^{DEM}
c43	3539594	33	5	3538641	19	3	1	5	300	1.15	0.03	953
c45	8123167	28	4	8099878	237	12	9	23	1404	0.79	0.29	23289
c47	7227178	433	15	7225961	65	4	3	7	430	1.04	0.02	1217
c48	6660629	108	6	6659199	82	4	2	7	430	0.99	0.02	1430
c51	775933	14	4	775566	17	4	2	7	1186	0.39	0.05	367
c52	870090	18	5	869150	20	5	2	9	1538	0.74	0.11	940
c53	685613	59	4	675583	592	15	13	29	5388	0.80	1.48	10030
c54	776389	486	15	775311	305	8	4	15	2630	0.79	0.14	1078
c58	8803020	200	7	8801307	236	6	4	11	1857	0.54	0.02	1713
c59	8251017	171	3	8246290	1328	5	3	9	840	0.55	0.06	4727
c60	8640233	473	4	8638060	1026	4	1	7	840	0.65	0.03	2173
c61	9442635	1220	4	9427891	8867	4	1	7	1186	0.22	0.16	14744
c63	7938162	2237	5	7918829	6444	5	1	9	919	0.06	0.24	19333
c64	7484868	22167	15	7472057[a]	27385	7	5	13	1353	-0.56	0.17	12811
c65	14743592	199	5	14735966	175	4	2	7	430	0.50	0.05	7626
c66	15314501	271	6	15310143	304	5	3	9	487	0.60	0.03	4358
c67	18794916	589	5	18783416	1003	5	1	10	566	0.93	0.06	11500
c68	23158260	713	3	23136233	1187	3	1	5	261	0.77	0.10	22027
c70	28580107	1490	5	28560443	2918	6	3	11	606	0.94	0.07	19664
c71	35500155	2280	5	35486229	3258	5	3	9	566	0.92	0.04	13926
c74	43412879	3706	5	43388332	3540	4	2	7	372	0.78	0.06	24547
c77	56511745	2364	3	56499091	9038	5	1	9	430	0.21	0.02	12654
c80	65211203	10442	5	65179230	20455	6	3	11	606	0.61	0.05	31973
c81	68722477	9351	4	68686527	21739	4	1	7	430	–	0.05	35950
c82	69093005	22837	4	69034837[a]	19572	3	1	5	2703	–	0.08	58168
c87	16663432	1151	5	16655454	1999	4	2	7	604	-0.24	0.05	7978
c88	27685724	2078	5	27675741	2044	4	1	7	856	0.74	0.04	9983

P-SDP under 12 threads. Subproblems solved by CPLEX v12.5 with 1 thread in both SDP and P-SDP
[a] Time limit exceeded (8h)

We can observe in Table 4.9 that in most of the instances Outer P-SDP requires more elapsed time than SDP (and the time is one order of magnitude higher than the time reported in Tables 4.4 and 4.8 for Inner P-SDP). However, the incumbent value obtained by Outer P-SDP is smaller than the value obtained by SDP and Inner P-SDP. In relative terms ($IG^{outer}\%$) the difference is not very high, but in absolute terms (Δz^{DEM}) the difference can be significative. See in Section 5.7 some concluding remarks about the applicability of the outer parallelization for other types of algorithms.

Table 4.10 and Figure 4.6 show the results in terms of elapsed time, speedup and efficiency for instances c85 and c86 when using 12, 24, 48 and 96 threads for Inner P-SDP versus SDP (execution using only one auxiliary thread for CPLEX). The new headings are as follows: S^{top}_{th}, *top speedup* defined as $S^{top}_{th} = \frac{t^{DEM}_{serial}}{t^{top}_{th}}$ and $E^{top}_{th}\%$, *top efficiency* defined as $E^{top}_{th}\% = 100 \cdot \frac{S^{top}_{th}}{th}$, where $t^{top}_{th} = \sum_{t \in \mathcal{T}} \frac{t^{serial}_{e}}{\min(th, |\mathcal{R}^t||\mathcal{Z}|)}$ and t^{serial}_{e} is the elapsed time for stage t in SDP such that $t^{DEM}_{serial} = \sum_{t \in \mathcal{T}} t^{serial}_{e}$.

The concepts S_{th}^{top} and E_{th}^{top} consider the top speedup and top efficiency that could be achieved, respectively, given the specifications of the SDP algorithm and its Inner P-SDP parallelization under ideal conditions (as if time is not lost by communication and synchronisation). For example, in instance c85 $t_{serial}^{DEM} = 26180 = 59 + 10 + 8987 + 17125 = t_1 + t_2 + t_3 + t_4$ and $\{\mathcal{R}^t\} = \{1, 3, 27, 486\}$; therefore, with 48 threads, $t_{48}^{top} = \frac{59}{1} + \frac{10}{3} + \frac{8987}{27} + \frac{17125}{48} = 752$, $S_{48}^{top} = \frac{26180}{752} = 34.81$ and $E_{48}^{top} = 100 \cdot \frac{34.81}{48} = 72.53\%$.

Table 4.10: Inner P-SDP scalability for instances c85 and c86

	c85					c86				
th	t^{DEM}	S_{th}	$E_{th}\%$	S_{th}^{top}	$E_{th}^{top}\%$	t^{DEM}	S_{th}	$E_{th}\%$	S_{th}^{top}	$E_{th}^{top}\%$
1	26180	1	100	1	100	18201	1	100	1	100
12	2446	10.70	89.19	11.69	97.44	2537	7.17	59.79	10.48	87.37
24	1320	19.83	82.64	22.75	94.77	1372	13.27	55.28	18.29	76.22
48	756	34.63	72.15	34.81	72.53	685	26.57	55.36	29.17	60.77
96	590	44.37	46.22	45.61	47.51	539	33.77	35.18	41.55	43.29

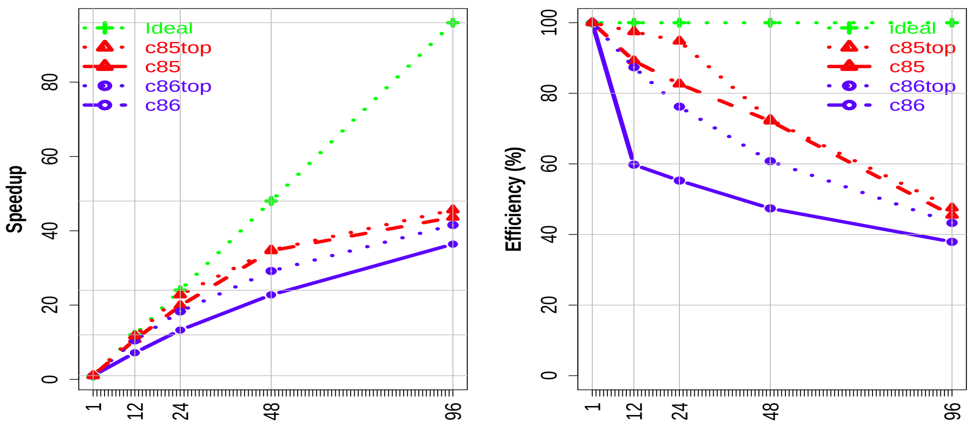

Figure 4.6: Scalability of instances c85 and c86

Observe in Table 4.10 and Figure 4.6 the remarkable scalability of the Inner P-SDP algorithm. The speedup increases almost linearly with up to 48 threads and the elapsed time with 96 threads is 44 and 34 times faster than for the serial version in instances c85 and c86, respectively. When comparing the efficiency and the top efficiency, we can observe that time lost by communication and synchronization is not significant in the implementation of Inner P-SDP for instance c85 and is quite small for instance c86.

4.5 Conclusions

The so-named inner and outer parallelization strategies have been introduced as an extension of the SDP methodology presented in Cristobal *et al.* [2009]; Escudero *et al.* [2013b] as well as some refinements in the algorithm, for obtaining fast quasi-optimal solutions to large-scale multistage stochastic mixed 0-1 problems, where the (exogenous) uncertainty is represented by nonsymmetric scenario trees. The parallel computing versions of the algorithm are referred to as Inner P-SDP and Outer P-SDP. The inner strategy parallelizes several variations of the optimization of MIP subproblems attached to the set of subtrees for each stage modeler-driven generated by defining the set of consecutive time periods to be included in each stage. The variations are due to generations by recursion of so-named references levels for the multistage linking variables among periods of immediate successor stages. Since the variations of the subproblems for each stage are independent from each other, they are optimized in parallel. Based on an extensive computational experience to assess the validity of the proposed approach, the results of the parallelization are remarkable in the testbeds we have experimented with. They basically depend on the number of threads that are allocated for parallel optimization of the subproblems in synchronized/asynchronized executions.

The outer parallelization strategy executes simultaneously as many versions of the serial algorithm as the number of so-named paths (each path managed by a main thread). The paths interchange information about the incumbent solution at the end of each FtB execution. This parallelization strategy obtains a slightly better solution value than SDP/Inner P-SDP (i.e., the serial or inner version of the algorithm), but at the price of a sometimes higher computing time, in the testbeds we have experimented with. The main advantage of the outer parallelization strategy is that it can be used in many other environments, where one of its main potential applications being the parallelization of heuristic algorithms while solving very large-scale instances.

The plain use of the state-of-the-art MIP solver of choice, CPLEX, cannot very frequently provide the optimal solution due to running out of memory or exceeding time limitation time, but it provides solutions with e.g. 0.78% quasi-optimality tolerance for an instance whose dimensions are 5.56 million constraints, 1.41 million 0-1 variables and 3.49 million continuous variables being stopped due to running out of memory at 8274 second instant time. For that instance, Inner P-SDP provides a solution value with a 0.16% gap versus the CPLEX solution value, requiring an elapsed time that is one order of magnitude smaller than CPLEX time, in particular, 978 and 7220 seconds of elapsed time where required by Inner P-SDP and SDP, respectively.

For the largest instances (up to 57.8 million constraints, 15.4 million 0-1 variables and 38.5 million continuous variables), CPLEX could not provide a solution before running out of memory while solving the LP relaxation, but Inner P-SDP and SDP got a feasible solution in up to 2446 and 26180 seconds of elapsed time, respectively, until reaching the allowed time; the solution quality, obviously, could not be assessed.

As a result of our computational experience with the inner parallelization of the SDP algorithm, we can withdraw the conclusion that the P-SDP scheme is highly efficient on parallelizable (usually, decomposition) algorithms, reaching a value up to 90% for 12 threads and, in any case, reducing the elapsed time of the SDP up to one order of magnitude with 12 threads, and while using 96 threads the time reduction is up to 44 times of the serial one.

Chapter 5

Perspectives on solving large-scale problems

It does not matter how slowly you go as long as you do not stop.
Confucius (attributed)

5.1 Introduction

The chapter aims to present several strategies in order to improve decomposition algorithms for solving large-scale problems. We describe an adaptation of the serial Branch-and-Fix Coordination algorithm presented in Chapter 3 to obtain an exact solution of larger-scale problems and faster solutions in medium-scale problems. Additionally, a series of matheuristic algorithms based on the improved BFC algorithm are proposed. These algorithms have the advantage of being applicable to any stochastic mixed 0-1 optimization problem without the high model dependence, as in the Stochastic Dynamic Programming presented in Chapter 4. However, the so-named *Dynamically-guided and stage-ordered Branch-and-Fix Coordination* (D-BFC) and its derived matHeuristic BFCs (H-BFC) are not yet able to solve very large-scale problems solved by the SDP algorithm, since the lower decomposition capabilities lead to memory limit issues.

The rest of the chapter is organized as follows: Section 5.2 presents the model ordering and node branching improvements of the Dynamically-guided and stage-ordered Branch-and-Fix Coordination (D-BFC). Section 5.3 introduces the objective and perspective of the matheuristic algorithms based on the D-BFC, referred to as H-BFCs. Section 5.4 details the D-BFC and H-BFCs procedure and illustrates each case with a branching tree figure. Section 5.5 reports the main results of a computational experience to assess the validity of the D-BFC to solve medium-scale problems and H-BFCs to solve large-scale problems. Section 5.6 describes an improved Outer-Inner parallelization perspective of the H-BFC algorithms for solving very large-scale problems. Finally, Section 5.7 summarizes the main conclusions.

5.2 D-BFC decomposition algorithm

Large-scale multistage mixed 0-1 optimization problems are very difficult to solve, mainly due to the number of constraints and the number of 0-1 variables. In order to improve the performance of the BFC algorithm presented in Section 3.3 for solving large-scale problems, let us analyse the conclusions of the P-BFC performance described in Section 3.8. On one hand, the inner parallelization reduces the execution time in each node by solving several subproblems in parallel whenever possible. On the other hand, the outer parallelization reduces the number of nodes to be visited by splitting the branching tree and by obtaining tight bounds earlier.

Let us incorporate the benefits shown in the parallel implementation to a new serial algorithm called *Dynamically-guided and stage-ordered Branch-and-Fix Coordination* (D-BFC). The two key aspects of the improvement are stage-wise model ordering and dynamic node branching.

5.2.1 Stage-wise model ordering

Let us extend the stage perspective of the problem structure, already used in the break stage scenario clustering, see Section 3.2, by an stage-wise variable ordering in the model and submodels used throughout the algorithm. First, stage-ordered models are expected to be, in general, faster to solve by using an optimization engine than group-ordered models, see Table 5.4 for a computational experience in Testbed 2. The reason is that problem structure information in a stage-ordered model is given to the optimization engine by time-wise variable dependency ordering. Therefore, the time needed to solve the problem or/and subproblems related to each node is expected to be smaller. Second, a stage-wise model ordering simplifies the variable control in BFC, see Algorithm 3.1, since a scenario-group control is no longer needed in D-BFC, see Algorithm 5.2. This aspect is likewise very useful in implementation terms, moreover when considering complex environment such as the outer parallelization.

The following model ordering procedure has been considered:

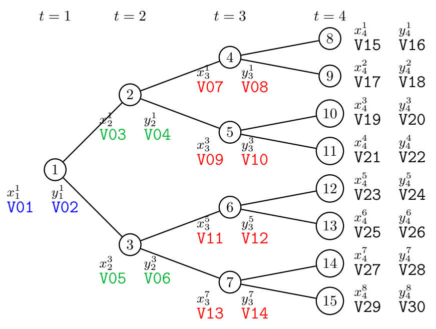

Figure 5.1: Group-wise ordering

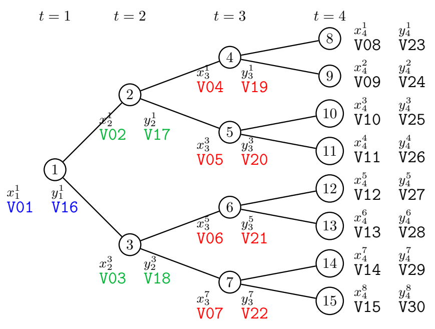

Figure 5.2: Stage-wise ordering

If in the original DEM the variables are not stage-wise ordered, then they will be ordered as follows: first the binary variables and second the continuous ones. An example is shown in Figure 5.1 and Figure 5.2 where the first shows the scenario group-wise ordering while the other shows the described stage-wise ordering. That is, the constraint system $\sum_{t' \in \mathcal{T}^t}[A,B][x,y] = h$ is variable ordered from (x_i, y_i) to $(x_{\rho_x(i)}, y_{\rho_y(i)})$, where $\rho_x : \mathcal{I}_x \to \mathcal{I}_x$ and $\rho_y : \mathcal{I}_y \to \mathcal{I}_y$ are bijections from the set of indices of $x-$ and $y-$ variables, respectively, that order them in terms of stages. Note that $\mathcal{I}_x$ $\mathcal{I}_x = \{1, \dots, nx\}$ and $\mathcal{I}_y$ $\mathcal{I}_y = \{1, \dots, ny\}$, where $nx = \sum_{t \in \mathcal{T}} nx_t$ and $ny = \sum_{t \in \mathcal{T}} ny_t$. See Aldasoro *et al.* [2013b] and Table 5.4 for a brief computational experience.

