COPPE
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia
UFRJ

ASYNCHRONOUS STOCHASTIC DUAL DYNAMIC PROGRAMMING ALGORITHM APPLIED TO HYDROTHERMAL COORDINATION

Felipe Dias de Rezende Machado

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadores: Carmen Lucia Tancredo Borges
André Luiz Diniz Souto Lima

Rio de Janeiro
Março de 2020

ASYNCHRONOUS STOCHASTIC DUAL DYNAMIC PROGRAMMING ALGORITHM APPLIED TO HYDROTHERMAL COORDINATION

Felipe Dias de Rezende Machado

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Orientadores: Carmen Lucia Tancredo Borges
André Luiz Diniz Souto Lima

Aprovada por: Prof. Carmen Lucia Tancredo Borges
Prof. André Luiz Diniz Souto Lima
Dr. Vitor Luiz de Matos
Prof. Laura Silvia Bahiense da Silva Leite

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2020

Machado, Felipe Dias de Rezende

Asynchronous Stochastic Dual Dynamic Programming Algorithm Applied to Hydrothermal Coordination/ Felipe Dias de Rezende Machado. – Rio de Janeiro: UFRJ/COPPE, 2020.

XI, 80 p.: il.; 29, 7cm.

Orientadores: Carmen Lucia Tancredo Borges

André Luiz Diniz Souto Lima

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2020.

Referência Bibliográfica: p. 73 – 80.

1. Stochastic Dual Dynamic Programming. 2. Parallel Computing. 3. Hydrothermal Coordination. I. Borges, Carmen Lucia Tancredo *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Acknowledgment

À minha mãe e ao meu pai por todas as lições de amor, companheirismo, amizade, e dedicação. E ao meu irmão, sempre pronto a me apoiar em tudo. Agradecimento também a minha cunhada Ana.

Agradecimentos a todos os tios e primos que não caberiam nessas páginas.

Agradeço a minha esposa Mariana por todo o carinho, paciência e compreensão, me acompanhando por todo o tempo desse trabalho, e à Cristina, Pedro e Tamires.

Aos orientadores André e Carmen por me direcionarem no caminho certo da pesquisa, ao vasto conhecimento desprendido e, o que foi muito importante, me ajudar a saber quando parar.

Aos amigos e companheiros de departamento Lilian, Cristiane, Luis Fernando, André Quadros, Roberto, Elvira, Hugo e Juan que me ajudaram a ter uma luz em diversos momentos que fiquei com problemas para avançar no trabalho.

Aos amigos e companheiros de projeto Bruno, Fábio, Pedro, Valk e Ana pela paciência e força quando desanimava.

Ao companheiros de outros departamentos Eduardo, Nícolas, Renan, Lígia e Paula pelo conhecimento e momentos compartilhados.

Aos amigos Diego, Monique, Bernardo, Renan, Giane, Caio, Ricardo, Tarcísio que aturaram minhas ausências e meu humor por tantas vezes.

Ao meu professor e mestre Antônio Cláudio (AC) que infelizmente não pode estar presente para ver a conclusão desse trabalho.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROGRAMAÇÃO DINÂMICA DUAL ESTOCÁSTICA ASSÍNCRONA APLICADA AO PROBLEMA DE COORDENAÇÃO HIDROTÉRMICA

Felipe Dias de Rezende Machado

Março/2020

Orientadores: Carmen Lucia Tancredo Borges
André Luiz Diniz Souto Lima

Programa: Engenharia Elétrica

A otimização do planejamento da geração de energia elétrica é muito importante para alcançar os custos mais baixos possíveis, contrabalanceando da melhor forma com a segurança da rede elétrica. São necessários altos recursos computacionais para resolver esse problema, que é multi-estágio, estocástico, complexo e de grande porte. Em alguns casos, o uso de paralelização torna-se imperativo. Um método amplamente utilizado para resolver problemas de planejamento energético de longo prazo é uma extensão da Programação Dinâmica Dual (PDD), denominada Programação Dinâmica Dual Estocástica (PDDE), que utiliza técnicas de amostragem para lidar com problemas da alta dimensionalidade. Neste trabalho é proposto um esquema paralelo assíncrono para a PDDE, que é capaz de superar o sincronismo intrínseco existente no método de paralelização tradicionalmente utilizado para a PDDE, e permitindo explorar melhor os recursos paralelos. Com isso, o algoritmo visa diminuir o tempo total da CPU para resolver o problema. Testes de consistência e desempenho foram aplicados para avaliar a abordagem assíncrona da PDDE e uma variante da mesma em um problema de tamanho equivalente ao sistema brasileiro real, onde se verificaram suas vantagens em relação ao método de paralelização convencionalmente utilizados.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ASYNCHRONOUS STOCHASTIC DUAL DYNAMIC PROGRAMMING ALGORITHM APPLIED TO HYDROTHERMAL COORDINATION

Felipe Dias de Rezende Machado

March/2020

Advisors: Carmen Lucia Tancredo Borges
André Luiz Diniz Souto Lima

Department: Electrical Engineering

Optimizing power generation planning is very important to achieve the lowest possible costs while keeping an adequate trade-off with electrical network security. High computational resources are required to solve this problem, which is a multistage, stochastic, complex and large scale problem. In some cases, the use of parallel schemes becomes mandatory. A widely used method to solve long-term energy planning problems is an extension of Dual Dynamic Programming (DDP) called Stochastic Dual Dynamic Programming (SDDP) which makes use of sampling techniques to be able to deal with high-dimensional state-spaces. In this work we propose an asynchronous SDDP parallel scheme capable of overcoming the intrinsic synchronism of the traditional parallel version of the SDDP method, thus allowing to better exploit the parallel resources and decrease the overall CPU time to solve the problem. Consistency and performance tests were applied to evaluate the proposed asynchronous SDDP approach and one variant of this algorithm in a problem equivalent to the real Brazilian system, where was verified advantages over using the traditional parallel schema.

Contents

List of Figures

Chapter 1

Introduction

Operation planning of large-scale hydrothermal systems is a very important and complex task, which requires a calculation process involving a large number of variables, constraints and uncertainties on a multi stage stochastic problem. In predominantly hydro systems, as for example in Brazil and Norway, it is of utmost importance to model the high uncertainty related to water inflows, which aggregates additional complexity to model and solve the problem [45] [25].

The uncertainties in mid and long term planning problems can be represented as a scenario tree. The more complex the problem is, the more dependent it becomes of computational resources to obtain a solution in acceptable time. Even though in principle it could be solved as a single large optimization problem, computational time can rapidly became prohibitive and decomposition methods like Dual Dynamic Programming (DDP) [6] or sampling based methods as Stochastic Dual Dynamic Programming (SDDP) [55] must be applied to better use the computational resources.

As presented in [55], the SDDP method does not traverse the entire scenario tree, but rather solves scenario samples in each iteration instead. Both DDP and SDDP are iterative methods that perform forward and backward passes, with a convergence criterion based on a tolerance between the difference in the upper and lower bounds for the former [6] or on a confidence interval for the latter [55]. Although asymptotic convergence of the SDDP method is guaranteed if resampling is applied [57], the convergence rate can be very slow, demanding too much time to reach acceptable results. For this reason, parallelization should be used to accelerate the standard SDDP method. However, the traditional parallel version of the SDDP algorithm [58] has an undesirable synchronization point in every time step, thus slowing down its parallel performance.

1.1 Objective and contributions of this work

In order to overcome the limitation of synchronization point at every stage, this work proposes an asynchronous SDDP algorithm that is more suitable for parallel environments. This idea was sketched in [20] but with neither a more thorough investigation nor a substantial implementation was employed. This approach is based on successful algorithms previously proposed in [62] and [9] for the DDP method in deterministic and stochastic problems, respectively, with the aim to decrease the effect of the synchronization point, therefore reducing CPU time and improving scalability.

In addition, this work aims to improve the level of scalability of parallel algorithms applied to the SDDP approach, by changing the parallelism paradigm. While traditional parallel SDDP approaches divide the sub problems to processes according to forward or backward scenarios the proposed algorithm divides the sub problems among processes by stage. This way the work load can be divided in a more efficient way, not only because the communication overhead is reduced, but also because the number of stages is usually high and using a small number of forward passes per iteration tends to be advantageous [33]. This makes the proposed approach useful both from the modeling and the computational point of view.