The stage-ordered DEM is broken in stage-ordered scenario cluster submodels, see Algorithm 5.1. It considers an original DEM (3.5) in mps format (Total.mps), and after ordering it obtains the Output.mps file and the cluster partitioning with Clusterc.mps, corresponding to models (3.1), for each $c \in \mathcal{C}$. Note that $\mathcal{R}$ denotes the scenario tree, and $o(x,y)$ denotes the order of variables. See Aldasoro *et al.* [2013b] for a detailed implementation.

5.2.2 Dynamic node branching

Let us improve the static node branching strategy, so far used in the previous BFC, see Algorithm 3.1, by implementing a dynamic node branching scheme. Note that in Chapter 3 the following criterion has been used: fixing first to 0 for minimization and to 1 for maximization, where opposite branching is considered in the already branched nodes. See more strategies in Escudero *et al.* [2009a] for guided static branching related to root node.

A dynamic node branching based on the binary variable values of cluster submodel solutions is presented. Tight feasible solutions are thus found earlier,

Algorithm 5.1: Model ordering

```
Step 1: Input file: read full model **Total.mps** with original variable order
        Input data: read $t^*$, $T$, $\mathcal{G}_t$, $\mathcal{R}$, $n_{xt}$, $n_{yt}$, $w^\omega$, $o(x,y)$.
Step 2: Calculate additional vectors.
Step 3: Reorder objective function coefficients $a$ and $b$
        Reorder columns of constraints matrices $A', A, B'$, and $B$
        Reorder bounds of the continuous variables $x$ and $y$
        according to the order of variables vector.
Step 4: Generate the stage-ordered full model **Output.mps**.
Step 5: Link full model variables with the corresponding cluster using
        the cluster tree matrix.
Step 6: Link rows to clusters. By default all are assigned.
        For $i = 1$ to nelements do.
          For $j = 1$ to $C$ cluster submodel do.
            If Column of element $i$ does not belong to Cluster $j$ then.
              Unlink row of element $i$ from Cluster $j$.
Step 7: Renumber cluster rows.
        Reorder the right-hand-side vector.
        Renumber the element vector and update corresponding row index.
Step 8: Generate stage-ordered **Clusterc.mps** files
```

so that fewer nodes are visited during the algorithm execution. In addition to the serial implementation improvement, this criterion has significant potential in outer parallelization schemes, since simultaneous dynamic node branching paths are executed.

The following node branching procedure has been considered:

- At a node, if the next $x-$ variables satisfy NAC, then the branching jumps to the first variable that does not satisfy NAC.

- Then, *Dynamically-Guided branching* is a potentially powerful strategy, let us use DG branching strategy to denote it. It considers the following criteria: start branching to σ_i parameter for variable x_i of DEM that corresponds to variable $x^c_{\langle i \rangle}$ in the related scenario cluster submodel (3.1), being $\langle i \rangle$ the index of variable x_i in the current cluster submodel c, where

$$\sigma_i = \begin{cases} 0, & \text{if} \quad \sum_{c \in \mathcal{C}_i} \overline{x}^c_{\langle i \rangle} \leq |\mathcal{C}_i| - \sum_{c \in \mathcal{C}_i} \overline{x}^c_{\langle i \rangle} \\ 1, & \text{otherwise} \end{cases}$$

and $\mathcal{C}_i \subseteq \mathcal{C}$ is the set of cluster submodels that have variable x_i in common. That is, the first branching is stated at the more frequent variable solution value under the set of scenario cluster models of that variable i in the previous iteration (where node x_{i-1} is fixed).

- The cluster submodels (3.1) where some $x-$ variables have been branched are just solved once in the Branch-and-Fix tree and stored for future needs. Only the cluster submodels where a new variable is branched are solved and only for the first time of the sequence of branched variables appear. This improvement expects a remarkable elapsed time improvement for instances with a large number or quite difficult submodels, in exchange of a remarkable storage to effort.

Table 5.1 and Table 5.2 show the performance differences between a static branching criterion (always to 0) and the Dynamically-Guided branching (DG). Let us take an example a problem with six clusters ($C1, C2, C3, C4, C5, C6$) when applying a $t^* = 1$ partitioning and has five binary variables (x_1, x_2, x_3, x_4, x_5) in stage $t = 1$. The headings are as follows: N counts the number of nodes that have been visited and $C1$, $C2$, $C3$, $C4$, $C5$ and $C6$ indicate the values of the binary variables in stage $t = 1$ obtained under the given branching fixations. Note that in the examples only the results related to the cluster submodels (3.1) of the last node are stored in memory.

Table 5.1 describes the variable branching performance applied in BFC, see Chapter 3 and Algorithm 3.1, i.e., static branching criterion (always to 0). After solving $N = 1$, variable x_1 already satisfies the NAC and it is value is equal to the branching criterion, therefore $x_1 = 0$ is fixed and the next variable is branched, $x_2 = 0$. As none of the clusters has already $x_2 = 1$ at the solution, all need to be resolved. Once $N = 2$ is solved, cluster $C1$ is found infeasible, therefore the branch must be pruned and go backwards. Now, as has been previously visited, x_2 is branched to the opposite direction, i.e., $x_2 = 1$. Later, $N = 3$ is visited and x_3 does not satisfy the NAC, therefore it is branched following the static criterion, $x_3 = 1$. The algorithm execution continues in this way until all nodes have been visited.

Table 5.2 describes the variable branching performance applied in D-BFC, i.e., Dynamically-Guided branching (DG). After solving $N = 1$, variables x_1 and x_2

already satisfy the NAC, their values are taken as new branching criterion. Therefore $x_1 = 0$ and $x_2 = 1$ are fixed and the next variable is branched to the most frequent value at the solutions, $x_3 = 1$, since it appears in four cluster solutions. Only the clusters where $x_3 = 1$ does not appear in the solution need to be resolved. Once $N = 2$ is solved, variable x_4 does not satisfy the NAC so it is fixed to the most frequent value, i.e., $x_4 = 0$. Only cluster $C6$ needs to be resolved. Now, $N = 3$ is solved and x_5 satisfies the NAC; being the last binary variable in stage $t^* = 1$, the solution is feasible and the branch is pruned. Going backwards, variable x_4 is fixed to the opposite direction, i.e., $x_4 = 1$. The algorithm execution continues this way until all nodes have been visited.

Table 5.1: BFC variable branching performance, example

N	C1	C2	C3	C4	C5	C6
1	$\{0,1,0,0,0\}$	$\{0,1,1,0,0\}$	$\{0,1,1,0,0\}$	$\{0,1,1,0,0\}$	$\{0,1,1,0,0\}$	$\{0,1,0,0,0\}$
Fix $x_1 = 0$. Branch $x_2 = 0$. Resolve $C1, C2, C3, C4, C5, C6$.						
2	**infeasible**	$\{\mathbf{0},\mathbf{0},1,0,0\}$	$\{\mathbf{0},\mathbf{0},1,0,0\}$	$\{\mathbf{0},\mathbf{0},1,0,0\}$	$\{\mathbf{0},\mathbf{0},1,0,0\}$	$\{\mathbf{0},\mathbf{0},1,0,0\}$
C1 infeasible: prune and go backwards. Branch $x_2 = 1$. Resolve $C1, C2, C3, C4, C5, C6$.						
3	$\{\mathbf{0},\mathbf{1},0,0,0\}$	$\{\mathbf{0},\mathbf{1},1,0,0\}$	$\{\mathbf{0},\mathbf{1},1,0,0\}$	$\{\mathbf{0},\mathbf{1},1,0,0\}$	$\{\mathbf{0},\mathbf{1},1,0,0\}$	$\{\mathbf{0},\mathbf{1},0,0,0\}$
Branch $x_3 = 0$. Resolve $C2, C3, C4, C5$.						
...						

Static branching criterion, always to 0.

Table 5.2: D-BFC variable branching performance, example

N	C1	C2	C3	C4	C5	C6
1	$\{0,1,0,0,0\}$	$\{0,1,1,0,0\}$	$\{0,1,1,0,0\}$	$\{0,1,1,0,0\}$	$\{0,1,1,0,0\}$	$\{0,1,0,0,0\}$
Fix $x_1 = 0$ and $x_2 = 1$. Branch $x_3 = 1$. Resolve C1, C6.						
2	$\{\mathbf{0},\mathbf{1},\mathbf{1},0,0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},0,0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},0,0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},0,0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},0,0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},1,0\}$
Branch $x_4 = 0$. Resolve C6.						
3	$\{\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{0},0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{0},0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{0},0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{0},0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{1},0\}$	$\{\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{0},0\}$
NAC satisfied: prune and go backwards. Branch $x_4 = 1$. Resolve $C1, C2, C3, C4, C5, C6$.						
...						

Dynamically-Guided branching, DG.

5.3 H-BFCs, D-BFC based matheuristic algorithms

Let us consider several matheuristic versions of the D-BFC algorithm, see Section 5.2, based on the relaxation of some steps in order to solve large and very large-scale problems.

The first algorithm, named H1-BFC, takes into account that the integer TNF model (3.10), z_{MIP}^{TNF}, is often the main bottleneck of the algorithm for large-scale problems, and considers not solving it. Therefore, the certainty of optimality is lost but knowing that earlier breakstages are recommendable for a better quality solution and smaller Branch-and-Fix tree, $\mathcal{BFT}$, tight bounds can be achieved even in large-scale problems, see Proposition 1 and Proposition 3.

The second algorithm, named H2-BFC, aims to solve very large-scale problems by adding features to H1-BFC. When solving models with a large number of 0-1 variables until the break stage, the breadth of the branch-and-fix tree, $\mathcal{BFT}$, becomes an algorithmic bottleneck. A strategy for cutting short the tree is implemented during the *backward branching*, with the following criterion: the backward branching is performed just in the previously branched variables, i.e., the variables that were jumped because they satisfied the NAC during the branching down, are again jumped during the backward branching. Additionally, the scenario cluster submodels could be quite large and then, the relaxation of (3.8) submodels, desirable.

The H3-BFC procedure conserves the characteristics of H2-BFC and considers stopping the branch-and-fix tree at the first feasible solution, let us denote it as the SF branching strategy.

In summary, the characteristics of BFC, D-BFC and H-BFCs are given in Table 5.3. The headings are as follows: *Algorithm*(d) the Branch-and-Fix Coordination algorithm version later used in Algorithm 5.2; *Stage ordering* whether the model is stage-wise ordered; DG, if the variable branching is dynamically decided; IB, if an incomplete backward branching is applied; SF, whether the algorithm execution is stopped at the first feasible solution found; z_{LP}^{TNF}, if model (3.6) is solved in Step 5; z_{f}^{TNF}, if (3.8) models are solved in Step 5; and finally, z_{MIP}^{TNF}, if model (3.10) is solved in Step 5.

Table 5.3: D-BFC and H-BFC algorithms

Algorithm (d)	Stage	Branching			Integer TNF models		
	ordering	DG	IB	SF	z_{LP}^{TNF}	z_{f}^{TNF}	z_{MIP}^{TNF}
BFC	no	no	no	no	yes	yes	yes
D-BFC (0)	**yes**	**yes**	no	no	yes	yes	yes
H1-BFC (1)	**yes**	**yes**	no	no	yes	yes	**no**
H2-BFC (2)	**yes**	**yes**	**yes**	no	yes	**no**	**no**
H3-BFC (3)	**yes**	**yes**	**yes**	**yes**	yes	**no**	**no**

DG : Dynamically-Guided branching
IB : Incomplete Backward branching
SF : Stop at First feasible solution.

5.4 D-BFC and H-BFC procedures

Based on Algorithm 3.1 and the improvements described in Section 5.2, the *Dynamically-guided and stage-ordered Branch-and-Fix Coordination* (D-BFC) procedure is detailed in Algorithm 5.2. The initialization phase in Step 0 allows the user to select the desired algorithm to be used, since variable d defines the difficulty level according to Table 5.3, i.e., whether D-BFC, H1-BFC, H2-BFC or H3-BFC is used. Therefore, Algorithm 5.2 summarizes the procedures of all cases.

Broadly speaking, the main differences between Algorithm 3.1 and Algorithm 5.2 are as follows: the stage-wise model ordering allows control not to be kept of the current scenario group that is being branched; the Dynamically-Guided branching, DG, stores the cluster binary variable solutions and selects the branching direction, as opposed to the static branching criteria and, finally, some steps of the procedure are relaxed depending on the chosen algorithm type (d).

The initialization phase in Step 0 does not solve the LP relaxation of the DEM (3.5) to obtain a lower bound, as it is done in Algorithm 3.1, since for large and very large-scale problems, significant elapsed time and memory use are expected. Additionally, the algorithm type is set, d, and a boolean variable controlling if the first feasible solution has been found is initialized, $first_{feas} := 0$.

The root node analysis in Step 1 remains the same, with the only difference being the storing of the variable branching criteria in σ, based on the binary variable values of the cluster submodel solutions.

The branching down procedure, Step 2 and Step 3, is significantly simpler than before (previous Step 2 to Step 5), since the stage-scenario group-variable control is no longer needed. Instead, controlling the binary variable number allowed by the stage-wise variable ordering, is enough to activate the backward branching. As previously described, the variable branching direction is chosen depending on the σ value.

The candidate TNF analysis in Step 4 by solving the scenario cluster MIP submodels (3.1), includes the update of the σ values, according to Section 5.2.2, and the set of boolean variable $first_{feas}$ to 1 if a feasible solution is found.

The integer TNF models in Step 5 include a series of solving processes that depend on the chosen algorithm type. Thus, the submodels (3.8) to obtain z^{TNF}_{fc} are only solved by D-BFC and H1-BFC algorithms, and the MIP model (3.10) to obtain z^{TNF}_{MIP} is only solved by the D-BFC algorithm (therefore D-BFC is the only exact algorithm). In addition, after every solving process the feasibility is checked

Algorithm 5.2: Serial D-BFC and H-BFCs

```
Step 0: (Initializations)
    Set algorithm type by $d \in \{0,1,2,3\}$, see Table 5.3.
    Set $z^{DEM} := \infty$, $i := 0$, $first_{feas} := 0$.
Step 1: (Root node)
    Solve the scenario cluster MIP submodels (3.1) to obtain $z^c \, \forall c \in \mathcal{C}$.
    Compute $z^0_{t^*} = \sum_{c \in \mathcal{C}} z^c$ and $(\sigma_i)_{i \in \mathcal{I}_x}$.
    If $x^c$ variables do not satisfy NAC (3.3), then go to Step 2.
    If $y^c$ variables do not satisfy NAC (3.4), then go to Step 5.
    Otherwise, $z^{DEM} := z^0_{t^*}$ is the optimal solution of DEM (3.5), STOP.
Step 2: (Next node)
    Reset $i := i + 1$. If $i > \sum_{t \in \mathcal{T}_1} nx_t$, then go to Step 7.
Step 3: (Branching)
    Set $x^c_{\langle i \rangle} := \sigma_i \, \forall c \in \mathcal{C}_i$.
Step 4: (Candidate TNF)
    Solve the scenario cluster MIP submodels (3.1) to obtain $z^c \, \forall c \in \mathcal{C}$,
    Compute $z = \sum_{c \in \mathcal{C}} z^c$ and $(\sigma_i)_{i \in \mathcal{I}_x}$.
    If $z \geq z^{DEM}$, then go to Step 6.
    If any variable $\mathbf{x}^c$ does not satisfy NAC (3.3) $\forall t \in \mathcal{T}_1, c \in \mathcal{C}$, then update $i$
    according to the first variable that do not satisfy NAC and go to Step 2.
    If all variables in $\mathbf{y}^c_t$ do satisfy NAC (3.4) $\forall t \in \mathcal{T}_1, c \in \mathcal{C}$,
    then update $z^{DEM} := z$, $first_{feas} = 1$ and go to Step 6.
Step 5: (Integer TNF models)
    Solve LP model (3.6) to obtain $z^{TNF}_{LP}$.
    If it is feasible, update $z^{DEM} := \min\{z^{TNF}_{LP}, z^{DEM}\}$, $first_{feas} = 1$.
    If $d \leq 1$, then solve the submodels (3.8) to obtain $z^{TNF}_{fc} \, \forall c \in \mathcal{C}$.
    If it is feasible, compute $z^{TNF}_f = \sum_{c \in \mathcal{C}} z^{TNF}_{fc}$ and
    update $z^{DEM} := \min\{z^{TNF}_f, z^{DEM}\}$, $first_{feas} = 1$.
    If $d = 0$, then solve MIP model (3.10) to obtain $z^{TNF}_{MIP}$.
    If it is feasible, update $z^{DEM} := \min\{z^{TNF}_{MIP}, z^{DEM}\}$.
Step 6: (Branch pruning).
    If $d = 3$ and $first_{feas} = 1$, then update $i := 1$ and go to Step 7.
    If $x^c_{\langle i \rangle}$ has been branched to $\sigma_i$ for any $c \in \mathcal{C}_i$, then go to Step 9.
Step 7: (Backward to previous node)
    Reset $i := i - 1$.
    If $d \geq 2$, update $i$ according to the explicitly branched down variables.
    If $i = 0$, then the solution value $z^{DEM}$ has been found, STOP.
Step 8: (Prune checking)
    If $x^c_{\langle i \rangle} = 1 - \sigma_i$ for any $c \in \mathcal{C}_i$, then go to Step 7.
Step 9: (Opposite branching)
    Reset $x^c_{\langle i \rangle} := 1 - \sigma_i \, \forall c \in \mathcal{C}_i$ and go to Step 4.
```

in order to update the incumbent solution z^{DEM} and the boolean variable $first_{feas}$.

The branch pruning in Step 6 includes the relaxation related to H3-BFC ($d = 3$), i.e., if a feasible solution has been already found, $first_{feas} = 1$, then the variable index i is set to 1 and the execution is redirected to Step 7, where it stops.

The backward branching in Step 7 is decided by checking the variable index i, stage and scenario group control are no longer needed. Additionally, the variable index i in H2-BFC and H3-BFC is updated according to explicitly branched down variables.

Finally the prune checking and the opposite branching direction, Step 8 and Step 9, are performed according to the σ values, and not according to a static branching criteria as in BFC, see Algorithm 3.1.

5.4.1 Solving performance example of BFC, D-BFC and H-BFCs

Let us compare in an illustrative way the performance of algorithms BFC, D-BFC and H-BFCs in order to visually analyse the differences, benefits and performance relaxations, see Figure 5.3, Figure 5.4, Figure 5.5 and Figure 5.6. The chosen problem corresponds to instance P3 of Testbed 2, see dimensions in Table 3.2, where break stage $t^* = 1$ is applied. Therefore, ten scenario cluster subproblems are created and the Branch-and-Fix tree, $\mathcal{BFT}$, has six binary variables.

Figure 5.3 illustrates the BFC resolution of instance P3 according to Algorithm 3.1, note that the performance of the Outer P-BFC is described in Figure 3.5; in this serial case one-path is considered so the algorithm root node is situated at the Branch-and-Fix tree root node (see node 1 double-lined). The resolution visits 24 nodes before ensuring optimality, where the deepest level of the Branch-and-Fix tree is reached six times (nodes 7, 8, 11, 12, 19 and 20). Only three nodes are jumped due to satisfying NAC and having a value equal to the static branching criterion (between nodes 10 and 11, 17 and 18, 18 and 19) which indicates that a dynamic branching criterion could improve the performance. Six feasible solutions, z, are found during the execution, and in all cases the incumbent solution, z^{DEM}, is updated (nodes 7, 8, 11, 19, 20 and 22). Note that node 22 is the only node where a feasible solution is found and is not situated in the last level, showing that with the current branching criterion the execution cannot easily find early feasible solutions. The nodes where a red *prune* label is displayed, indicate that the bound obtained by MIP submodels (3.1) is larger than the incumbent solution, z^{DEM}, therefore the branch is pruned.

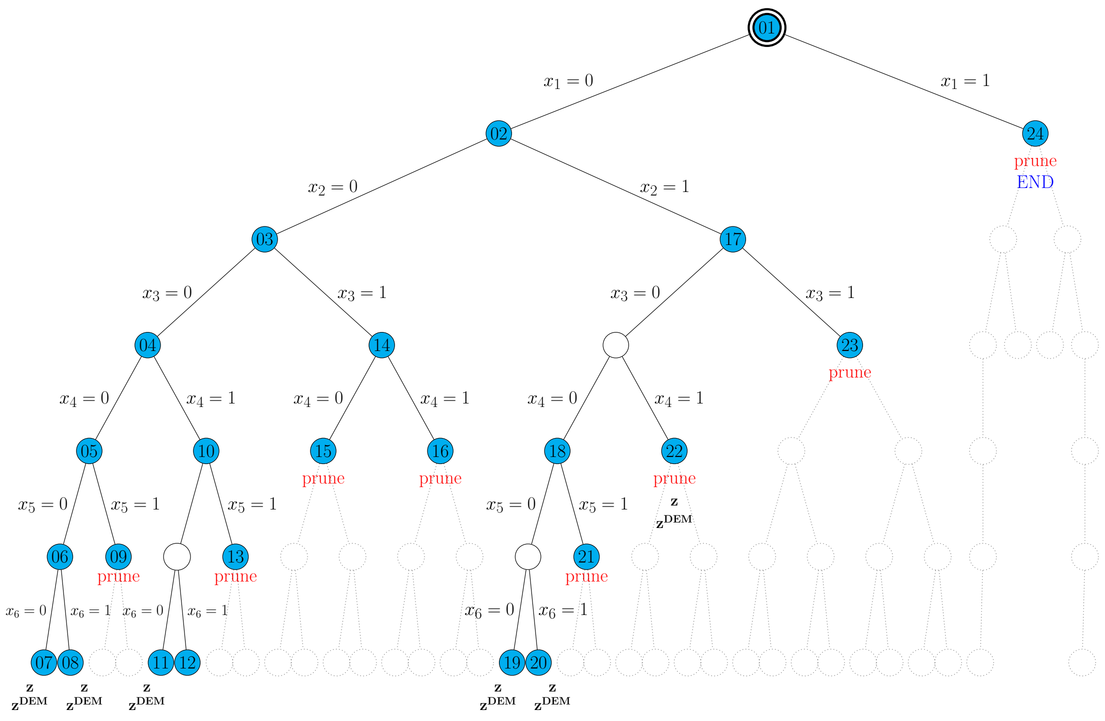

Figure 5.3: BFC performance for instance P3

Figure 5.4 illustrates the D-BFC and H1-BFC resolution of instance P3 according to Algorithm 5.2, being $d = 0$ and $d = 1$ respectively. Note that in this case both performances are equal since the integer TNF solution is found when obtaining the z_f^{TNF} bound, and no better solution is obtained by solving the MIP model (3.10) (by definition never solved by H1-BFC). The resolution visits 6 nodes before ensuring optimality, where the deepest level of the Branch-and-Fix tree is never reached. Three nodes are jumped after solving the root node (between nodes 1 and 2) due to satisfying NAC. Note that in this case no static branching criteria is fixed and the resolution dynamically decides the branch to take. This leads to a feasible solution being found at node 2, which is significantly earlier than BFC, see Figure 5.3, both in node number and tree depth terms. The tight incumbent solution, z^{DEM}, allows the successive visited nodes to be pruned since the bounds obtained by MIP submodels (3.1) are larger than the incumbent solution. Additionally, the pruned nodes are situated in upper levels so few backward branching movements are needed. All aspects considered, the D-BFC and H1-BFC algorithm show a significantly more efficient analysis of the Branch-and-Fix tree, $\mathcal{BFT}$.

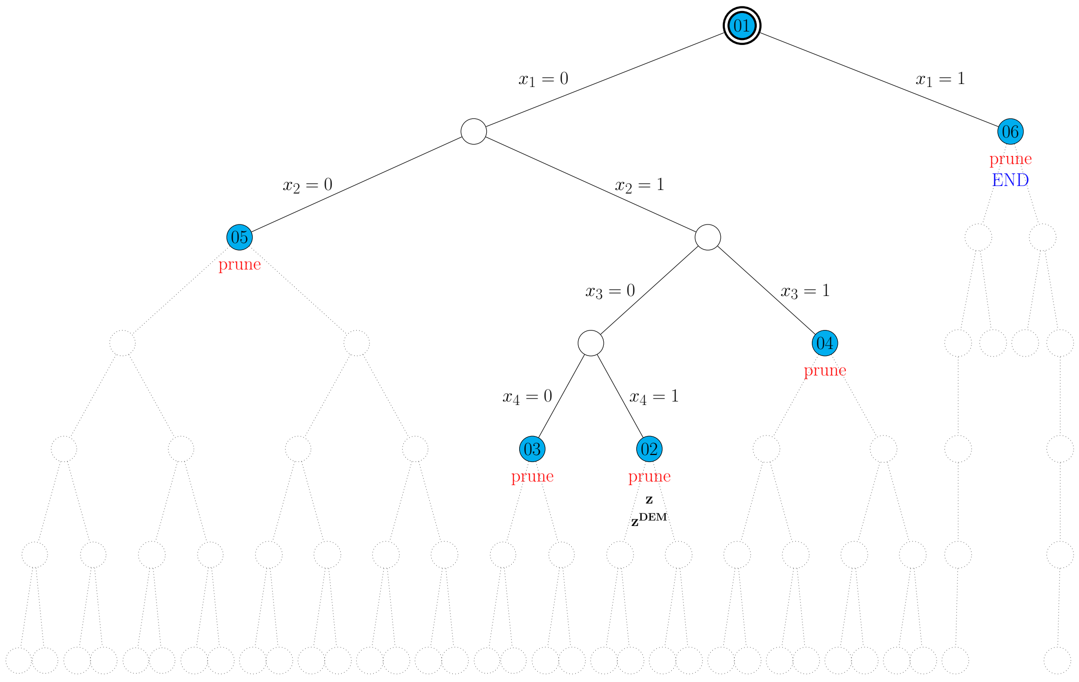

Figure 5.4: D-BFC and H1-BFC performance for instance P3

Figure 5.5 illustrates the H2-BFC resolution of instance P3 according to Algorithm 5.2, being $d = 2$. Note that H2-BFC is based on H1-BFC and the additional features are: only already visited variables are branched in the backward branching and no bound z_f^{TNF} by the MIP subproblems (3.8) are calculated. The resolution visits 3 nodes before finishing (without ensuring optimality), where the deepest level of the Branch-and-Fix tree is never reached. As in D-BFC and H1-BFC, the dynamic branching criterion results in a feasible solution being obtained at node 2. After pruning, the backward branching phase starts and as H2-BFC only opposite branches the variables already branched, in this case only x_4 is considered. Therefore, x_4 is opposite branched and pruned due to the bound obtained by MIP submodels (3.1) is larger than the incumbent solution. As all previously branched variables have been opposite branched, execution ends. The example shows that H2-BFC is intended to solve very large-scale problems by reducing the number of visited nodes with respect to H1-BFC; the dynamic branching criterion allows only the a priori most influential binary variables to be branched in the backward branching.

Figure 5.6 illustrates the H3-BFC resolution of instance P3 according to Algorithm 5.2, being $d = 3$. Note that H3-BFC is based on H2-BFC and the additional features is that the execution stops when finding a feasible solution. The resolution visits 2 nodes before finishing (without ensuring optimality), where the deepest level of the Branch-and-Fix tree is never reached. As in D-BFC, H1-BFC and H2-BFC, the dynamic branching criterion leads to a feasible solution being obtained at node 2. Consequently algorithm stops being the incumbent solution, z^{DEM}, the first feasible solution, z, is obtained. Figure 5.6 shows that H3-BFC aims to solve very large scale problems and reduce the execution. This feature is desirable when dealing with very large-scale problems and when big size data is stored, since memory overloads may appear and unexpectedly abort the execution. The example of solving the medium-scale instance P3 returns the same solution, which is optimal, in all considered cases. As shown in the computational experience reported in Section 5.5, in general the larger difficult level chosen, d, the larger the obtained optimality gap. On the other hand, larger difficulty level lead to obtain a tight bound for larger-scale problems.

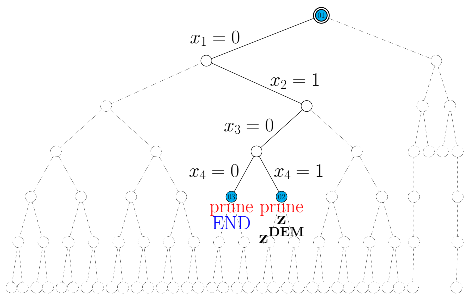

Figure 5.5: H2-BFC performance for instance P3

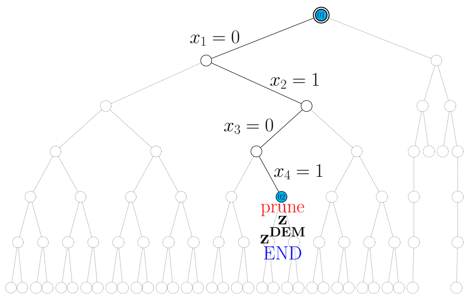

Figure 5.6: H3-BFC performance for instance P3

5.5 Computational experience

The computational experiments were conducted at the ARINA computational cluster at SGI/IZO-SGIker from UPV/EHU, see Section 1.5. For the reported experiments, a Intel Xeon type computing node was used, consisting of 8 cores with 48Gb of RAM.

The D-BFC and H-BFC algorithms were implemented in a `C++` experimental code which uses the state-of-the-art optimization LP/MIP solver CPLEX V12.5 called from the open source library COIN-OR V1.6.0 . The optimizer is used by the algorithm to solve the LP relaxation of the original DEM (3.5), the MIP submodels (3.1) for the set of scenario clusters $\mathcal{C}$ in different steps, the LP submodel (3.6), the MIP submodels (3.8) for the set of scenario clusters $\mathcal{C}$ and the MIP model (3.10).

The computational experience is reported as follows: Section 5.5.1 and Section 5.5.2 set out the performance of the D-BFC and H-BFC algorithms, respectively.

5.5.1 D-BFC compared to BFC in medium-scale problems

Let us see the influence of the stage-wise problem ordering, see Algorithm 5.1, compared to the group-wise ordering on Testbed 2. Additionally the use of CPLEX under COIN-OR and the plain use of *interactive CPLEX* are reported in order to compare the optimization engine influence (MIPgap %10^{-4} in both cases). Note that the stage-wise ordering is faster for all instances when using interactive CPLEX executions; the use of CPLEX under COIN-OR shows that stage-wise ordering is faster in 9 out of 10 instances where the optimal solution has been found. Comparing interactive CPLEX to CPLEX under COIN-OR for stage-wise ordered problems, the results show that the first is faster in 5 instances and the second in 4 instances. However, note the difference in instances P5 and P12 (only solved by CPLEX under COIN-OR). The influence of the optimization engine of choice is shown to be significant. Finally, let us compare the performance of the new CPLEX V12.5 and COIN-OR V1.6.0 with respect to CPLEX V12.2 and COIN-OR V1.3.1 by analysing the last column of Table 3.9. Now, the optimal solution of instances P5 and P12 is obtained within the defined time limit.