1.2 Relevance and application

Since it was first proposed in [55], the SDDP algorithm has been explored and improved by many researchers in different ways, as for example in [40] [57] [58] [63] [16] [30] [10]. Therefore, improving SDDP performance may have a positive impact in many applications presented by the scientific community.

In Brazil, the official long/mid term planning model NEWAVE ([46] [45] [43]) uses SDDP as its main solution algorithm. In this way, improvements in the SDDP performance allows further enhancements in the model, as for example a greater refinement in the system representation and constraints.

We note that this work modeled the hydrothermal planning problem in individual hydro plants, rather than equivalent reservoirs, [47] and also applying an accurate model of the hydro production function [19].

1.3 Arrangements

The dissertation is organized as follows:

- Chapter 2: introduces the Hydrothermal Coordination Problem, which is the context in which the algorithm was applied.

- Chapter 3: explains some parallel programming concepts that are applied in this work.
- Chapter 4: describes the Stochastic Dual Dynamic Programming approach usually applied in the literature, as well as its traditional parallel approach.
- Chapter 5: proposes an asynchronous approach to SDDP and one variant: the first one with a synchronous point at each step and the second one without any synchronous point.
- Chapter 6: presents the numerical experiments, which comprises a direct application to solve the HTC problem and an assessment of the algorithm consistency and performance.
- Chapter 7: presents the conclusions and future work.

Chapter 2

Mid/Long Hydrothermal Coordination Problem

Power generation planning comprises the determination of the best use of the generation resources. Traditionally, the problem consists in minimizing the operation cost, which is directly associated with the fuel used for thermal generation to satisfy all constraints along the planning horizon. The inclusion of hydro plants with reservoirs in the system creates an additional level of difficulty for power operation planning because water can be stored for further use. In addition, reservoirs can be spatially coupled in cascade, where the operation of upstream plants impacts the availability of water in downstream plants.

Therefore, the Hydrothermal Coordination Problem (HTC) consists of coordinating the use or storage of water, to achieve the objective of minimizing the high costs of generation of thermal plants. When controlling the stored water, some situations may occur due to uncertainty on the natural water inflows to the reservoirs (which is a random variable), leading to "right" or "wrong" decisions, as illustrated in Fig. 2.1. Such situations are associated to combinations of high or low inflows in the future with the decisions that are taken in the present: right decisions arise when water is stored today and low inflows occur in the future, or when water is stored today and high inflows occur in the future. By contrast, wrong decisions are those where water is not stored today and low inflows occur in the future, or when water is stored today and high inflows occur in the future.

One way to represent the sequence of such decisions along time is by considering a discrete time discretization combined with a discretization of the random variables, resulting in a discrete scenario tree, as shown in Fig. 2.2. Each time step will be referred to in this work as a "stage", and the uncertainty in the HTC problem is related to the inflows to the hydro plants.

Ideally, the HTC problem should be modeled considering as many components and constraints as possible, thus resulting in a better representation of reality. How-

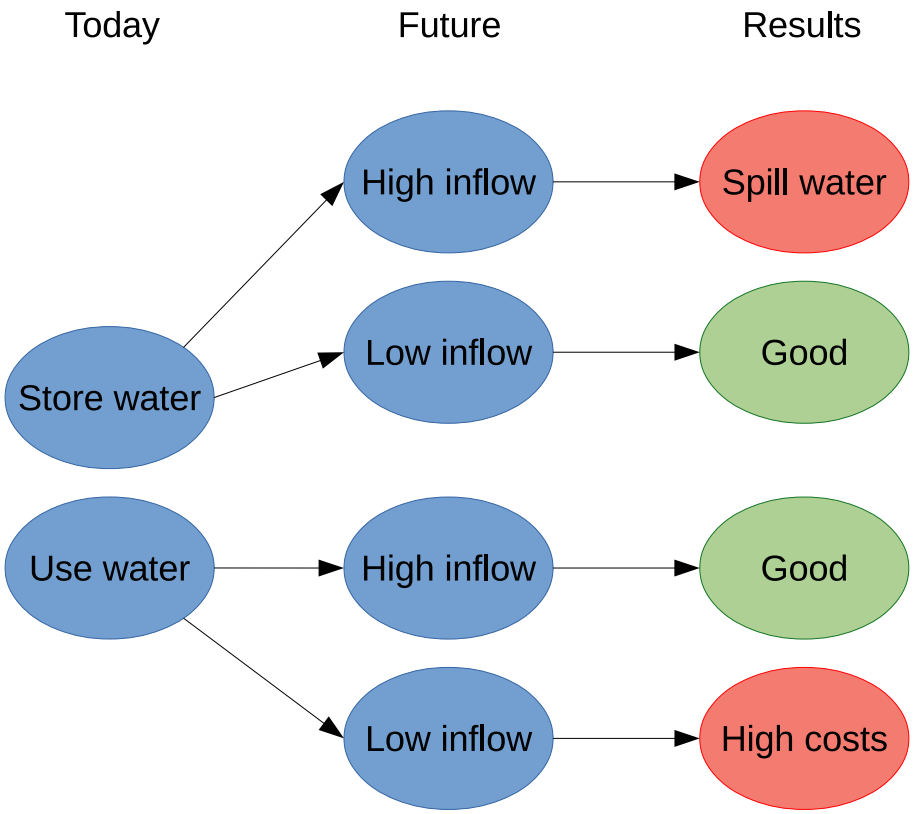

Figure 2.1: Water use decision.

ever, since this may lead to a computationally intractable problem, HTC is usually split into long, mid and short term problems, each one with an emphasis on the aspects that are most suitable for its corresponding time discretization and planning horizon [25] [45] [27] [21].

This work is related to the long and mid term planning problems, taking into account in the latter a proper coupling with a cost-to-go function provided by the former. The aim is to solve multi-stage optimization problems expressed as a linear programs (LPs), with the objective function of minimizing system operation costs and considering both linear constraints and convex piecewise linear approximations of nonlinear constraints. The variables of the problem represent major hydro and thermal plants characteristics, taking into account cascaded hydro plants along the river basins and a single node representation of the system. The following assumptions are also made:

- Each stage of the planning horizon is split into three load blocks to better represent intra-stage aspects. The consideration of load blocks allows to consider the load variation inside a stage, then improving the precision over considering one single value as average. Moreover, this approach is computationally less

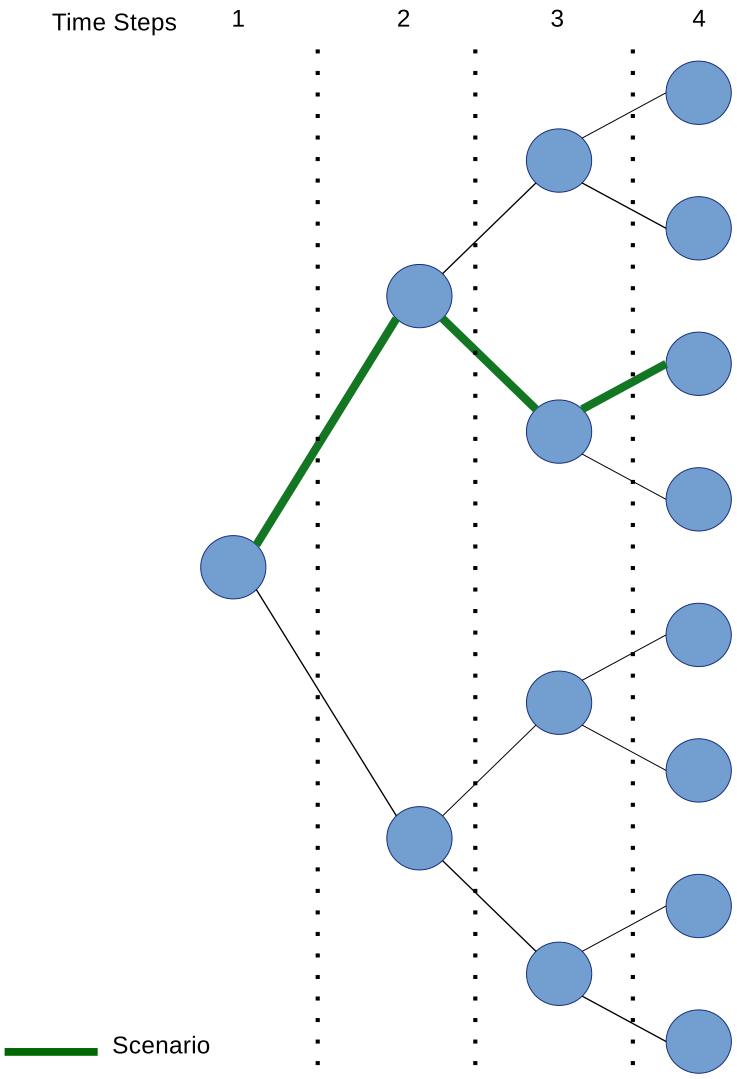

Figure 2.2: Scenario tree representation.

costly than increasing the number of stages by decreasing the duration of each time step;

- to enable complete recourse for the overall multistage stochastic problem that is decomposed into smaller subproblems, slack variables are included for all constraints with high artificial costs;

- Uncertain inflows to reservoirs are stage-wise independent.

The planning horizon contains several stages, leading to a huge scenario tree. In order to solve this problem, the sampling based approach of dual dynamic programming called stochastic dual dynamic programming (SDDP) is applied, as explained in Chapter 4.

The following sections will describe the objective function and constraints of the subproblem for each stage t and value for the random variables s in which the

problem is decomposed, as well as additional constraints related to variables bounds. In the sequel each possibility s for the values of the random variables in each stage will be denoted as "stage scenario".

2.1 Objective function

The objective function comprises thermal generation costs plus the expected value of future costs, associated to the subproblems of all subsequent nodes:

$$minimize \sum_{p=1}^{NL} \sum_{i=1}^{NT} ct_i gt_i^{t,s,p} + Q_t(v^{t,s}) \tag{2.1}$$

where:

NL = Number of load blocks;
NT = Number of thermal plants;
ct_i = Linear incremental cost of each thermal plant (piecewise linear costs can be modeled by splitting the generation range in several segments);
$gt_i^{t,s,p}$ = Generation of thermal plant on stage t, stage scenario s and load block p.

Function $Q_t(v^{t,s})$ is the so called future cost function (FCF) for stage t, which is a lower approximation of the expected value of the future costs beyond this stage, as a function of storages in the reservoirs. Such function is obtained by Benders cuts generated during the course of the DDP or SDDP solution algorithms, and is evaluated for the vector of final storages $v^{t,s}$ in the reservoirs at the corresponding stage and stage scenario. Some artificial costs are also applied for practical reasons: slack variables with high penalty costs to allow violation of some constraints, as well as small penalty values to avoid unnecessary spillage and water waste.

2.2 Load supply

A load supply constraint is applied to each stage, scenario and load block:

$$\sum_{i=1}^{NH} gh_i^{t,s,p} + \sum_{j=1}^{NT} gt_j^{t,s,p} + def^{t,s,p} = d^{t,p} \quad p = 1, ..., NL \tag{2.2}$$

where:

NL = Number of load blocks,
NH = Number of hydro plants,
NT = Number of thermal plants,
$d^{t,p}$ = Demand of load block p and stage t;
$gh_i^{t,s,p}$ = Generation of hydro plant i at stage t,
stage scenario s and load block p;
$gt_j^{t,s,p}$ = Generation of thermal plant j at stage t,
stage scenario s and load block p;
$def^{t,s,p}$= Energy deficit at stage t,
stage scenario s and load block p;
$d^{t,p}$ = Load demand of stage t and load block p.

2.3 Water balance

Water balance constraints are applied to each hydro plant i and stage t, taking into account the topological structure of upstream and downstream plants along the river courses:

$$\begin{aligned} v_i^{t,s} - v_i^{t-1,r} + \sum_{p=1}^{NL}(qt_i^{t,s,p} + qs_i^{t,s,p}) + \\ - \sum_{p=1}^{NL} \sum_{j \in \Omega_i^{up}} (qt_j^{t,s,p} + qs_j^{t,s,p}) = I_i^{t,s} \quad i = 1, ..., NH \end{aligned} \tag{2.3}$$

where:

$v_i^{t,s}$ = Storage at reservoir i at the end of stage t and scenario s;
$v_i^{t-1,r}$= Storage at reservoir i at the end of the ascendant node r
(in stage $t-1$) of scenario s;
$qt_i^{t,s,p}$= Turbined outflow of hydro plant i at stage t,
scenario s and load block p;
$qs_i^{t,s,p}$= Spilled outflow of hydro plant i stage t,
stage scenario s and load block p;
$I_i^{t,s}$ = Water inflow hydro plant i at step t,
stage scenario s;
Ω_i^{up} = Set of upstream hydro plants to hydro plant i.

Such constraints imply both a spatial coupling (due to variables $qt_j^{t,s,p}$ and $qs_j^{t,s,p}$, related to set of upstream hydro plants Ω_i^{up}) and a time coupling (due to variables $v_i^{t,s}$ and $v_i^{t-1,r}$) among variables and constraints of the problem formulation.

2.4 Hydro plant production function

The generation of the hydro plants is modeled to represent water head effects when transforming the turbined outflow into electrical energy. One approach is to apply the so-called Approximate Hydro Production Function (AHPF) proposed in [19], which consists in a piecewise linear model that also takes into account the effect of spillage. Therefore, the following inequality constraints are considered for each load block, hydro plant, stage and stage scenario:

$$\begin{aligned} gh_i^{t,s,p} \leq \gamma_{0,i,t}^{(k)} + \gamma_{v,i,t}^{(k)}(v_i^{t-1,r} + v_i^{t,s})/2 \\ +\gamma_{q,i,t}^{(k)} qt_i^{t,s,p} + \gamma_{s,i,t}^{(k)} qs_i^{t,s,p} \\ k = 1, ..., NCUT_i; i = 1, ..., NH; p = 1, ..., NL \end{aligned} \tag{2.4}$$

where:

$NCUT$= Number of linear aproximations for the hydro production function of hydro plant i,
$\gamma_{0,i,t}^{(k)}$ = independent term of cut k, hydro plant i, and stage t;
$\gamma_{v,i,t}$ = term related to reservoir storage for cut k of hydro plant i, for stage t
$\gamma_{q,i,t}^{(k)}$ = term related to turbined outflow for cut k of hydro plant i, for stage t;
$\gamma_{s,i,t}^{(k)}$ = term related to spillage for cut k of hydro plant i, for stage t.

All γ coefficients to all cuts are pre-calculated before solving the problem.

2.5 Future cost function (FCF)

In order to couple the last stage to water values given by a long term model, a so-called future cost function (FCF) is applied. It is represented by a set of linear inequalities that provide the expected value of system operating costs in the future (related to thermal generation and energy deficit) as a function of the vector of storages in the reservoirs at the end of the planning horizon.

This work considered a future cost function provided by the long term model NEWAVE [46]. However, since the official use of the NEWAVE model for the Brazilian system considers the representation of hydro plants as equivalent reservoirs (EER), conversion coefficients ρ were used to translate the value of stored energy in each EER into water values for the individual storage in each reservoir, as shown below. We note that the future cost function was obtained by the NEWAVE model with a configuration that is spatially compatible with the study cases considered in this work, where all hydro plants are inside one EER.

$$
\begin{aligned}
FCF^{T,s} \geq \pi_{0,T}^{(k)} + \sum_{j=1}^{NEER} \pi_{j,T}^{(k)} E_{eer_j}^{T,s} \\
k = 1, ..., NCUT_{FCF}; \\
E_{eer_j}^{T,s} = \sum_{i \in \Omega_j} \rho_i v_i^{T,s}
\end{aligned}
\tag{2.5}
$$

where:

T = last stage of the planning horizon;
$NCUT_{FCF}$ = Number of cuts (linear inequalities) that compose the FCF from the long term model;
$NEER$ = Number of EERs;
$\pi_{0,T}^{(k)}$ = Constant term of cut k of the FCF;
$\pi_{j,T}^{(k)}$ = Coefficient related to EER j for cut k of the FCF;
$E_{eer_j}^{T,s}$ = Energy stored in EER j, stage T and for stage scenario s;
Ω_j = Set of hydro plants associated to EER j;
ρ_i = Conversion coefficient for hydro plant i;
$v_i^{T,s}$ = Reservoir storage for hydro plant i at the end of stage T and stage scenario s;

Chapter 3

Parallel programming

This chapter aims to explain parallel programming concepts and terms to ensure a proper understanding and to avoid ambiguities in this text.