Table 5.4: Effect of model ordering in elapsed time. Testbed 2

Instance	CPLEX under COIN		interactive CPLEX	
	by groups	by stages	by groups	by stages
P1	21	16	3	3
P2	1690	991	21	18
P3	–[a]	–[a]	–[a]	–[a]
P4	–[a]	–[a]	–[a]	–[a]
P5	3256	4767	4	2
P6	4315	4160	3842	1603
P7	677	492	530	420
P8	–[a]	–[a]	–[a]	–[a]
P9	–[a]	–[a]	–[a]	–[a]
P10	12	8	187	102
P11	253	141	190	186
P12	17338	14435	–[a]	–[a]
P13	1277	739	2220	1203
P14	923	498	1678	1039

[a]: Time limit (6h) exceeded

Table 5.5 shows the performance of D-BFC with respect to the previous BFC algorithm, described in Chapter 3, and CPLEX for Testbed 2, see Table 3.2 and Table 3.3. Note that both BFC and D-BFC require a smaller elapsed time than CPLEX in all cases. The D-BFC algorithm is faster than BFC in 8 of 14 instances, including the most difficult ones, P3 and P4. The obtained solution value, z^{DEM}, is optimal in all instances; P8 and P9 show a small difference (lower than the optimality gap 0.01%) due to the different model variable ordering, the optimization engine obtains the optimality gap in different points. Fewer integer TNF models, n^{TNF}, are solved by D-BFC whereas in instances P6 and P7 the total number of visited nodes, n^n, is higher. Note that the number of clusters submodels solved n^c per visited node n^n is significantly lower by avoiding unnecessary resolves.

Table 5.5: D-BFC performance. Testbed 2.

	CPLEX			BFC						D-BFC					
	z^{DEM}	$OG\%$	t_{DEM}	n^n	n^{TNF}	n^c	z^{DEM}	$GG\%$	t_{DEM}	n^n	n^{TNF}	n^c	z^{DEM}	$GG\%$	t_{DEM}
P1	-156324	$*$	23	1	0	8	-156324	*	1	1	0	8	-156324	*	2
P2	-6146.04	$*$	3049	3	1	21	-6146.04	*	45	3	1	15	-6146.04	*	299
P3	-292102[a]	0.11	–	24	6	240	-292109	*	332	6	1	51	-292109	*	41
P4	-283915[a]	0.30	–	12	6	120	-283938	-0.01	1727	11	5	92	-283938	-0.01	786
P5	-6067.3	0.27	–	1	0	6	-6067.51	*	1	1	0	6	-6067.51	*	1
P6	-35959.9	$*$	4239	3	2	24	-35959.9	*	88	7	1	56	-35959.9	*	102
P7	-269441	$*$	461	3	2	21	-269441	*	115	13	1	25	-269441	*	108
P8	-154798[a]	0.04	–	1	0	9	-154814	-0.01	26	1	0	9	-154798	*	18
P9	-225746[a]	0.10	–	1	0	10	-225754	*	109	1	0	10	-225784	-0.02	65
P10	38156.6	$*$	25	1	0	5	38156.6	*	1	1	0	5	38156.6	*	1
P11	39805.7	$*$	530	3	2	15	39805.7	*	10	4	1	20	39805.7	*	11
P12	41502.3[a]	0.03	–	3	2	18	41502.3	*	47	4	1	24	41502.3	*	17
P13	41337.4	$*$	1187	3	2	21	41337.4	*	40	4	1	22	41337.4	*	30
P14	41783.5	$*$	509	3	2	27	41783.5	*	119	4	1	12	41783.4	*	54

*: Optimality gap achieved ($< 0.01\%$)
[a]: Time limit (6h) exceeded

5.5.2 H-BFCs compared to SDP in large-scale problems

Table 5.6 shows the performance of H2-BFC with respect to the previous BFC algorithm described in Chapter 3 and CPLEX for Testbed 2, medium-scale problems. As shown in Table 5.5 the number of visited integer TNF models, n^{TNF}, is already low when using D-BFC, therefore, the H1-BFC performance is almost equivalent. Table 5.6 shows that H2-BFC significantly reduces the number of visited nodes, n^n, the number of integer TNF models, n^{TNF}, and the number of solved cluster submodels n^c. However the obtained solution value, z^{DEM}, remains the same, showing that the dynamic guided variable branching efficiently directs the branching to a tight bound and allows the incomplete backward branching.

Table 5.7 shows the performance of H2-BFC and H3-BFC with respect to the previous SDP algorithm described in Chapter 4 and CPLEX for Testbed 3, large-scale problems, introduced in Table 4.1 and Table 4.2.

Table 5.7 shows that the H-BFC obtains better solutions, in general, than the previous SDP algorithm for Testbed 3, and the elapsed time is competitive with respect to CPLEX for 12 out of 18 instances. We have seen that in general, the larger the breakstage, the quality of the solution and the elapsed time is worst but larger problems can be solved.

Table 5.6: H-BFC performance. Testbed 2

	CPLEX			BFC						H2-BFC					
	z^{DEM}	$OG\%$	t_{DEM}	n^n	n^{TNF}	n^c	z^{DEM}	$GG\%$	t_{DEM}	n^n	n^{TNF}	n^c	z^{DEM}	$GG\%$	t_{DEM}
P1	-156324	$*$	23	1	0	8	-156324	*	1	1	0	8	-156324	*	1
P2	-6146.04	$*$	3049	3	1	21	-6146.04	*	45	3	1	15	-6146.04	*	4
P3	-292102[a]	0.11	–	24	6	240	-292109	*	332	3	1	21	-292109	*	13
P4	-283915[a]	0.30	–	12	6	120	-283938	-0.01	1727	5	2	32	-283938	-0.01	379
P5	-6067.3	0.27	–	1	0	6	-6067.51	*	1	1	0	6	-6067.51	*	1
P6	-35959.9	$*$	4239	3	2	24	-35959.9	*	88	1	1	8	-35959.9	*	13
P7	-269441	$*$	461	3	2	21	-269441	*	115	1	1	7	-269441	*	39
P8	-154798[a]	0.04	–	1	0	9	-154814	-0.01	26	1	0	9	-154798	*	19
P9	-225746[a]	0.09	–	1	0	10	-225754	*	109	1	0	10	-225784	-0.02	65
P10	38156.6	$*$	25	1	0	5	38156.6	*	1	1	0	5	38156.6	*	2
P11	39805.7	$*$	530	3	2	15	39805.7	*	10	1	1	5	39805.68	*	2
P12	41502.3[a]	0.03	–	3	2	18	41502.3	*	47	1	1	6	41502.27	*	4
P13	41337.4	$*$	1187	3	2	21	41337.4	*	40	1	1	7	41337.42	*	8
P14	41783.5	$*$	509	3	2	27	41783.5	*	119	1	1	9	41783.5	*	31

*: Optimality or goodness gap achieved ($< 0.01\%$)
[a]: Time limit (6h) exceeded

Table 5.7: H-BFC performance. Testbed 3

	CPLEX			SDP					H-BFC								
	z^{DEM}	$OG\%$	t_{DEM}	*n iter*	*n prob*	z^{DEM}	$GG\%$	t_{DEM}	H	t^*	C	n^n	n^{TNF}	n^c	z^{DEM}	$GG\%$	t_{DEM}
c43	3498249[b]	0.06	16212	5	566	3539594	1.18	33	H3	1	3	19	1	0	3508437	0.29	578
c44	4211366[b]	0.03	24252	15	2256	4249979	0.92	145	H2	1	3	15	8	31	4233786	0.53	333
c45	8036004	$*$	37	4	372	8123167	1.08	28	H3	1	2	18	1	0	8127716	1.14	238
c46	8087808	$*$	375	5	487	8139108	0.63	51	H2	1	2	51	14	75	8093652	0.07	1131
c47	7151251[b]	0.09	9421	15	2256	7227178	1.06	433	H3	2	9	39	1	0	7187715	0.51	259
c48	6594167[b]	0.12	9007	6	708	6660629	1.01	108	H3	2	9	31	1	0	6632095	0.58	310
c49	993334	$*$	1	15	1857	1004692	1.14	23	H2	1	2	1	1	2	993433	0.01	6
c50	1005119	$*$	2	3	261	1006477	0.14	4	H2	1	2	11	0	15	1005119	0.00	14
c51	772567	$*$	16	4	1186	775933	0.44	14	H2	1	3	7	1	13	772573	0.00	29
c52	862754	$*$	20	5	1538	870090	0.85	18	H2	1	3	3	2	7	864033	0.15	24
c53	670234[b]	0.32	15091	4	1186	685613	2.29	59	H3	7	648	532	1	0	694772	3.66	805
c54	769236[b]	0.19	16938	15	5806	776389	0.93	486	H2	4	81	219	110	829	784165	1.94	963
c55	1163290[b]	0.07	8948	5	919	1165132	0.16	73	H2	4	16	9	5	30	1164946	0.14	145
c56	1126270	$*$	43	3	501	1128968	0.24	20	H2	1	2	1	1	2	1129249	0.26	64
c57	7174215[b]	0.06	10064	4	372	7256183	1.14	91	H3	3	8	36	1	0	7213330	0.55	1347
c58	8753936[b]	0.02	23991	7	729	8803020	0.56	200	H3	3	8	62	1	0	8915326	1.84	2141
c59	8200795[b]	0.08	15974	3	840	8251017	0.61	171	H3	3	27	92	1	0	8262957	0.76	1297
c60	8582624[b]	0.13	17757	4	1186	8640233	0.67	473	H3	3	27	117	1	0	8636077	0.62	1501

$*$ Optimality gap or goodness gap achieved ($< 0.01\%$)
[b] Out of memory (35Gb)

5.6 Perspectives on parallelizing the H-BFC decomposition algorithm

The solution based dynamic branching paradigm that improved the performance of D-BFC with respect to BFC can be also applied to the parallel H-BFC algorithms. Thus, the outer-inner parallelization of the serial BFC algorithm, see Section 3.6, is extended to a solution based *dynamic path assignment* Outer-Inner PH-BFC.

The Outer-Inner PH-BFC algorithm defines the path root nodes based on the binary variables that do not verify the NAC. As opposed to the static path root node definition in the Outer-Inner P-BFC described in Algorithm 3.4, the dynamic path assignment branches the most influential variables, allowing feasible solutions and more balanced path work loads (therefore lower waiting time for a synchronization phase) to be obtained earlier. The algorithm is particularly suitable for H2-BFC and H3-BFC based Outer-Inner PH-BFC algorithm when solving very large scale problems, since both the path root nodes and the branching direction are dynamic and a feasible solution is more likely to be found in parallel.

Algorithm 5.3 describes the *dynamic path assignment* Outer Inner PH-BFC procedure. The serial D-BFC and H-BFC procedure in Algorithm 5.2 is reduced to only consider the H2-BFC and H3-BFC cases, as shown in Step 0. The general scheme of the Outer-Inner parallelization corresponds to the implementation in Chapter 3. This includes the dead/active path control and the synchronization phase in Step 5, Step 8 and Step 9 that are related to the outer parallelization approach; additionally, Step 1 and Step 4 solve the scenario cluster subproblems in parallel following an inner parallelization approach. However, the main contribution of the current procedure is *path splitting phase.* The execution thread assignation, see Section 1.3 , is set by the modeler in *Level b*, but the path root nodes are not defined in a static way (as opposed to the previous Outer-Inner BFC in Algorithm 3.4). The procedure starts by defining a pure inner execution, then if the number of current paths is lower than the number chosen by the modeler, each path is split when branching a binary variable. Thus, the path root node definition is based on the scenario cluster submodel solution, so it is every path reassignment.

Figure 5.7 shows an example of the Outer-Inner PH-BFC dynamic path assignment, where four threads are available and the user decides to use four execution paths. The root node is solved by four task threads in an inner parallelization strategy, i.e. $(1 \times 4 \times 1)$, supposing a single auxiliary thread per optimization engine call. Variables that already satisfy the NAC are jumped, $x_1 = 0$, $x_2 = 1$ and $x_3 = 0$, and the branching of variable x_4 is split in two paths by reassigning root node coordinates, therefore, an Outer-Inner PH-BFC environment is created, $(2 \times 2 \times 1)$. Then, if variable x_5 does not satisfy the NAC, both paths are again split to define a total number of four paths executing a pure outer strategy $(4 \times 1 \times 1)$. The four paths are already defined based on the obtained solutions and follow the steps described in Figure 3.4.

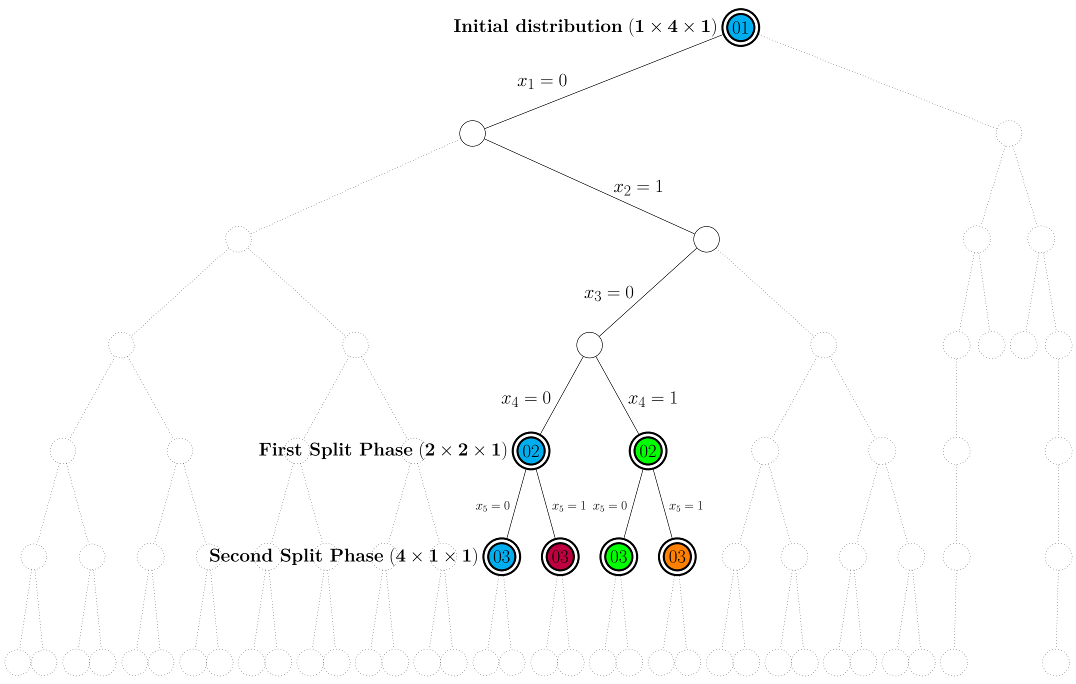
Initial distribution $(1 \times 4 \times 1)$
01
$x_1 = 0$
$x_2 = 1$
$x_3 = 0$
$x_4 = 0$
$x_4 = 1$
First Split Phase $(2 \times 2 \times 1)$
02
02
$x_5 = 0$
$x_5 = 1$
$x_5 = 0$
$x_5 = 1$
Second Split Phase $(4 \times 1 \times 1)$
03
03
03
03

Figure 5.7: Outer-Inner PH-BFC with dynamic four-path assignment example

5.7 Conclusions

The conclusions drawn from the parallelization of the BFC algorithm provide a better understanding of the algorithm performance. This Parallel Computing based additional knowledge can be applied to improve the serial BFC in Stochastic Optimization terms. Therefore, the interaction between both scientific areas creates a constant feedback that can again help to improve the parallelization.

The *Dynamically-guided and stage-ordered* Branch-and-Fix Coordination (D-BFC) and its derived matheuristics (H-BFCs) improve the serial Branch-and-Fix Coordination algorithm to obtain an exact solution of larger scale problems and faster solutions in medium-scale problems.