The purpose of parallel processing is to reduce computational time as well as to make some programs viable to be used in practice. A parallel scheme can be made in several ways [52], depending on computer hardware, software architecture, and operational system. Prior to writing parallel models, these characteristics must be considered. Some classifications are used to create these parallel models, such as: how data is accessed, memory storage, and the parallelism grain size.

3.1 Definitions

- CPU: Central Processing Unit. Consists on a group of *processors* that can contain some *cores*. Computers may have more than one CPU, but desktops usually have only one CPU with multi-cores.
- Node: Node is the term used to a complete computer inside a computer *cluster*.
- Cluster: Computer cluster is a group of computers working together to solve a problem.
- Task: Group of instructions to be executed by processors.
- Communication: Sending of data between tasks over shared memory or network connections.
- Thread: Lightweight processes instantiated and linked from a larger program. Executes a small portion of the program and has their program counters, instructions and even owns a part of the memory.
- Latency: Time to send data between tasks.

- Scalability: Capacity of increasing hardware resources to the complete computer architecture. It means the capacity of a parallel model to improve performance as the hardware to execute the program is increased.

3.2 Model classification

Parallel programming models use Flynn's taxonomy [24], which classifies them in two dimensions: "Instruction Stream" and "Data Stream", with two possible states: "Single" or "Multiple". As a result, four possible combinations associated with Processor Unit (PU) can be generated, as described below.

3.2.1 SISD - Single Instruction Single Data

During a clock cycle, only one stream of instruction is used on PU, using a single input of data stream, Fig. 3.1. shows a single processor computer used in small scale to specific purposes. As an example, Internet of Things (IoT) devices and computers to simple activities make use of the SISD architecture.

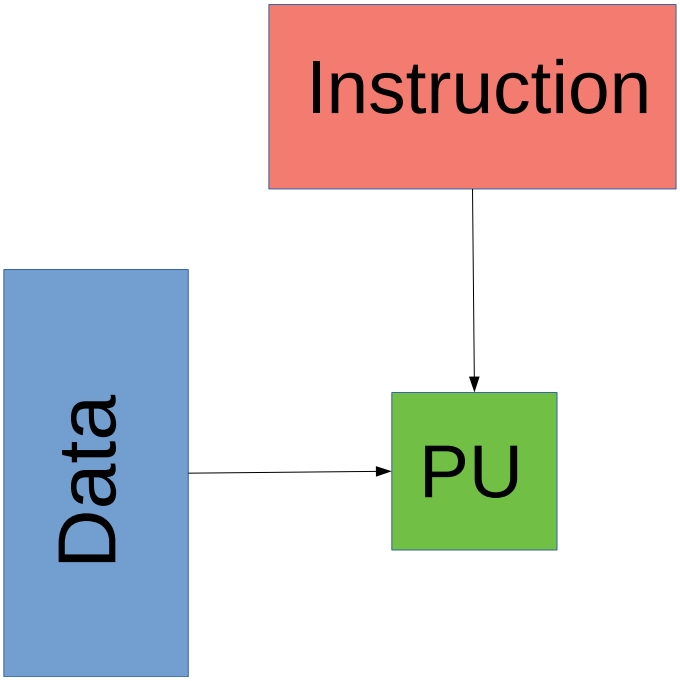

Figure 3.1: SISD - Single Instruction Single Data scheme.

3.2.2 SIMD - Single Instruction Multiple Data

During a clock cycle, only one stream of instruction is used on PU with a multiple data stream input (Fig. 3.2). It is considered as a computer with parallel processor

using vectorial data processing. This configuration is applied on common cases like image processing, which is used by Graphic Processor Units (GPU), for example.

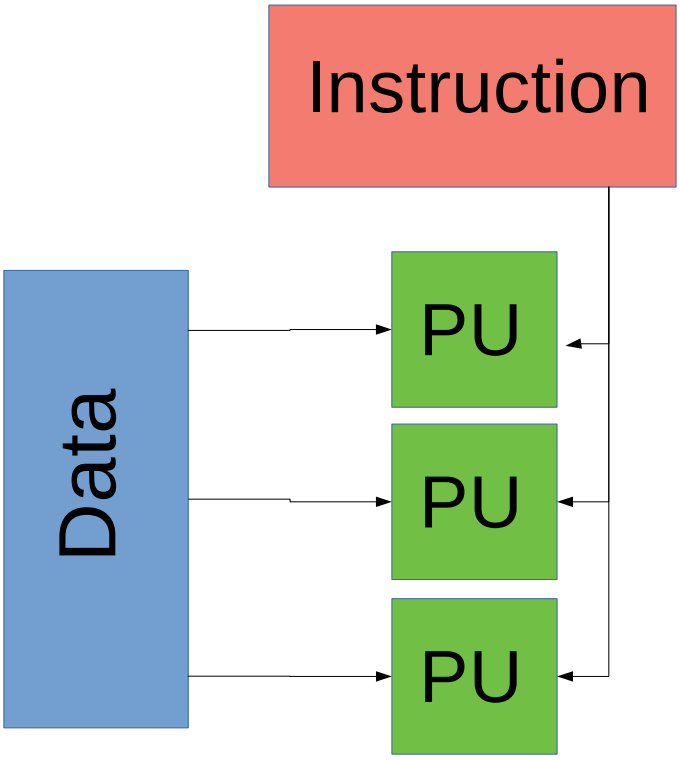

Figure 3.2: SIMD - Single Instruction Multiple Data scheme.

3.2.3 MISD - Multiple Instruction Single Data

The single data stream can be processed by a stream of multiple instructions on PU during a clock cycle, as shown in Fig. 3.3. The use of this classification is less spread than others.

3.2.4 MIMD - Multiple Instruction Multiple Data

Multiple data streams can be processed by a stream of multiple instructions during a clock cycle on a PU, as shown in Fig. 3.4. This classification disseminated to most of the current Central Processor Units (CPU).

3.3 Memory classification

Memory in parallel programming can be classified by Processor Unit access as follows:

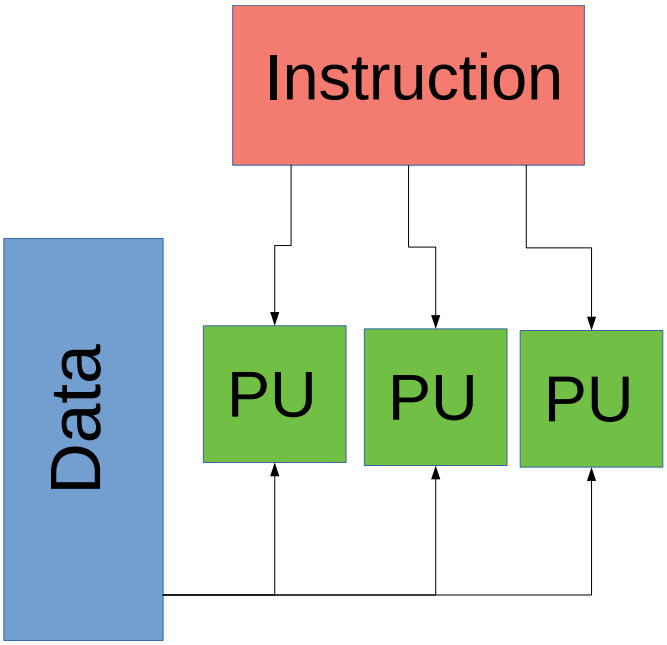

Figure 3.3: MISD - Multiple Instruction Single Data scheme.