The improvement is based on two aspects: *stage-wise model ordering* and *dynamic node branching*. The first is drawn from the inner parallelization conclusions, where the execution time in each node is reduced by solving several subproblems in parallel; the stage perspective model ordering provides structural

Algorithm 5.3: Dynamic path assignment Outer-Inner PH-BFC

```
Level a:  (Declaring optimization and MPI variables) [All task threads].
          Set execution thread distribution ($a \times b \times c$) GLOBAL START.
Level b:  (Definition of the global environment) [All task threads].
          Begin a pure inner parallel execution ($1 \times (a \cdot b) \times c$). Set $number_{paths} = 1$
Step 0:   (Initializations)
          Set algorithm type by $d \in \{2, 3\}$, see Table 5.3.
Step 1:   (Root node)
          Level c: Secondary MPI communication.   [All task threads]
          The main thread gathers the solution and the solution value
          of all submodels.
Steps 2 and 3:   (Next node) and (Branching)
Path splitting phase
          If $number_{paths} < a$, then split the current path and update
          $number_{paths} := 2 \cdot number_{paths}$. Previous main thread continues
          execution and one task thread becomes main thread and goes to Step 0.
          Level c: All paths gather $number_{paths}$.   [All main threads]
Step 4:   (Candidate TNF)
          Level c: Secondary MPI communication.   [All task threads]
          The main thread gathers the solution and the solution value
          of all submodels.
          If a feasible solution of DEM is obtained, then update $z^{DEM}_{path}$.
          Go to Synchronization phase.
Step 5:   (Integer TNF models)
          At the end of Step 5, update $z^{DEM}_{path}$, if all nodes have been visited
          at the $\mathcal{BFT}_{path}$, then set $dead_{path} = 1$ (dead), else set $dead_{path} = 0$ (active).
          Go to Synchronization phase.
Step 6:   (Branch pruning).
Step 7:   (Backward to previous node)
          If all the nodes have been visited at the $\mathcal{BFT}_{path}$, then set $dead_{path} = 1$ (dead).
          Go to Synchronization phase.
Steps 8 and 9:   (Prune checking) and (Opposite branching)
Synchronization phase
          Level c:  (All paths gather $z^{DEM}_{path}$ and $dead_{path}$) [All main threads].
          Set $z^{DEM} = min(z^{DEM}, min_{path}\{z^{DEM}_{path}\})$.
          If all $dead_{path} = 1$ (dead) then
              Level d:  (Finish MPI environmment) [All task threads].
              GLOBAL END.
          else
              Level c:  (Variable branching exchange) [All main threads].
              Dead paths are reassigned by splitting active path $\mathcal{BFT}_{path}$.
              All paths update root node $\overline{\mathcal{N}}_{path}$.
          Paths where $dead_{path} = 0$ continue branching, go to Step 8.
          Paths where $dead_{path} = 1$ restart algorithm, go to Step 1.
```

information to the optimization engine by time-wise dependency ordering. The second is drawn from the outer parallelization conclusions, where the number of visited nodes is reduced by splitting the branching tree and by obtaining early tight bounds; the dynamic node branching guides the executions towards a feasible solution based on cluster submodel solutions as opposed to the static branching criterion presented in BFC.

A series of D-BFC based matheuristics are presented, namely H1-BFC, H2-BFC and H3-BFC, based on the relaxation of some steps of the serial algorithm in order to solve large and very large-scale problems. The matheuristic to be used is related to the problem size by increasingly relaxing exact algorithm steps but at the same time facilitating feasible solution being obtained. Algorithm H1-BFC is based on the exact D-BFC algorithm but does not solve the MIP model (3.10) to obtain z^{TNF}_{MIP}, therefore it is not an exact algorithm and is intended to solve large-scale problems. Algorithm H2-BFC additionally relaxes the solving of the submodels (3.8) to obtain z^{TNF}_{fc} and, more importantly, performs an incomplete backward branching, i.e., only previously branched variables are opposite-branched, in order to solve very large-scale problems. Algorithm H3-BFC is intended to prevent memory overflows in very large-scale problems and big data by stopping the execution at the first feasible solution found.

The computational experience first analyses the effect of the stage-wise variable ordering, where both algorithms obtain the optimal solution for the medium-scale problems included in Testbed 2. The low optimality gap and small elapsed time obtained by the matheuristic H2-BFC when solving the medium scale problems in Testbed 2 is significant. Finally, the large-scale problems included in Testbed 3 are solved by the H2-BFC and H3-BFC matheuristic algorithms and compared with the results obtained by the SDP algorithm, showing that the H-BFC algorithms obtain better solutions than the SDP algorithm and for 12 out of 18 of the instances the elapsed time is competitive with respect to CPLEX.

Finally, a new perspective for the combined Outer-Inner parallelization of the matheuristic H2-BFC algorithm is described. The new paradigm is based on the dynamic branching improvement presented by D-BFC that enables the *dynamic path assignment* Outer-Inner PH-BFC, where the path root nodes are cluster solution based. The new scheme is intended to obtain a feasible solution for very large-scale problems.

Chapter **6**

Conclusions and future research

Σα βγεις στον πηγαιμό για την Ιθάκη,	When you set sail for Ithaca,
να εύχεσαι νάναι μακρύς ο δρόμος,	wish for the road to be long,
γεμάτος περιπέτειες, γεμάτος γνώσεις.	full of adventures, full of knowledge.

Constantine P. Cavafy, an extract of the *Ithaca* poem.

6.1 Conclusions

This work studies the potential of the joint conception of Parallel Computing and Stochastic Optimization, where decomposition algorithms provide a suitable common environment. Parallel Computing not only allows a serial algorithm to be executed in parallel, but also hybrid algorithms can be designed where different aspects are simultaneously analysed. Thus, by working in parallel the knowledge of the algorithm behaviour is significantly deeper and Stochastic Optimization based improvements can be added to the serial algorithm. Simultaneously, research on Stochastic Optimization improves the Parallel Computing efficiency by rethinking the model, by decomposing the model based on mathematical optimization properties and by dividing the tasks among threads in a model based way. The most important conclusions of the work are as summarized below.

Distributed memory parallelization by MPI is a very powerful tool for reducing the execution time when solving several mixed 0-1 optimization problems. In order to increase the parallel computing performance, measured by the speed-up and the efficiency, a balanced work-load among threads and a reduced communication are the two key aspects.

Additionally, it is possible to combine the modeller-defined distributed memory parallelization with the optimization engine internal shared memory parallelization.

A broad computational experience has been performed to analyse the marginal effect of each parallelization when using CPLEX as an optimization engine. The results show that the best thread assignation depends on the number and the size of the testbed problems. Optimization engine level parallelization is preferred, in general, when solving medium or large-scale problems. However, the marginal effect of adding more auxiliary threads to CPLEX are shown to be decreasing and, therefore, after a certain amount it is preferable to solve more problems simultaneously by using distributed memory parallelization. The most efficient combination is testbed and available computational resources dependent.

All things considered, an efficient way has been introduced for parallel solving of several small and medium-scale mixed 0-1 optimization problems. Interestingly, this environment can be used for parallel subproblem solving of decomposition algorithms, i.e., a inner parallelization strategy.

The Multipath Branch & Bound algorithm is intended to solve medium-scale problems by adding parallel computing features to the Branch & Bound algorithm. For that purpose, the B&B tree is branched in parallel by using multiple paths, allowing nodes to be visited simultaneously, earlier pruning of branches by gathering intermediate path solutions and reassigning dead paths. In general, this simultaneous executions can be used for creating hybrid environments where parallel-conceived algorithm paths work with different roles or tasks, i.e., a outer parallelization strategy.

The Multipath Branch & Bound algorithm reduces the elapsed time of the corresponding serial version of the Branch & Bound algorithm but it is only capable of solving small-scale problems, since the amount of nodes that are needed to be visited significantly increases when considering more binary variables. In summary, parallel computing alone is not capable of compensating the size enlargement. Consequently, Mathematical Optimization based improvements are needed, in cooperation with Parallel Computing, in order to solve large and very large-scale problems.

The BFC methodology obtains risk neutral optimal solutions to medium-scale and large-scale multistage stochastic mixed 0-1 problems, where (exogenous) uncertainty is represented by nonsymmetric scenario trees. The superiority of the serial version of the decomposition algorithm BFC over plain use of a state-of-the-art MIP solver, is based on the stage perspective of the problem structure by the break stage scenario clustering and the coordinated branching of the subproblems. Parallel computing versions of the BFC algorithm are presented, referred to as P-BFC algorithms.

The inner level parallelization of the BFC algorithm, i.e. Inner P-BFC, performs, when possible, steps of an BFC execution in parallel, and therefore it is internal parallelization. MIP submodels attached to the set of scenario clusters are created by the modeler-defined break stage, say t^*. Since the cluster submodels are independent, they are optimized in parallel, as well as the integer TNF submodels to satisfy the explicit NAC. An extensive computational experience shows that the elapsed time is usually one order of magnitude smaller than the time required by plain use of CPLEX, even where the optimizer alone fails to solve the problem. The savings in elapsed time obtained by using 16 paths rather than a single one in P3 and P4 are 71.99% and 61.49%, respectively. Additionally, the most efficient configuration of break stage, t^*, number of task threads and number of auxiliary threads for the optimization engine internal parallelization, is problem dependent.

The outer level parallelization of the BFC algorithm, i.e. Outer P-BFC, defines a path as combinations of a set of 0-1 variables as initial condition and an iterative execution status gathering by global communication. The results of using the outer parallelization show a significant reduction of the elapsed time that is consistent when increasing the number of available threads.

Comparing the Inner P-BFC and the Outer P-BFC algorithms, we can conclude that the first is more efficient when solving problems that visit few nodes and/or the parallelizable subproblem solving consumes the major part of an iteration time. On the other hand, the Outer P-BFC is more intensive in thread use and, therefore, more efficient when the number of nodes to be visited is large or the non-parallelizable problem solving is very significant in elapsed time terms. Additionally, the number of task threads to be used in the inner approach is bounded by the number of cluster subproblems, whereas in the outer approach that number is bounded by the number of binary variables to be branched, which is very frequently much larger than the number of cluster subproblems.

However, the most efficient parallelization of the BFC algorithm consists of using inner parallelization on the paths resulting from outer parallelization, i.e. a hybrid parallel algorithm referred to as Outer-Inner BFC. The computational experience show that the marginal effect of adding more threads to the inner or the outer approach is model dependent and varies with the number of available threads. The elapsed time can be several orders of magnitude smaller than the time required by the serial version of the algorithm, specially for break stage $t^* = 1$ if the computer resources allow it.

An extension of the SDP methodology has been presented including some refinements in the algorithm to solve the production planning problem used as a testbed. The SDP obtains fast quasi-optimal solutions for large and very large-scale multistage stochastic mixed 0-1 problems, where the (exogenous) uncertainty is represented by nonsymmetric scenario trees.

The inner level parallelization of the SDP algorithm, i.e. Inner P-SDP, solves the variations of the optimization of MIP subproblems in parallel, which are attached to the set of subtrees for each modeler-driven stage. Since the variations of the subproblems for each stage are independent from each other, they are optimized in parallel following a synchronized/asynchronized depending on the linking variable dependences. Based on an extensive computational experience the proposed approach obtains quasi-optimal solutions for large and very large-scale problems. The elapsed time required by the inner parallelization is up to one order of magnitude smaller than that of the serial version of the algorithm, where efficiency is up to 90% when using 12 threads, and the performance depends on computer resources availability.

The outer level parallelization of the SDP algorithm, i.e. Outer P-SDP, defines a path as a combination of alternative solutions of the first stage of the Front-to-Back and a different perturbation performed during the Back-to-Front phase. Additionally paths interchange information about the incumbent solution at the end of each Front-to-Back execution. Regarding the large-scale and very large-scale problems included in the testbed, the Outer P-SDP obtains a slightly better solution value than the serial SDP/Inner P-SDP, but at the price of a sometimes higher computing time.

The Parallel Computing based additional knowledge of the BFC algorithm can be used to improve the serial BFC in stochastic optimization terms. The *Dynamically-guided and stage-ordered Branch-and-Fix Coordination* (D-BFC) and its derived matHeuristics (H-BFCs) obtain an exact solution of larger scale problems and faster solutions in medium-scale problems. The improvements are based on stage-wise model ordering and dynamic node branching. The first is drawn from the Inner P-BFC conclusions, where the gain is obtained by reducing the execution time of each node subproblem solving. The second is drawn from the Outer P-BFC conclusions, where the gain is obtained by obtaining early tight bounds.

A series of D-BFC based matheuristics are presented, namely H1-BFC, H2-BFC and H3-BFC, by increasingly relaxing exact algorithm steps but at the same time facilitating the intention of a feasible solution. The computational experience shows the large-scale problems included in Testbed 3 are solved by the H2-BFC and H3-BFC matheuristic algorithms. Compared to the serial SDP algorithm, the H-BFC algorithms obtain better solutions than the SDP algorithm and for 12 out of 18 of the instances the elapsed time is competitive with respect to CPLEX.

Finally, a new scheme to obtain a feasible solution for very large-scale problems is presented. Based on the dynamic branching improvement of the D-BFC algorithm, a *dynamic path assignment* Outer-Inner PH-BFC is proposed. The main contribution corresponds to a dynamic definition and reassignment of paths according to cluster subproblem solutions.

6.2 Future research

The parallelization of the PD-BFC and PH-BFCs could lead to very large scale problems being solved in case of the outer parallelization and an additional performance improvement when considering the combined outer-inner version.

Moreover, a key aspect for future research corresponds to the decomposition of model (3.10), botteleneck step of PD-BFC, which could be achieved by using Nested Benders, Nested BFC, Lagrangian Decomposition among other decomposition algorithms, including their corresponding inner or outer parallel implementations.

Hybrid environment based on the algorithms presented in this memoir could also lead very large scale problems being solved. Among them, the following algorithms are in order: SDP inside BFC for getting an initial tight bound, BFC inside SDP for solving medium scale subproblems and, more interestingly, an Outer SDP-BFC parallelization where some paths can be oriented to branch the binary variable tree and other paths to search for a better solutions perturbing the obtained incumbent.

Other parallel hardware infrastructures could open new research directions, such as Graphic Processing Units (GPUs) where the parallelization can be massive by using a significantly larger number of threads. On the other hand, each of these units has a significantly lower calculation capability than the corresponding CPUs. Therefore, the design of massively decomposition algorithms is needed to exploit the GPU paralellization capabilities.

As a further direction for future research we are planning to extend the parallel versions of the SDP and BFC algorithms to consider risk averse strategies as opposed to the risk neutral one already considered in this work. An effective expansion of these algorithms consists of allowing risk averse strategies that require cross scenarios constraints is not a trivial task, see in Rockafellar and Uryasev [2000], Fabian *et al.* [2010], Aranburu *et al.* [2012], Shapiro *et al.* [2013], Escudero *et al.* [2014], among others, some types of risk averse strategies that does not require cross scenario constraints and, then, it is easier to be implemented in decomposition algorithms. However, we favour the strategy introduced in Escudero *et al.* [2014] for the reasons that there have been presented. An additional piece of research consists of expanding the algorithms to allow a mixture of exogenous and endogenous uncertainties in the main parameters.

A final piece of research consists of applying inner and outer parallelization paradigms to solve nonlinear stochastic problems, particularly in energy planning as a key industrial sector.

Appendix A

Message Passing Interface syntax

The following sections will describe the layout groups described in Section 2.2 of a general message-passing parallel program by using Message Passing Interface (MPI).

Section A.1 describes the MPI variable declaration. Section A.2 introduces the starting functions of the MPI environment. Section A.3 presents the thread grouping functions. Section A.4 describes the message-passing communication possibilities and functions. Section A.5 presents the finishing functions of the MPI environment. Section A.6 introduces a specific example code.

A.1 Declaring MPI variables

The content of this section may not appear in every MPI code since a big part of its functions have as input or output common `C++` variables.

`MPI_Group` and `MPI_Comm` are among the most frequent MPI variables. The first one creates *groups* of thread whereas the second one creates *communicators*, that is, a group of threads plus a context of communication, in other words, a collection of threads that can send messages to each other.

By default, a *communicator* called `MPI_COMM_WORLD` is created at the beginning of each program; it consists of all the threads at the execution environment. It is very useful for global communications but sometimes it is necessary to consider also the communication among threads of a subgroup. That is the case of the code that we have developed.