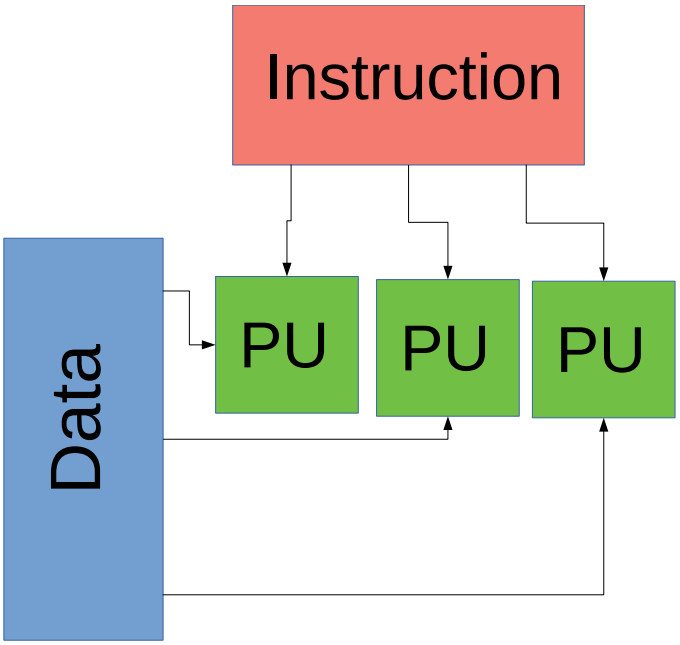

Figure 3.4: MIMD - Multiple Instruction Multiple Data scheme.

3.3.1 Shared memory

All processors have direct access to the same global memory address, and can operate independently accessing the same memory space and also do not need to communicate data. Moreover, when one processor changes data in memory, all processors have access to this modification. This behavior tends to improve performance since communication is not needed, but a layer of memory access control has to be used to prevent memory miswriting/misreading.

There are two kinds of shared memory to be considered [71]:

UMA - Uniform Memory Access

PUs can access direct shared memory with the same latency, as shown in Fig. 3.5. This kind of shared memory does not have good scalability because the increase in the number of processors also increases the number of memory access requests, and only one processor can make this access at a time.

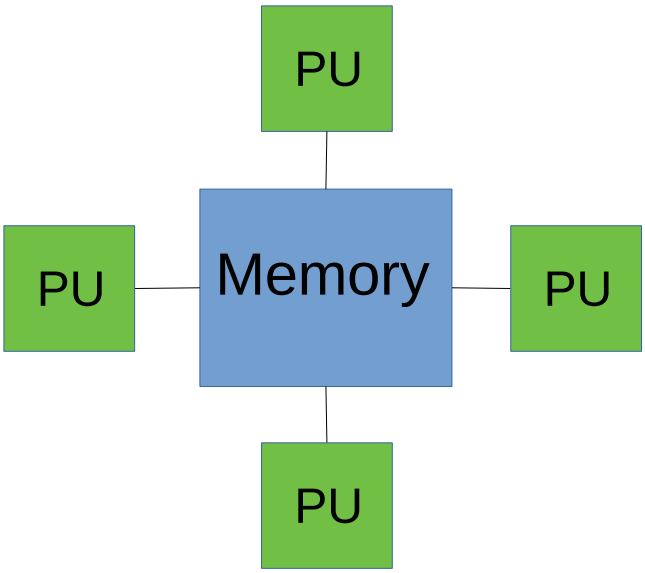

Figure 3.5: Shared memory scheme.

When a PU has a cache memory, this architecture has to reflect memory modifications to the cache in all PUs simultaneously.

NUMA - Non Uniform Memory Access

This kind of memory architecture has a global memory that is accessed by all PUs, but the latency is not the same, since this access is made through an interconnection bus that is even located in the same hardware. This scheme has a higher scalability than UMA because memories and PUs can be inserted into the same bus.

3.3.2 Distributed memory

In this memory architecture each PU maps its own memory and does not depend on other PUs, thus not leading to a global memory scheme as in Section 3.3.1. A memory change does not affect the memory of other PUs, so communication is required to reflect this change (Fig. 3.6 shows this architecture scheme). There are some similarities between this architecture and NUMA (described in Section 3.3.1),

since both have an interconnection bus between PUs. However, in the NUMA architecture the PUs maps memory into global memory, which does not apply to distributed memory.

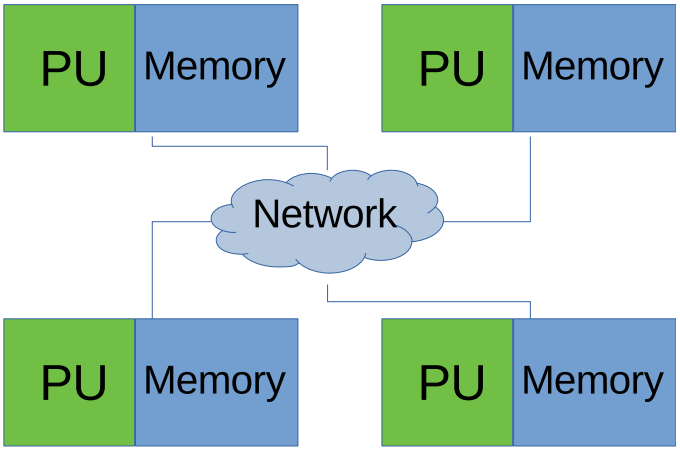

Figure 3.6: Distributed memory scheme.

This architecture has a good scalability, because an increase in the number of PUs in the network increases the available memory, since each PU has its own memory, and interconnection overload does not exist. However, this architecture requires an explicit data communication among PUs.

3.3.3 Hybrid memory

Hybrid memory consists of using shared and distributed memories simultaneously, as shown in Fig. 3.7. Shared memory is used by PUs with physically shared memory, like current CPUs or GPUs, and distributed memory is used for communicating over a network connection.

The scalability of this architecture is the greatest among the mentioned architectures, because inserting a computer in the network is relatively easier than inserting another PU on a computer. However, memory management is an issue, because it must be done manually by the programmer.

Due to its high scalability and flexibility of use, this memory architecture was employed to all implemented algorithms, then using shared or distributed memory as convenient. If the processing occurs in the same processor shared memory was used, otherwise distributed memory was applied by communicating data

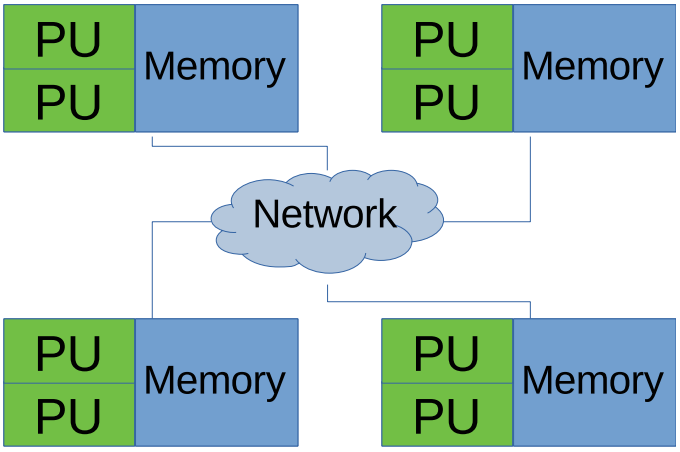

Figure 3.7: Hybrid memory scheme.

3.4 Parallelism grain size

Grain size is a measure of the amount of computation running on a task [35]. Some characteristics, like running and communication time, as well as hardware architecture, are considered when classifying the granularity in different levels, as described in [35] and shown in Fig. 3.8.

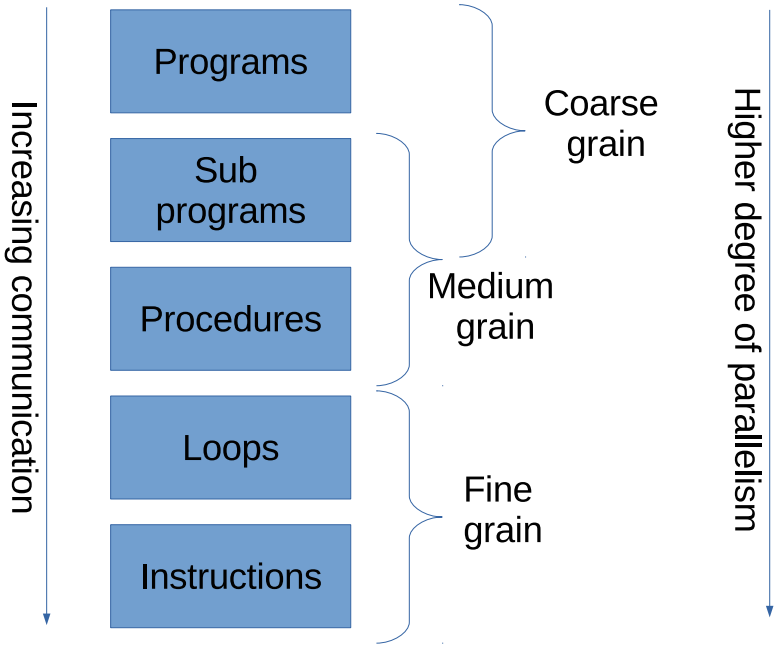

Figure 3.8: Grain size level. Remade from [37]

- *Instruction level*: Processors' instructions can be processed in parallel. Pro-

grammers do not usually apply this parallelism level due to hard implementation, which is usually the work of a compiler;

- *Loop level*: This parallelism is usually applied by SIMD machines, mainly by vectorized machines. It can be made by compiler or programmer on MIMD machines, where a loop is split to parallel execution. This level is considered as fine grain;

- *Procedure level*: This level consists of split procedures, or even a rewrite of some parts to different processes. It is usually made by the programmer and requires some communication. This level is considered as medium grain;

- *Sub program level*: Code is split in large parts and sometimes becomes different for each process. It requires communication and is considered as medium grain like the *Procedure level*, or coarse grain if source code grouped in very large parts;

- *Program level*: Program level grain is the more practical level and is usually applied to embarrassingly parallel problems. A low amount of communication is required.

The grain size applied to this work was chosen considering a *procedure level* aiming achieving a better scalability. The finer the grain the more is the dependence on hardware architecture, which directly affects the scalability. Furthermore, a coarse grain is more suited to embarrassingly parallel problems, which are not applied to this work, since increasing the number of processors also increases the communication. Then, a medium grain size is more suited to this work.

3.5 Evaluation of parallelism performance

A very common measure to evaluate performance on parallel programs is the speedup and efficiency metrics [71].

Speedup is a ratio between serial and parallel execution time, as shown in Equation (3.1):

$$S(p) = \frac{T_s}{T_p(p)} \tag{3.1}$$

where:

p = number of processors,
T_s = serial execution time associated to the best algorithm known in the literature,
$T_p(p)$= parallel execution time to p processors

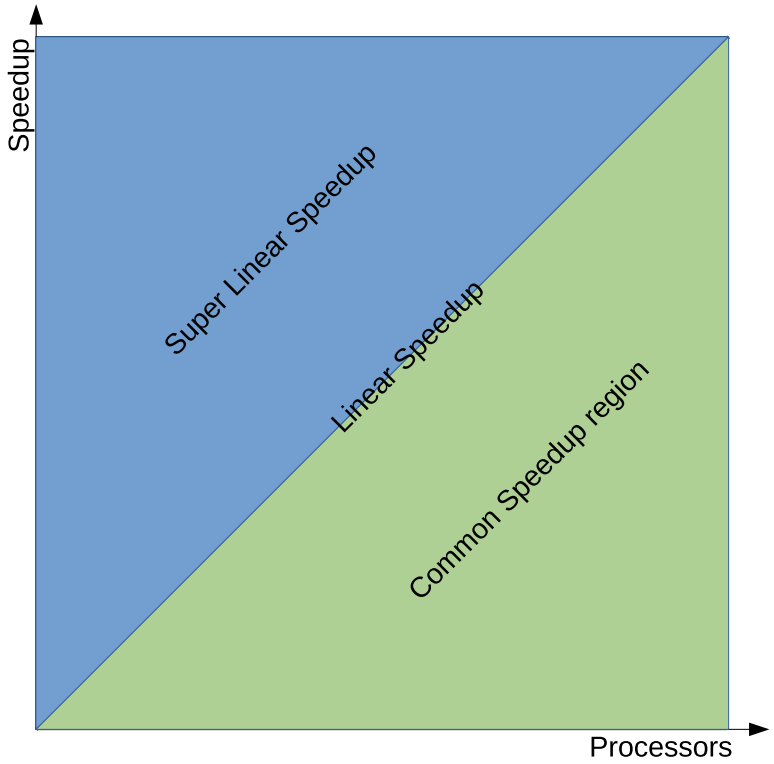

Figure 3.9: Speedup regions.

The efficiency metric is derived from speedup and measures how scalable a parallel algorithm is, as shown in Equations (3.2) or (3.3):

$$E(p) = \frac{T_s}{T_p(p)p} \tag{3.2}$$

$$E(p) = \frac{S(p)}{p}100\% \tag{3.3}$$

A speedup chart can be drawn to evaluate performance for increasing values of p, as shown in Fig. 3.9. This chart shows three different regions, whose meanings are as follows:

- *Linear speedup*: if $S(p) = p$, which means that increasing the number of processors to computation adds no overhead on CPU time;
- *Superlinear speedup*: if $S(p) > p$, it may happen that the serial algorithm to which the parallel algorithm is being compared to is not the fastest one. However, in some cases it may be due to hardware features, such as more memory in the parallel system than in the serial system;
- *Common speedup region*: if $S(p) < p$, which is the expected speedup region, since $T_p(p)$ will probably be smaller than $\frac{T_s}{p}$, due to communication overhead

and idle processing time. Therefore, the most common scenario is the one where $S(p)$ saturates on increasing p.

3.5.1 Amdahl 's law

In the work [2], Amdahl stated that speedup can be calculated based on a factor f of the serial fraction of the algorithm and the number of processors, as shown in Equation (3.4). In this computation, the problem size was fixed to $T_s + T_p = 1$.

$$S(p) = \frac{T_s}{fT_s + (1-f)\frac{T_s}{p}} = \frac{p}{1 + (p-1)f} \tag{3.4}$$

where:

T_s= serial time,
f = factor of only serial part of algorithm
$0 \leq f \leq 1$

In Equation (3.4), increasing the number of processors to infinity leads to the maximum value of speedup, which is:

$$\lim_{p \to \infty} S(p) = \frac{1}{f.} \tag{3.5}$$

However, Amdahl had a pessimistic view of the parallel performance, because in his speedup calculation a constant problem size was assumed. The section below describes a different point of view.

3.5.2 Gustafson 's law

In [28], Gustafson discussed Amdahl's law and argued, using scalability concepts, that speedup should be measured by fixing the CPU time to solve the problem and not the problem size, as in Amdahl's law. Usually, a parallel program has a fixed serial time due to data input and handling, and a parallel time that can be split among the processors. Therefore, adding more processors implies on reducing the parallel time but not the serial time, as shown in Equation (3.6). This scaled speedup compares an equivalent serial time by taking into account the processing time of all processors ($T'_s + pT'_p$) with the time considering p processors ($T'_s + T'_p$). The total time was fixed in $T'_s + T'_p = 1$ to perform calculations.

$$S_{scaled}(p) = \frac{T'_s + pT'_p}{T'_s + T'_p} \tag{3.6}$$

where:

$S_{scaled}(p)$= scaled speedup considering a fixed time,
T'_s = fixed serial time,
T'_p = parallel time

Considering $f' = \dfrac{T'_s}{T'_s + T'_p}$ as the fraction of the serial part of algorithm:

$$S_{scaled}(p) = f' + (1 - f')p \tag{3.7}$$

where:

$S_{scaled}(p)$= scaled speedup considering a fixed time,
f' = factor of only serial part of scaled algorithm
$0 \leq f' \leq 1$

We note that the serial factor f, from Amdahl's law, is different from f' in Gustafson's law. The first one considers the size of problem fixed. The second one considers that the problem size increases when the number of processors p is increased too. Therefore, f' is a fraction of an increasing problem size. In Gustafson's law the time is fixed as well as the serial part, therefore increasing the number of processors has positive aspects to the speedup and efficiency evaluation. This is a more optimistic view about the speedup calculations of parallel models as compared to Amdahl's law.