As it is shown in Section A.6 the following MPI variables are declared:

```
MPI_Group  orig_group,new_group;
```

```
MPI_Comm   new_comm;
```

For other MPI variables see Pacheco [1996] and Snir *et al.* [1995]

A.2 Beginning of a MPI environment

`MPI_Init` corresponds to the first MPI function that must be executed on a program. It allows the system to use the MPI library and therefore its features. It must be called just once and no MPI functions can be called before this.

```
MPI_Init(&argc,&argv);
```

A.3 Identifying and grouping threads

In order to have a functional point-to-point communication environment it is essential to identify each thread with a *rank* number, so that the sender and the receiver can be easily identified. Additionally, the *rank* allows to work on a Single Program Multiple Data (SPMD) paradigm; it implies the execution of different tasks in different threads with a single program. Schematically:

```
if  (rank == 0) DO X
else if (rank == 1) DO Y
```

To do so, the MPI library provides the `MPI_Comm_rank` function:

```
 MPI_Comm_rank(MPI_COMM_WORLD,&original_rank);
```

The first argument defines the *communicator* and the corresponding *rank* of the thread is stored on the second argument. Note that in general a thread has a different *rank* for each *communicator* that is involved on.

Frequently, the tasks to be executed are divided among the available threads. Function `MPI_Comm_size` allows to determine the number of threads in a communicator (stored on the second argument):

```
 MPI_Comm_size(MPI_COMM_WORLD,&original_size);
```

These sentences are the key to perform global communication. However, the parallelization strategy presented in Section 2.4 needs to consider subgroups of threads. For that purpose, let us consider the case where the number of threads is bigger than the number of submodels to solve, that is `nmodel < original_size` in Section A.6. It would mean that some threads would not actually solve any subproblem. In order to simplify the thread of gathering the final solutions, a new group can be created by considering only the active threads. The steps are as follows:

Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group. The handle is stored on the second argument.

```
 MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

Form a new group as a subset of global group using MPI_Group_incl.

```
if (nmodel < original_size) {
 MPI_Group_incl(orig_group, nmodel, ranks1, &new_group);
} else {
 MPI_Group_incl(orig_group, original_size, ranks2, &new_group);
}
```

Where ranks1 is an array of nmodel elements containing the *ranks* of the threads to be part of the new group. The handle of the new group is stored on &new_group. On the other hand, ranks2 contains the ranks of the MPI_COMM_WORLD, so notice that if nmodel $\geq$ original_size the new group will correspond to the existing MPI_COMM_WORLD group.

Let us create a new communicator for the new group using MPI_Comm_create, so that the new communicator &new_comm is created with the threads new_group extracted from communicator MPI_COMM_WORLD.

```
 MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
```

A.4 Communication functions

Message-passing communication is the core of a MPI environment. The basic functions to develop such an exchange correspond to point-to-point communication are: MPI_Send and MPI_Recv. The first function sends a message to a designated thread, whereas the second receives a message from a thread. As pointed out in Pacheco [1996], in order for the message to be successfully communicated, the system must append some information to the data that the application program wishes to transmit. This information is called the **envelope of a MPI message**. It will contain the following basic information:

- The rank of the receiver
- The rank of the sender

- A tag
- A communicator

Every message-passing function defines as arguments the previous aspects. In this context global communication functions transfer information among threads inside a *communicator*. As example let us consider the specific functions used in Section A.6 code: `MPI_Gatherv` and `MPI_Allgatherv`.

Observe that each submodel is assigned to a thread in the parallel computing strategy. After the solving task is performed, the value of the objective function is gathered at the submodel ranked 0. The corresponding function is defined as follows:

```
MPI_Gatherv(&zq_loc[0], assignment[pid], MPI_INT,
   &zq[0],assignment, inicial, MPI_INT, 0, new_comm);
```

where `MPI_Gatherv` is used to gather information of type vector in a specific thread (for scalar values, use `MPI_Gather`). The first three arguments define the vector to be sent, that is, a vector stored from `&zq_loc[0]` of length `assignment[pid]` and of type `MPI_INT` (in other words, a vector of integer values). The thread ranked 0 at the `new_comm` communicator receives a vector of type `MPI_INT` from each thread of the same communicator. The size of the vectors is defined at the vector `assignment` and their position at the new vector `&zq[0]` is established by the vector `inicial`.

Additionally, if the application demands all threads to have access to a gathered information, `MPI_Allgatherv` should be used. This function is an extension of `MPI_Gatherv`, since the same structure is repeated for every thread (in other words, the gather vector will be stored in every thread). Notice that the only difference at argument level is that this function does not ask to define the receiver rank. This extension to all group threads by using the `All` prefix is valid for practically every communication function (for specific information see Snir *et al.* [1995]). Applied to the previous example the corresponding piece of code would be (not included in Section A.6):

```
MPI_Allgatherv(&zq_loc[0], assignment[pid], MPI_INT,
   &zq[0],assignment, inicial, MPI_INT, new_comm);
```

Finally, a useful group of functions allows the user to simultaneously communicate and operate with the variables under consideration. This family of functions is packed at `MPI_Reduce` and combines values from all threads. In this particular case, all threads (therefore `MPI_Allreduce`) store at the `ncols` variable the value obtained by taking the maximum value (`MPI_MAX`) among the local `ncols_max_loc` variables.

```
MPI_Allreduce(&ncols_max_loc,&ncols, 1, MPI_INT, MPI_MAX,new_comm);
```

Additional operations can be executed with `MPI_Reduce`, see Table A.1.

Table A.1: Operation options of the *Reduce* function

`C` **function**	**Function purpose**
`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_SUM`	Sum
`MPI_PROD`	Product
`MPI_LAND`	Logical AND
`MPI_LOR`	Logical OR
`MPI_MAXLOC`	Maximum and location
`MPI_MINLOC`	Minimum and location

It is important to notice that the global communication functions (in this case `MPI_Gatherv` and `MPI_Reduce`) have a blocking nature (in other words, the program execution will be suspended until the message buffer is safe in every thread involved at the communication). It means that these functions can be used for both transferring data and for synchronising processes. Observe that in the proposed implementation it could have a negative bottleneck effect since if the work-load is not balanced a thread can be waiting until the slowest one finishes the execution of assigned tasks. On the other hand, it allows not to overwrite solutions before being sent to the coordinator thread.

Additionally the user can force a global synchronization phase on a specific point by including function `MPI_Barrier(comm)`, so that it is blocked until every thread at the *communicator* `comm` has executed it.

The MPI library provides a rich variety of functions for message-passing tasks, such as `MPI_Broadcast` for sending a variable from a root node to the rest of the communicator, or `MPI_Scatter` that sends a part of a vector at every member of the communicator, the opposite of `MPI_Gather` . For further information, see Snir *et al.* [1995].

A.5 Finishing the MPI environment

`MPI_Finalize` is the last MPI function that must be executed on a program. It cleans up unfinished tasks and closes the interaction with the MPI library. It must be called just once and no MPI functions can be called after this.

```
MPI_Finalize();
```

A.6 Example code

The *example_mpi.cpp* code, see below, is an example code that includes a general layout of a message-passing parallel program by using Message Passing Interface (MPI), for a complete program that can be compiled and linked see Aldasoro *et al.* [2012]. (`C++` keywords are shown in blue, comments in green and MPI elements in red).

```
int main(int argc, char **argv){
// DECLARING GLOBAL AND MPI VARIABLES
int imod,i,j,loc,pid,npr,ncols,ncols_max_loc=0;
int assignment[nmodel],inicial[nmodel],nmodel=44,
    assignmentX0[nmodel],iniX0[nmodel],threadsCplex=2;
double zq[nmodel];
MPI_Group  orig_group,new_group;    //MPI groups
MPI_Comm   new_comm;    //MPI communicator

// DEFINITION OF THE GLOBAL ENVIRONMENT
MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&original_size); // Num. threads
    MPI_Comm_rank(MPI_COMM_WORLD,&original_rank); // Who am I?

//NEW COMMUNICATOR FOR ACTIVE THREADS
int ranks1[nmodel]; int *ranks2;  ranks2=new int[original_size];
for(i=0;i<nmodel;i++) {ranks1[i]=i;}
for(i=0;i<original_size;i++) {ranks2[i]=i;}
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
if (nmodel < original_size){
    MPI_Group_incl(orig_group, nmodel, ranks1, &new_group);
    } else {
    MPI_Group_incl(orig_group, original_size, ranks2, &new_group);}
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Group_rank (new_group, &pid);
MPI_Group_size (new_group, &npr);

// ASSIGNING WORK LOAD
for(i=0;i<npr;i++) assignment[i]=int(nmodel/npr);
for(i=0;i<(nmodel-npr*int(nmodel/npr));i++)
    assignment[i]=assignment[i]+1;        inicial[0]=0;
for(i=1;i<npr;i++) inicial[i]=inicial[i-1]+assignment[i-1];

// CREATING THE COIN-OR/CPLEX MODELS
OsiCpxSolverInterface *sol1;
sol1=new OsiCpxSolverInterface[assignment[pid]];
for(loc=0;loc<assignment[pid];loc++){
     CPXENVptr env = sol1[loc].getEnvironmentPtr();
     CPXsetintparam(env, CPX_PARAM_THREADS, threadsCplex);}
for(loc=0;loc<assignment[pid];loc++) {
    imod=inicial[pid]+loc;      const char* final;
    final=""; char buffer[33]; final=itoa(imod+1, buffer,10);
    char model[80];     strcpy (model,"Cluster");
    strcat (model,final);     puts (model);
    sol1[loc].setObjSense(1);         sol1[loc].readMps(model);
    if(sol1[loc].getNumCols() > ncols_max_loc)
        ncols_max_loc = sol1[loc].getNumCols();}

// DECLARING LOCAL VARIABLES
MPI_Allreduce(&ncols_max_loc,&ncols,1,MPI_INT,MPI_MAX,new_comm);
double *zq_loc;   zq_loc=new double[assignment[pid]];
double **x0;x0=new double*[nmodel];
     for(i=0;i<nmodel;i++) x0[i]=new double [ncols];
double *x0_loc;   x0_loc=new double[assignment[pid]*ncols];
double *x0vector;   x0vector=new double[nmodel*ncols];
     for(i=0;i<npr;i++) {iniX0[i]=inicial[i]*ncols;
      assignmentX0[i]=assignment[i]*ncols;}

// SOLVE ASSIGNED MODELS
for(loc=0;loc<assignment[pid];loc++){
     imod=inicial[pid]+loc;
     sol1[loc].branchAndBound(); zq_loc[loc]=sol1[loc].getObjValue();
     for(j=0;j<sol1[loc].getNumCols();j++)
         x0_loc[loc*ncols+j]=sol1[loc].getColSolution()[j];}

// GLOBAL MPI COMMUNICATION (GATHERING SOLVE INFORMATION)
MPI_Gatherv(&zq_loc[0],assignment[pid],MPI_DOUBLE,&zq[0],
     assignment,inicial,MPI_DOUBLE,0,new_comm);
MPI_Gatherv(x0_loc,assignmentX0[pid],MPI_DOUBLE,x0vector,
     assignmentX0,iniX0,MPI_DOUBLE,0,new_comm);

MPI_Finalize(); // EXECUTION ENDS IN ALL PROCESSORS
return 0;}
```

example_mpi.cpp

Appendix B

P-BFC programming codes

This appendix describes the C++ programming codes included in the attached CD and used in the computational experience of Chapter 3.

Common program files:

BBcutsCplex.cpp, external function that solves the scenario cluster submodels (3.1) and (3.8).

BBcutsCplexzf.cpp, external function that solves the integer TNF model (3.10).

bintotal.cpp, external function that tests if binary relaxed variables satisfy integrality constraints.

itoa.h, header file created by Lukás Chmela that converts integers to ASCII code.

nabin.cpp, external function that tests if binary variables of cluster problems satisfy integrality and non-anticipativity constraints.

nacont.cpp, external function that tests if continuous variables of cluster problems satisfy non-anticipativity constraints.

nirea.cpp, external function that creates random variables using a non computer dependent seed.

param3asim.cpp, external function that builds the full DEM in compact representation.

param4asim.cpp, external function that builds the DEM for each scenario cluster submodel.

pm.h, header file that includes the necessary header files of COIN-OR and CPLEX optimization engines.

vectors.cpp, external function that builds the necessary auxiliary dimensional vectors for creating the full DEM.

vectorsasim.cpp, external function that builds the necessary auxiliary dimensional vectors for creating the full DEM when the scenario tree is non symmetric.

Serial BFC:

maintotalCplex.cpp, main program that executes the serial Branch-and-Fix Coordination algorithm.

Inner P-BFC:

mainbfcmpi.cpp main program that executes the Inner Parallel Branch-and-Fix Coordination algorithm.

Outer-Inner P-BFC:

cabecera.cpp, external function that prints the execution configuration.

hilos.cpp, external function that manages the synchronization phase of the Outer P-BFC algorithm.

mainbfcmpipath.cpp main program that executes the combined Outer-Inner Parallel Branch-and-Fix Coordination algorithm.

Appendix C

P-SDP programming codes

This appendix describes the `C` programming codes included in the attached CD and used in the computational experience of Chapter 4.

Common program files:

backtofront_central.c, external function that performs the Back-to-Front phase in an intermediate stage.

backtofront_final.c, external function that performs the Back-to-Front phase in the last stage.

fronttoback.c, external function that performs the Front-to-Back phase in any stage.

generar_parametros_inciertos.c, external function that creates the random variables and problem dimension vectors.

initial_solve.c, external function that performs the Front-to-Back phase in any stage of the initial iteration.

itoa.h, header file created by Lukás Chmela that converts integers to ASCII code.

nirea.c, external function that creates random variables using a non computer dependent seed.

reference_levels.c, external function that perturbs the solution obtained in the Front-to-Back phase in order to create new reference levels.

sdp.h, header file that defines the global variables.

Serial SDP files:

parametros_serie.c, external function that defines the variable values needed to define a serial SDP execution.

results_fronttoback_serial.c, external function that stores and prints the solution obtained at the end of an Front-to-Back phase.

results_initial_solve_serial.c, external function that stores and prints the solution obtained at the end of an Front-to-Back phase in the initial iteration.

sdp.c, main program that executes the serial version of the SDP algorithm.

Inner P-SDP files:

gather_mupi_global.c, external function that gathers the μ and π vectors using MPI.

gather_stock.c, external function that gathers the subproblem solutions obtained in a Front-to-Back phase using MPI.

gth_btf_central_i.c, external function that gathers the subproblem solutions obtained in a central stage of a Back-to-Front phase using MPI.

gth_btf_final_i.c, external function that gathers the subproblem solutions obtained in the last stage of a Back-to-Front phase using MPI.

gth_mu_pi.c, external function that gathers the μ and π vectors inside a task thread subgroup using MPI.

inner.h, header file that defines the global variables related to the inner paradigm implementation.

mainsdp_inner.c, main program that executes the inner version of the SDP algorithm.

parametros_i.c, external function that defines the variable values needed to define an inner SDP execution.

res_ftb_i.c, external function that gathers using MPI the solution obtained at the end of an Front-to-Back phase.

res_initial_i.c, external function that gathers using MPI the solution obtained at the end of an Front-to-Back phase in the initial iteration.

Outer P-SDP files:

gather_btf_final_outer.c, external function that gathers the subproblem solutions obtained in the last stage of a Back-to-Front phase using MPI.

gather_ultima_solucion.c, external function that gathers the subproblem solutions obtained in a Front-to-Back phase using MPI.

mainsdp_outer.c, main program that executes the outer version of the SDP algorithm.

outer.h, header file that defines the global variables related to the outer paradigm implementation.

parametros_outer.c, external function that defines the variable values needed to define an outer SDP execution.

results_fronttoback_outer.c, external function that gathers using MPI the solution obtained at the end of an Front-to-Back phase.

results_initial_solve_outer.c, external function that gathers using MPI the solution obtained at the end of an Front-to-Back phase in the initial iteration.

solution_pool_outer.c, external function that executes the solution pool at the initial iteration and scatters the corresponding results to each main thread.

solution_pool_outer_ftb.c, external function that executes the solution pool at the end of the Front-to-Back phase and chooses the solution to be taken.

Appendix D

Predoctoral training and visibility

The fulfilled predoctoral training in Operations Research and Parallel Computing comprehends master program courses, workshops and an international visiting research student fellowship:

- **Visiting research student fellowship**. *University of Edinburgh & Edinburgh Parallel Computing Center. September-December 2013.* (Supervisor: Prof. Jacek Gondzio).
- **Master in Statistics and Operations Research**. *Universitat Politècnica de Catalunya & Universitat de Barcelona. 2011.* (Full program).
- **Master in Advanced Computer Systems**. *University of the Basque Country UPV/EHU. 2012.* (Fulfilled course: Programming Parallel Systems).
- **Master in Mathematical Modelling, Statistics and Computing**. *University of the Basque Country UPV/EHU. 2012.* (Fulfilled course: Models in Logistics).
- **Master in High Performance Computing**. *University of Edinburgh & Edinburgh Parallel Computing Center. 2013.* (Courses attended as auditing student: HPC Architectures, Message-Passing Programming, Parallel Numerical Algorithms, Parallel Programming Languages, Programming Skills, Threaded Programming).
- **Summer School in Programming and Tuning Massively Parallel Systems**. *Barcelona Supercomputing Center. July 2014.*

Throughout the detailed training process several supercomputer facilities have been used as practical environments:

- **ARINA**, from *University of the Basque Country, UPV/EHU.* Used for distributed programming and threaded programming training as well as the computational experience of this work and related publications.

- **HECToR**, from *Edinburgh Parallel Computing Centre (EPCC)*, British academic national supercomputer. Used for distributed programming, threaded programming, debugging tools and profiling tools training.

- **MinoTauro**, from *Barcelona Supercomputing Center, BSC.* Used for GPU programming training.

Certain main results of this memory have lead to several publications:

Scientific papers:

- U. Aldasoro, L.F. Escudero, M. Merino, J.F. Monge and G. Pérez. *On parallelization of a Stochastic Dynamic Programming algorithm for solving large-scale mixed 0-1 problems under uncertainty* **Submitted**, 2014.

- U. Aldasoro, L.F. Escudero, M. Merino and G. Pérez. *An algorithmic framework for solving large-scale multistage stochastic mixed 0-1 problems with nonsymmetric scenario trees. Part II: Parallelization* **Computers and Operations Research (COR)** 40:2950-2960, 2013.

Working papers:

- U. Aldasoro, M.A. Garín, M. Merino and G. Pérez. *Generating cluster submodels from a multistage stochastic mixed integer optimization model using break stage* **Working paper series Biltoki** DT.2013.02. `https://addi.ehu.es/handle/10810/10446`

- U. Aldasoro, M.A. Garín, M. Merino and G. Pérez. *MPI parallel programming of mixed integer optimization problems using CPLEX within COIN-OR* **Working paper series Biltoki** DT.2012.01. `https://addi.ehu.es/handle/10810/7274`

Furthermore, the main results of this memory have also been presented in several national and international meetings:

- **Euro Mini Conference on Stochastic Programming and Energy Applications (ECSP2014).** *P-SDP, a parallel stochastic dynamic programming algorithm for solving large-scale mixed 0-1 problems under uncertainty.* (U. Aldasoro, L. F. Escudero, M. Merino and G. Pérez). Paris (France). September 2014.

- **20th Conference of the International Federation of Operation Research Societies (IFORS 2014).** *On parallelizing decomposition algorithms for solving stochastic multistage mixed 0-1 problems.* (U. Aldasoro, L. F. Escudero, M. Merino and G. Pérez). Barcelona (Spain). July 2014.

- **Society for Industrial and Applied Mathematics (SIAM 2014).** *Parallelized Branch-and-Fix-Coordination algorithm (P-BFC) for solving large-scale multistage stochastic mixed 0-1 problems* (L. F. Escudero, U. Aldasoro, M. Merino and G. Pérez). San Diego, California (USA). May 2014.

- **Computational Management Science Conference (CMS2014).** *P-BFC, Parallelized Branch-and-Fix Coordination algorithm for solving large-scale multistage mixed 0-1 problems* . (G. Pérez, U. Aldasoro, L. F. Escudero and M. Merino). Lisboa (Portugal). May 2014.

- **IV Jornadas de Investigación de la Facultad de Ciencia y Tecnología UPV/EHU.** *Stochastic optimization and parallelization (Poster)* (U. Aldasoro, L. Aranburu, L. F. Escudero, M.A. Garín, M. Merino, G. Pérez, C. Pizarro and A. Unzueta). Leioa (Spain). February 2014.

- **Computational Linear Algebra and Optimization for the Digital Economy.** *Parallel stochastic optimization* (Poster). (U. Aldasoro, L. Aranburu, L. F. Escudero, M.A. Garín, M. Merino, G. Pérez, C. Pizarro and A. Unzueta). International Center of Mathematical Sciences, Edinburgh (UK). November 2013.

- **XXXIV Congreso Nacional de Estadística e Investigación Operativa (SEIO).** *On parallelizing BFC: An exact algorithm for solving large-scale stochastic multistage mixed 0-1 optimization problems.* (G. Pérez, U. Aldasoro, L. F. Escudero and M. Merino). Castellón (Spain). September 2013.

- **XIII International Conference on Stochastic Programming (ICSP2013).** *Parallelized Branch-and-Fix Coordination (P-BFC) for solving large-scale multistage mixed 0-1 problems.* (G. Pérez, M. Merino, L. F. Escudero, M.A. Garín and U. Aldasoro). University of Bergamo (Italy). July 2013.

- **25th European Conference on Operational Research (Euro XXV).** *Parallel computing via break stage scenario clustering for multistage stochastic* *programming.* (G. Pérez, M. Merino, L. F. Escudero, M.A. Garín and U. Aldasoro). Vilnius (Lithuania). July 2012.

- **Seminars on Optimization under Uncertainty and Risk Management**. *Nonsymmetric decomposition algorithm BFC for multistage stochastic MPI 0-1 models.* (G. Pérez, U. Aldasoro, L. F. Escudero, M.A. Garín and M. Merino). URJC, Móstoles (Spain). June 2012.

- **XXXIII Congreso Nacional de Estadística e Investigación Operativa (SEIO)**. *MPI parallel programming for solving a large number of mixed integer optimization problems using CPLEX within COIN-OR.* (U. Aldasoro, L. F. Escudero, M. Merino and G. Pérez). Madrid (Spain). April 2012.

- **Computational Management Science Conference (CMS2012)**. *Parallel algorithm for solving multistage stochastic mixed 0-1 problems using BFC-MS with CPLEX within COIN-OR and MPI.* (G. Pérez, U. Aldasoro, L. F. Escudero and M. Merino). London (United Kingdom). April 2012.

- **III Jornadas de Investigación de la Facultad de Ciencia y Tecnología UPV/EHU.** *Mixed integer stochastic optimization: algorithms and applications (Poster)* (U. Aldasoro, L. Aranburu, L. F. Escudero, M.A. Garín, M. Merino, G. Pérez and A. Unzueta). Leioa (Spain). February 2012.

Glossary

active path	28
asynchronized execution	5
auxiliary thread	4
bandwidth	7
Branch-and-Fix tree	44
break stage	40
candidate TNF	45
common variables	45
compact representation	12
coordinator task thread	5
CPU time	23
dead path	28
efficiency	24
elapsed time	23
global communication	5
inner parallelization	26
integer TNF	45
latency	7
linking variables	13
main thread	4
matheuristic	1
NAC based cluster set	43
non-anticipativity constraints	9
outer parallelization	37
path	4
primary communication	5
reference level	79
scenario	10
scenario cluster decomposition	41
scenario group	11
secondary communication	5
solution pool	86
speedup	24
splitting variable representation	13
stage	10
subordinated task thread	5
synchronized execution	5
task thread	4
task thread subgroup	5
tertiary communication	5
thread	4
thread assignation	4
thread rank	16
Twin Node Family	45
twin nodes	45

Nomenclature

$\langle i \rangle$	index of variable i in the current cluster. 110
$\mathcal{A}_g$	set of scenario group g and its ancestors. 11
$\tilde{\mathcal{A}}_\ell$	set consisting of nodes in $\mathcal{A}_\ell$, such that their variables have nonzero elements in constraints associated with the nodes in the immediate successor subproblems to leaf node ℓ. 77
A	constraint matrix of the 0-1 variables. 13
a	vector of the objective function coefficients of the 0-1 variables. 13
$a(g)$	immediate ancestor node of node g. 76
$\mathcal{BFT}$	B&B tree for optimizing the scenario cluster submodels in a coordinated way. 44
$\mathcal{BFT}^c$	*B&B* tree associated with scenario cluster c. 44
$\mathcal{BFT}_{path}$	*B&B* tree associated with the corresponding *path*. 54
B	constraint matrix of the continuous variables. 13
b	vector of the objective function coefficients of the continuous variables. 13
$\mathcal{C}$	set of scenario clusters. 40
$\mathcal{C}^g$	set of cluster that have scenario group g in common. 43
$\mathcal{C}_i$	set of cluster that have variable x_i in common. 111
C	number of scenario clusters. 40
c	scenario cluster index. 40
$dens$	constraint matrix density (in %). 23

d	Branch-and-Fix Coordination algorithm version. 113
$E_{th}\%$	efficiency of the execution when using th threads, in percentage. 24
$E_{th}^{top}\%$	top efficiency that could be achieved given the model scenario tree using th threads. 103
$\mathcal{F}$	set of families. 45
$F_r(\overline{V}_{a(r)}^{z})$	solution value of the subproblem related to root node r and linking variable values $\overline{V}_{a(r)}^{z}$ in reference level z. 83
$\mathcal{G}$	set of scenario groups. 11
$\mathcal{G}^c$	set of scenario groups in cluster c. 41
$\mathcal{G}_t$	set of scenario groups in stage t. 11
$\mathcal{G}_t^c$	set of scenario groups for cluster $c \in \mathcal{C}$ in stage $t \in \mathcal{T}$. 41
g	scenario group index. 11
$GAP_{t^*}^0$	optimality gap (in %) of the solution value $z_{t^*}^0$. 44
GAP_{LP}	optimality gap (in %) of the LP relaxation solution value z_{LP}. 62
$GG\%$	optimality gap (in %) of the incumbent solution value with respect to the MIP solver solution value. 96
h	vector of *rhs* coefficients. 12
$\mathcal{I}$	set of indices of the variables in vector x_t^c, for $c \in \mathcal{C}, t \in \mathcal{T}$. 44
$\mathcal{I}_x$	set of indices of the x variables. 109
$\mathcal{I}_y$	set of indices of the y variables. 109
$IG^{inner}\%$	improvement gap (in %) of the inner parallelization solution value with respect to the serial solution value. 102
$IG^{outer}\%$	improvement gap (in %) of the outer parallelization solution value with respect to the serial solution value. 102
i	variable index. 49
$\mathcal{J}_f$	Twin Node Family. 45
$\mathcal{L}_r$	set of leaf nodes in the subtree whose root node is r. 77
ℓ	leaf node index. 77
$\lambda_\ell(\cdot)$	convex curve that gives the expected solution value of the immediate successor subproblems of leaf node ℓ. 78

λ_ℓ	piecewise linear EFV approximation of $\lambda_\ell(\cdot)$. 83
$\mathcal{M}_r$	set of nodes in the subtree rooted at node r. 77
m	total number of constraints. 23
μ_ℓ^z	parameter to define the piecewise linear function for leaf node ℓ and reference level z. 79
$\overline{\mathcal{N}}_{path}$	root node of the corresponding *path*. 54
n	total number of variables. 94
nx	total number of binary variables. 23
ny	total number of continuous variables. 23
nel	number of nonzero coefficients in the constraint matrix. 23
$niter$	number of full iterations performed. 96
$nprob$	number of subproblems in which the DEM is decomposed. 95
nz	number of generated reference levels. 96
n^n	number of twin nodes explored. 66
n^{TNF}	number of integer TNF encountered by the algorithm. 66
n^c	number of cluster submodels solved. 121
Ω	set of scenarios. 11
Ω^g	set of scenarios in scenario group g. 11
Ω_c	set of scenarios in cluster c. 40
ω	scenario index. 11
$OG\%$	optimality gap (in %) achieved for the MIP problem. 95
$\mathcal{P}$	set of periods along the time horizon. 76
p	period index. 78
π_ℓ^z	parameter vector to define the piecewise linear function for leaf node ℓ and reference level z. 79
$\mathcal{Q}^c$	set of active nodes in $\mathcal{BFT}^c$ for cluster c. 44