This work used a mixed point of view of Amdahl's and Gustafson's law [4], because the number of subproblems to be solved in each SDDP iteration (which depends on the number of forward scenarios) grows when the number of processors is increased (like in Gustafson's point of view). The proposed algorithm does not follow the same assumption, and can be scaled without increasing the work load, until the number of processors is equal to the number of stages (i.e. $p = T$). Therefore, the adopted serial time is related to the problem with 1 processor executed with the SDDP algorithm.

3.6 Parallel programming models

Parallel programming models consider the computer system architecture to perform processors and memory arrangements, and can be made by different ways [18] on more general groups: *pure parallel programming*, *heterogeneous parallel programming* and *hybrid parallel programming*.

3.6.1 Pure parallels programming models

The models associated with pure parallel programming can be of two types: pure shared or distributed memory. Shared memory models imply different tasks using

the same memory region to read or write. There are some alternatives to achieve this: to execute the program more than once sharing the same memory region, (see Fig. 3.10) and thread models with Portable Operating System Interface for Linux (POSIX) and OpenMP (see Fig.3.11), using threads.

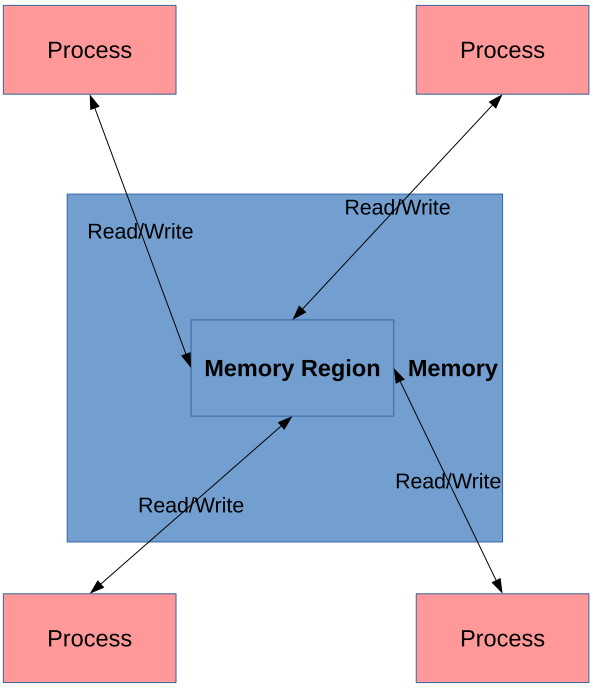

Figure 3.10: Pure parallel models: Shared memory without threads.

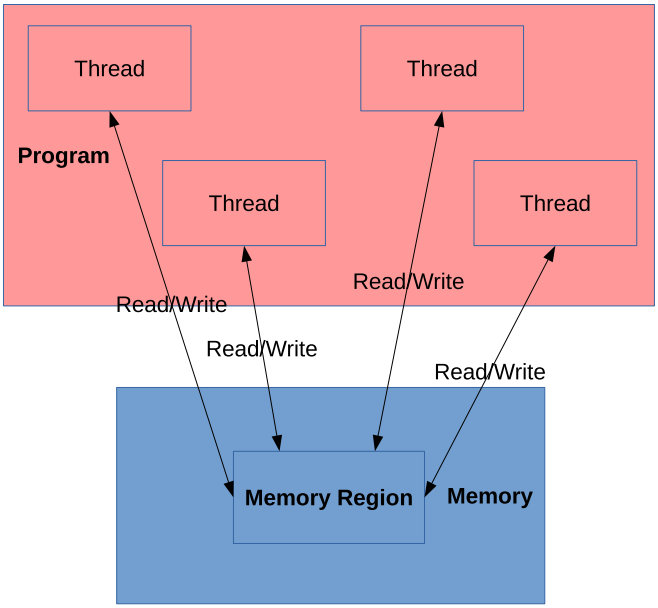

Figure 3.11: Pure parallel models: Shared memory with threads.

Distributed shared memory is another possibility of pure parallel programming

model and Message Passing Interface (MPI) can be used. In this work, parallel programming focuses on MPI communication between processes due to its high scaling capability.

Message Passing Interface (MPI)

MPI is a message communication-based protocol used by processes to communicate with each other. The implementation is not a language but rather a library, being supported by several languages [29] like C, Fortran, Java, and Python.

Processes will be included in groups (called communicators) and receive an identification (called rank id) for each communicator. The initial communicator is "MPI_COMM_WORLD", which contains all processes initiated by the program.

Communication among processes can be collective or point to point, depending on what is best suited for the situation. Both forms of communication have associate parameters: communicator, the rank id, and in some cases a tag that identifies a specific message.

Synchronism is another characteristic exploited by sending and receiving data. Messages can be synchronous, such that all processes involved in communication have to be at this code point to communicate with each other, or can be asynchronous, when processes can send the message without the receiver being in this code point. Asynchronous communications apply user or system buffered data, and the system buffered depends on the MPI library implementation.

Implementations of MPI libraries have to follow the IEEE standard [3], and some methods must exist to "point to point" and collective communications. Some examples are "MPI_Send" and "MPI_Reveive" to "point to point" (both to synchronous or asynchronous communications), and "MPI_Broadcast" to collective communications. These libraries can contain additional methods to improve user experience.

3.6.2 Heterogeneous parallel programming models

In 2001, NVIDIA introduced the first programmable GPU: GeForce3, and later, in 2003, the Siggraph/ Eurographics Graphics Hardware workshop, held in San Diego, first introduced the GPGPU (General-purpose GPU) concept [41]. At this point, programming to GPU was already possible, also allowing programmers to use all GPU cores. Considering that, a computer that already possesses a CPU could use a GPU as well.

NVIDIA creates a language to enable the programmer to use GPU hardware: CUDA (Compute Unified Device Architecture) [53]. This language was specific to NVIDIA cards, so the community worked together to create OpenCL (Open Computing Language) standards to enable the use of heterogeneous parallel programming

with other graphic cards [38].

3.6.3 Hybrid parallel programming models

The hybrid parallel programming aims to combine different models to suit problem needs. Pure shared memory parallel models have better use of memory since they do not have to communicate with each other, as opposed to the distributed memory parallel model. When the problem allows combining coarse grain with a fine grain, a hybrid MPI+OpenMP can be used, for example. A very intensive computation and coarse grain of problem suits to a hybrid CUDA+MPI. The survey [18] exploited some possibles hybrid combinations.

This work could use different parallel programming models, but the available resources limited the choice. Then, a pure parallel programming model using MPI was applied, once the available resources are restricted to nodes with a limited number of cores and the scalability resides on increasing the number of nodes.

Chapter 4

Stochastic Dual Dynamic Programming

The main objective of this chapter is to explain the Stochastic Dual Dynamic Programming (SDDP technique) that is widely applied to solve large multistage stochastic problems. We start describing a general stochastic programming problem, using the decomposition-based dual dynamic programming (DDP) approach as a reference method to solve it, until reaching the SDDP method in particular, which is a sampling-based extension of DDP. Later on, we discuss the SDDP tree traversing protocol with *forward* and *backward* passes, that apply the Benders decomposition technique to build piecewise linear approximation of the future cost function for each stage. Finally, we discuss the application of parallel strategies to SDDP.

4.1 Stochastic Programming

A key point in the hydrothermal coordination problem is the representation of uncertainties, especially stochastic inflows to the reservoir in predominantly hydro systems [45] [65] [25] [27].

Consider a multi-stage problem with multiple scenarios, as shown in Figure 2.2, where both the time steps and the inflow scenarios are discrete. The so-called "complete" tree consists of all possibilities for this model. Each time step is denoted by t, each possible inflow on time step as s, and each combination of (t, s) defines a node in the tree, to which is a subproblem SP_t^s is related, with conditional probability p_t^s to the parent (previous) node of the tree. The total cost from node (t,s) until the end of the planning horizon is described by Equation 4.1.