$\mathcal{R}^t$	set of root nodes of the subtrees related to stage t. 77
$\mathcal{R}_{th}^{FtB}$	set of root nodes of the subtrees related to thread th in the FtB scheme. 86
$\mathcal{R}_{th}^{BtF}$	set of root nodes of the subtrees related to thread th in the BtF scheme. 86
(rhs)	right hand side vector. 8
r	root node index. 77
$\mathcal{S}_\ell$	set of immediate successor nodes to leaf node ℓ. 77

S_{th}	speedup of the execution when using th threads. 24
S_{th}^{top}	top speedup that could be achieved given the model scenario tree using th threads. 103
σ_i	more frequent binary solution for variable i. 110
$\mathcal{T}$	set of stages along the time horizon. 11
$\mathcal{T}_1$	set of ancestor stages to break stage t^* (including itself). 43
$\mathcal{T}_2$	set of sucessor stages to break stage t^*, i.e., $\mathcal{T}_2 = \mathcal{T} - \mathcal{T}_1$. 43
$\mathcal{T}^t$	set of ancestor stages to stage t (including itself) whose variables have nonzero elements in the constraints of stage $t \in \mathcal{T}$. 11
T	last stage in the time horizon, such that $T = \|\mathcal{T}\|$. 11
$t(g)$	stage to which scenario group g belongs. 76
t	stage index. 11
t^*	break stage. 40
t^{DEM}	elapsed time for obtaining the solution value of the original DEM. 66
t_{LP}	elapsed time for obtaining the solution value of the LP relaxation of the original DEM. 95
th	thread index. 25
$\overline{V}_\ell^z$	solution of the variables in vectors $x-$ and $y-$ related to ancestor scenario groups to leaf node ℓ and reference level z. 83
w^ω	likelihood associated to scenario ω. 12
w_g	likelihood associated with scenario group g. 12
$\xi_r^{z'}$	perturbation on the linking constraint system related to root node r and reference level z'. 81
x	vector of binary variables. 12
y	vector of continuous variables. 12
$\hat{y}$	upper bound vector of variables in vector y. 12
z^{DEM}	solution value of the original DEM. 62
z_{path}^{DEM}	solution value of the original DEM obtained in the corresponding $path$. 57
z_{LP}	solution value of the LP relaxation of the original DEM. 62
$z_{t^*}^0$	lower bound of the solution value of the original DEM obtained at the root node for break stage t^*. 44

$\mathcal{Z}_\ell$	set of indices of currently active reference levels related to leaf node ℓ. 80
z	reference level index. 79

Acronyms

B&B	Branch & Bound algorithm. 15
BD	Benders Decomposition. 8
BFC	Branch-and-Fix Coordination multistage algorithm. 39
BtF	Back-to-Front scheme in the SDP algorithm. 80
COIN-OR	COmputational INfrastructure for Operations Research. 7
CPU	Central Processing Unit. 3
D-BFC	Dynamically-guided and stage-ordered Branch-and-Fix Coordination. 107
DEM	Deterministic Equivalent Mode. 8
DG	Dynamically-Guided branching. 110
EFV	Expected Future Value. 76
FtB	Front-to-Back scheme in the SDP algorithm. 80
GPU	Graphic Processing Unit. 3
H-BFC	D-BFC based matheuristic algorithms. 107
IB	Incomplete Backward branching. 113
LP	Linear Programming. 7
MIP	Mixed Integer Programming. 39
MPI	Message Passing Interface. 7
MPP	Massively Parallel multiProcessor. 6
NAC	Non-Anticipativity Constraints. 9
P-BFC	Parallel Branch-and-Fix Coordination. 39
P-SDP	Parallel Stochastic Dynamic Programming. 75
PC	Parallel Computing. 1

PH-BFC	Parallel H-BFC matheuristic algorithm. 122
RAM	Random-Access Memory. 7
SDP	Stochastic Dynamic Programming. 75
SF	Stop at First feasible solution. 113
SIO	Stochastic Integer Optimization. 8
SO	Stochastic Optimization. 1
SPMD	Single Program Multiple Data. 16
UPV/EHU	Universidad del País Vasco/Euskal Herriko Unibertsitatea. 7

Bibliography

S. Ahmed, A.J. King, and G. Parija. A multi-stage stochastic integer programming approach for capacity expansion under uncertainty. *Journal of Global Optimization*, 26(1):3–24, 2003.

T. Al-Khamis and R. M'Hallah. A two-stage stochastic programming model for the parallel machine scheduling problem with machine capacity. *Computers & Operations Research*, 38:1747–1759, 2011.

U. Aldasoro, M.A. Garín, M. Merino, and G. Pérez. MPI parallel programming of mixed integer optimization problems using CPLEX within COIN-OR. *Working paper series Biltoki*, 2012. `https://addi.ehu.es/handle/10810/7274`.

U. Aldasoro, L.F. Escudero, M. Merino, and G. Pérez. An algorithmic framework for solving large-scale multistage stochastic mixed 0-1 problems with nonsymmetric scenario trees. Part II: Parallelization. *Computers & Operations Research*, 40:2950–2960, 2013.

U. Aldasoro, M.A. Garín, M. Merino, and G. Pérez. Generating cluster submodels from a multistage stochastic mixed integer optimization model using break stage. *Working paper series Biltoki*, 2013. `https://addi.ehu.es/handle/10810/10416`.

U. Aldasoro, L.F. Escudero, M. Merino, J.F Monge, and G. Pérez. On Parallelization of a Stochastic Dynamic Programming algorithm for solving largescale mixed 0-1 problems under uncertainty. *Submitted*, 2014.

A. Alonso-Ayuso, L.F. Escudero, M.A. Garín, M.T. Ortuño, and G. Pérez. An approach for strategic supply chain planning based on stochastic 0-1 programming. *Journal of Global Optimization*, 26:97–124, 2003.

A. Alonso-Ayuso, L.F. Escudero, and M.T. Ortuño. BFC, a Branch-and-Fix Coordination algorithmic framework for solving some types of stochastic pure and mixed 0-1 programs. *European Journal of Operational Research* , 151:503–509, 2003.

A. Alonso-Ayuso, L.F. Escudero, M.A. Garín, M.T. Ortuño, and G. Pérez. On the product selection and plant dimensioning problem under uncertainty. *Journal of Global Optimization*, 33:307–318, 2005.

A. Alonso-Ayuso, L.F. Escudero, and M. T. Ortuño. On modelling planning under uncertainty in manufacturing. *Statistics and Operations Research Transactions*, 31:109–150, 2007.

A. Alonso-Ayuso, L.F. Escudero, and C. Pizarro. *Introduction to Stochastic Programming.* Ed. Dikinson, 2009.

L. Aranburu, L.F. Escudero, A. Garín, and G. Pérez. *Stochastic models for optimizing immunization strategies in fixed-income security portfolios under some sources of uncertainty*, pages 173–220. World Scientific Publisher, "Applications to finance, energy planning and logistics", 2012.

E.M.L. Beale. On minimizing a convex function subject to linear inequalities. *Journal of the Royal Statistical Society, Ser. B*, 17:173–184, 1955.

G. Belvaux and L.A. Wolsey. Modelling practical lot-sizing problems as mixed-integer programs. *Management Science*, 47:993–1007, 2001.

J.F. Benders. Partitioning procedures for solving mixed variables programming problems. *Management Science*, 1:238–252, 1962.

P. Beraldi, L. Grandinetti, R. Musmanno, and C. Trik. Parallel algorithms to solve two-stage stochastic linear programs with robustness constraints. *Parallel Computing*, 26:1889–1908, 2000.

J.R. Birge and F.V. Louveaux. *Introduction to Stochastic Programming. 2nd edition.* Springer, 2011.

J.R. Birge, C.J. Donohue, D.F. Holmes, and O.G. Svintsitski. A parallel implementation of the nested decomposition algorithm for multistage stochastic linear programs. *Mathematical Programming*, 75:327–352, 1996.

J.R. Birge. Stochastic programming computation and applications. *INFORMS Journal of Computing*, 9:111–133, 1997.

J. Blomval and P.O. Lindberg. A Riccati-based primal interior point solver for multistage stochastic programming. *European Journal of Operational Research*, 143:452–461, 2002.

J. Blomval. A multistage stochastic programming algorithm suitable for parallel computing. *Parallel Computing*, 29:431–445, 2003.

C.C. Carøe and R. Schultz. Dual decomposition in stochastic integer programming. *Operations Research Letters*, 24:37–45, 1999.

C.C. Carøe and J. Tind. L-shaped decomposition of two-stage stochastic programs with integer recourse. *Mathematical Programming*, 83:451–464, 1998.

A. Charnes and W.W. Cooper. Chance-constrained programming. *Management Science*, 5:73–79, 1959.

COIN-OR. COmputational INfrastructure for Operations Research. *INFORMS Foundation*. `http://www.coin-or.org/`.

CPLEX. IBM ILOG CPLEX Optimizer
. `www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`.

M.P. Cristobal, L.F. Escudero, and J.F. Monge. On Stochastic Dynamic Programming for solving large-scale production planning problems under uncertainty. *Computers & Operations Research*, 36:2418–2428, 2009.

D.E. Culler, A. Gupta, and J.P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

G.B. Dantzig and P.W. Glynn. Parallel processors for planning under uncertainty. *Annals of Operations Research*, 22:1–21, 1990.

G.B. Dantzig. Linear programming under uncertainty. *Management Science*, 1:197–206, 1955.

B.H. Dias, M.A. Tomin, A.L.M. Mercato, T.P. Ramos, R.B.S. Brandi, I.Ch. da Silva jr., and J.A.P.Filho. Parallel computing applied to stochastic dynamic programming for long term operation planning of hydrothermal power systems. *European Journal of Operational Research*, 229:212–222, 2013.

C. Dillenberger, L.F. Escudero, A. Wollensak, and Z. Wu. On practical resource-allocation for production planning and scheduling with period overlapping setups. *European Journal of Operational Research*, 75:275–286, 1994.

L.F. Escudero, J.L. de la Fuente, C. García, and F.J. Prieto. A parallel computation approach for solving multistage stochastic network problems. *Annals of Operations Research*, 90:1–21, 1999.

L.F. Escudero, M.A. Garín, M. Merino, and G. Pérez. A two-stage stochastic integer programming approach as a mixture of Branch-and-Fix Coordination and Benders Decomposition schemes. *Annals of Operations Research*, 152:395–420, 2007.

L.F. Escudero, M.A. Garín, M. Merino, and G. Pérez. On BFC-MSMIP: an exact branch-and-fix coordination approach for solving multistage stochastic mixed 0-1 problems. *TOP*, 17:96–122, 2009.

L.F. Escudero, M.A. Garín, M. Merino, and G. Pérez. A general algorithm for solving two-stage stochastic mixed-integer programs. *Computers & Operations Research*, 36:2590–2600, 2009.

L.F. Escudero, M.A. Garín, M. Merino, and G. Pérez. On an exact algorithm for solving large-scale two-stage stochastic mixed-integer problems: theoretical and computational aspects. *European Journal of Operational Research*, 204:105–116, 2010.

L.F. Escudero, M.A. Garín, M. Merino, and G. Pérez. On BFC-MSMIP strategies for scenario cluster partitioning and Twin Node Families branching selection and bounding for multi-stage stochastic mixed integer programming. *Computers & Operations Research*, 37:738–753, 2010.

L.F. Escudero, M.A. Garín, M. Merino, and G. Pérez. An algorithmic framework for solving large-scale multistage stochastic mixed 0-1 problems with nonsymmetric scenario trees. *Computers & Operations Research*, 39:1133–1144, 2012.

L.F. Escudero, M.A. Garín, M. Merino, and G. Pérez. On solving strong multistage nonsymmetric stochastic mixed 0-1 problems. In H.-J. Luth D. Katte and K. Schemdders, editors, *Operations Research Proceedings, selected papers presented to OR2011, Zurich, Switzeland, 506-514*. Springer, 2012.

L.F. Escudero, M.A. Garín, G. Pérez, and A. Unzueta. Scenario cluster decomposition of the lagrangian dual in two-stage stochastic mixed 0-1 optimization. *Computers & Operations Research*, 40:362–377, 2013.

L.F. Escudero, J.F. Monge, D. Romero Morales, and J. Wang. Expected future value decomposition based bid price generation for large-scale network revenue management. *Transportation Science*, 47:181–197, 2013.

L.F. Escudero, M.A. Garín, M. Merino, and G. Pérez. On time stochastic dominance introduced by mixed integer-linear recourse in multistage stochastic programs. *In second revision*, 2014.

L.F. Escudero. On a mixture of the Fix-and-Relax Coordination and Lagrangean Substitution schemes for multistage stochastic mixed integer programming. *TOP*, 17:5–29, 2009.

C. Fabian, G. Mitra, D. Roman, and V. Zverovich. An enhanced model for portfolio choice with SSD criteria: a constructive approach. *Mathematical Programming*, 108:541–569, 2010.

E. Fragniere, J. Gondzio, and J.P. Vial. Building and solving large-scale stochastic programs on an affordable distributed computing system. *Annals of Operatiosn Research*, 99:167–187, 2000.

J. Gondzio and R. Kouwenberg. High-performance computing for asset liability management. *Operations Research*, 49:879–891, 2001.

R. Hemmecke and R. Schultz. Decomposition methods for two-stage stochastic integer programs. In S.O. Krumke M. Grötschel and J. Rambau, editors, *Online Optimization of Large Scale Systems*, pages 601–622. Springer, 2001.

J.L Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2003.

L. Hong, L. Zhong-Hua, and Ch. Xue-Bin. Parallel computing for dynamic asset allocation based on the stochastic programming. In *Information Engineering (ICIE), 2010 WASE International Conference, vol. 2, p.p. 172-176*, 2010.

P. Kall and S.W. Wallace. *Stochastic Programming.* John Wiley, 1994.

J. Krarup and O. Bilde. Plant location, set covering and economic lot size: An 0 (mn)-algorithm for structured problems. In L. Collatz, G. Meinardus, and W. Wetterling, editors, *Numerische Methoden bei Optimierungsaufgaben Band 3, p.p. 155-180.* Birkhäuser Basel, 1977.

G. Laporte and F.V. Louveaux. An integer L-shaped algorithm for the capacitated vehicle routing problem with stochastic demands. *Operations Research*, 50:415–423, 2002.

X. Li, J. Wei, T. Li, G. Wang, and W.W.-G. Yeh. A parallel dynamic programming algorithm for multi-reservoir system optimization. *Advances in Water Resources*, 67:1 – 15, 2014.

J.T. Linderoth, A. Shapiro, and S. Wright. The empirical behavior of sampling methods for stochastic programming. *Annals of Operations Research*, 142:215–241, 2006.

J.T. Linderoth. *Topics in Parallel Integer Optimization.* PhD thesis, Georgia Intitute of Technology, Atlatna, GA, USA, 1998.

M. Lucka, I. Melichercik, and L. Halada. Application of multistage stochastic programs solved in parallel in portfolio management. *Parallel Computing*, 34:469–485, 2008.

A.J. Miller, G.L. Nemhauser, and M.W.P. Savelsbergh. On the capacitated lot-sizing and continuous 0-1 knapsack polyhedra. *European Journal of Operational Research*, 125:298–315, 2000.

J.M. Mulvey and A. Ruszczynski. A new scenario decomposition method for large-scale optimization method. *Operations Research*, 43:477–490, 1995.

G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience, 1988.

S.S. Nielsen and S.A. Zenios. Scalable parallel Benders decomposition for stochastic linear programming. *Parallel Computing*, 23:1069–1088, 1997.

P.S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers, 1996.

A. Pagès-Bernaus, G. Pérez-Valdés, and A. Tomasgard. A parallelised distributed implementation of a Branch-and-Fix Coordination algorithm. *Submitted, in second revision*, 2014.

M.V.F. Pereira and L.M.V.G. Pinto. Multistage stochastic optimization applied to energy planning. *Mathematical Programming*, 52:359–375, 1991.

G.Ch. Pflug and A. Pichler. *Multistage Stochastic Optimization.* Springer, 2014.

A. Piazza and B.K. Pagnoncelli. The optimal harvesting problem under price uncertainty. *Annals of Operations Research*, 217:425–445, 2014.

Y. Pochet and L.A. Wolsey. Solving multi-item lot-sizing problems using strong cutting planes. *Management Science*, 37(1):53–67, 1991.

R.T. Rockafellar and S. Uryasev. Optimization of conditional value-at-risk. *Journal of Risk*, 2:21–41, 2000.

R.T. Rockafellar and R.J-B Wets. Scenario and policy aggregation in optimisation under uncertainty. *Mathematics of Operations Research*, 16:119–147, 1991.

S.M. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, 1995.

A. Ruszczyinski. Parallel decomposition of multistage stochastic programming problems. *Mathematical Programming*, 58:201–228, 1993.

S. Sen and H.D. Sherali. Decomposition with branch-and-cut approaches for two stage stochastic mixed-integer programming. *Mathematical Programming, Ser. A*, 106:203–223, 2006.

A. Shapiro, W. Tekaya, J. Paulo da Costa, and M. Pereira Soares. Risk neutral and risk averse Stochastic Dual Dynamic Programming method. *European Journal of Operational Research*, 224:375–391, 2013.

J.F. Shapiro. Mathematical programming models and methods for production planning and scheduling. In *Logistics of production and inventory, vol. 4, p.p. 371-443*. North-Holland, 1993.

M. Snir, S.W. Otto, D.W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

J.P. Sousa and L.A. Wolsey. A time indexed formulation of nonpreemptive single-machine scheduling problems. *Mathematical Programming*, 54:353–367, 1992.

A. Stivala, P. J. Stuckey, M. Garcia de la Banda, M. Hermenegildo, and A. Wirth. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing*, 70(8):839 – 848, 2010.

R. van Slike and R.J-B Wets. L-shaped linear programs with application to optimal control and stochastic programming. *SIAM Journal on Applied Mathematics*, 17:638–663, 1969.

H. Vladimirou. Computational assessment of distributed decomposition methods for stochastic linear programs. *European Journal of Operational Research*, 108:653–670, 1998.

H.M. Wagner and T.M. Whitin. Dynamic Version of the Economic Lot Size Model. *Management Science*, 5:89–96, 1958.

R.J-B Wets. Stochastic programs with fixed recourse: The equivalent deterministic program. *SIAM Review*, 16:309–339, 1974.

R.J-B Wets. On the relation between stochastic and deterministic optinmization. In A. Bensoussan and J.L. Lions, editors, *Control Theory, Numerical Methods and Computer Systems Modeling, p.p. 350-361*. Springer-Verlag, 1975.

L.A. Wolsey. MIP modelling of changeovers in production planning and scheduling problems. *European Journal of Operational Research*, 99:154–165, 1997.

Z. Zhang, S. Zhang, Y. Wang, Y. Jiang, and H. Wang. Use of parallel deterministic dynamic programming and hierarchical adaptive genetic algorithm for reservoir operation optimization. *Computers & Industrial Engineering*, 65(2):310 – 321, 2013.

P.H. Zipkin. Models for design and control of stochastic, multi-item batch production systems. *Oper. Res.*, 34(1):91–104, January 1986.