Time step is the discretization of smaller portion of the horizon study, and it depends on problem being solved, i.e., long, mid and short term can use different sizes of time step. Different time steps associated with there inflow possibilities

can be grouped forming a sub tree when algorithm is solving the problem, the so called "stage". This work considered one time step to each stage, and to avoid misunderstanding problems with "time step" and "step" (explained later on Chapter 5), the term "time step" was called along this work as "stage".

$$C_t^s = f(x_t^s) + \sum_{s \in \Omega_{t+1}} p_{t+1}^s f_{SP_{t+1}^s} \tag{4.1}$$

where:

C_t^s = Total cost for node (t,s),
x_t^s = Decision variables of the problem for node (t,s),
$f(x_t^s)$ = Present cost function for the subproblem of node (t,s),
Ω_{t+1} = Set of all descendant subproblems of node (t, s) in stage $t+1$,
p_{t+1}^s = Conditional probability of node $(t+1, s)$, related to its parent node in stage $t-1$,
$f_{SP_{t+1}^s}$ = Future cost function (or recourse function, as known in the stochastic programming literature), for the subproblem of node (t, s)

The straightforward approach to solve this problem is to to gather the variables and constraints related to all nodes of the scenario tree in one single problem and solve it by a given optimization method. However, it is possible to notice that increasing t and s may lead the size of the tree to become huge and computationally impractical to handle. Therefore, decomposition of the problem becomes mandatory, and one efficient method is based on Benders decomposition [5]. Such technique was applied by the L-shaped method [66], which decomposes the problem into one master problem for the first stage and several subproblems ("slave problems") for the second stage, one for each discrete possibility of chosen random variable. The link between the two stages is done by adding constraints (Benders cuts) to the first (master) problem, and considering the variables in second stage subproblems related to the first subproblem as state variables. The values of the state variables are replaced before solving each subproblem. An extension of this method for the multistage setting was proposed in [6] and was labeled as "dual dynamic programming" (DDP).

4.2 Dual Dynamic Programming

Dual Dynamic Programming aims to decompose the overall problem and solve all nodes of the tree iteratively, until the method reaches convergence by satisfying a given stopping criterion. Each DDP iteration constructs a new linear cut for the piecewise approximation of the cost-to-go function of each node, whose arguments are the state variables coming from the corresponding parent node of the scenario tree. An iteration of the tree traversing strategy of the traditional DDP approach

consists of performing two steps: *forward* and *backward* passes, which are described as follows:

- *Forward* passes: solve subproblems from the first to the last stage, obtaining the solution for the variables of each node and sending them forward to set the state variables of the corresponding subproblems. Figure 4.1 illustrates a *forward* pass of the DDP approach.
- *Backward* passes: solve subproblems from the last to the first stage, creating Benders cuts, and sending them to be added in the subproblems of the corresponding parent node, as shown in Figure 4.2.

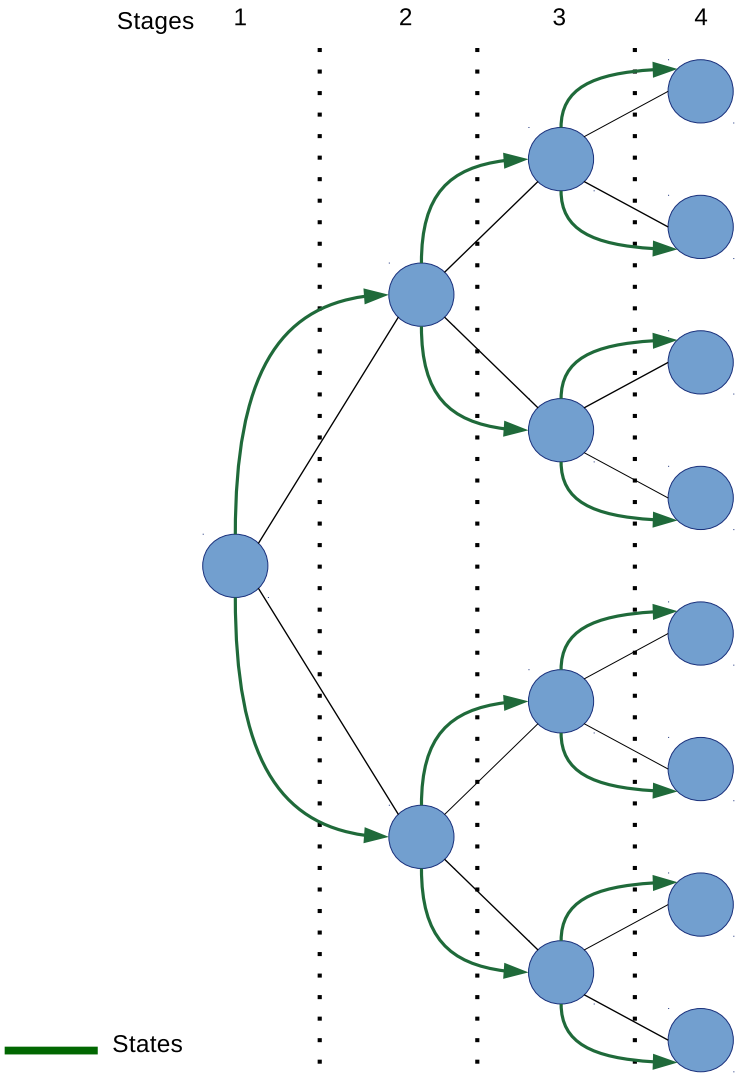

Figure 4.1: DDP forward pass.

The convergence of the method occurs when the difference between the upper and lower bounds for the optimal solution of the overall problem, which are computed

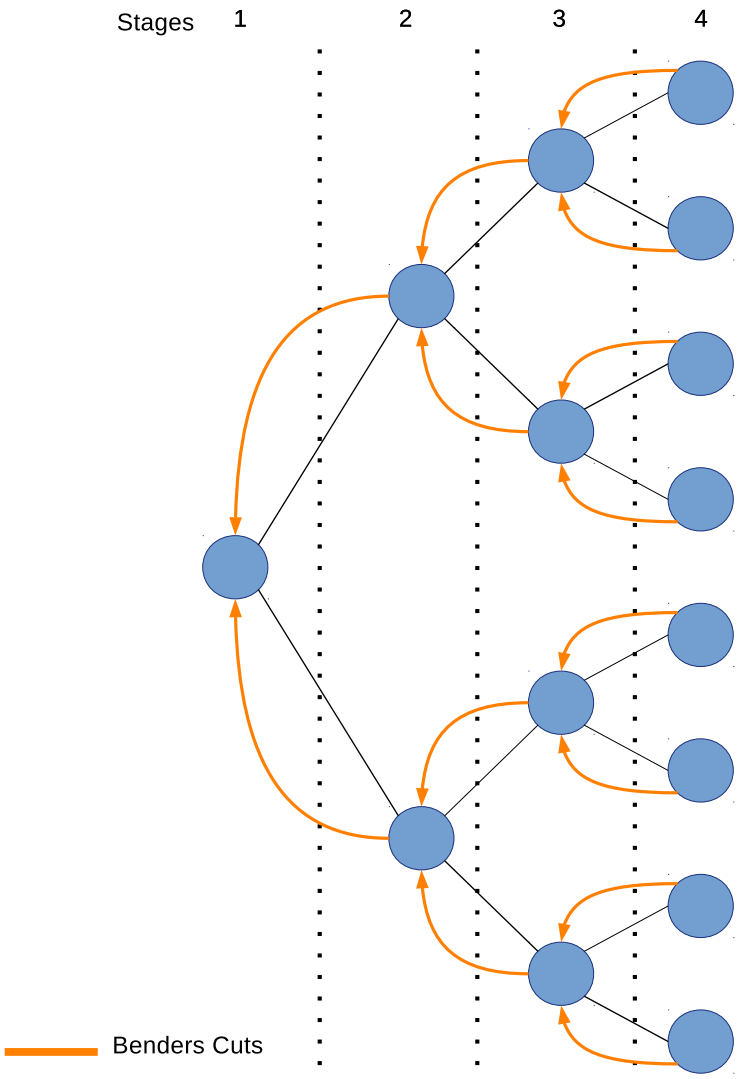

Figure 4.2: DDP backward pass.

in each iteration, is below a given tolerance δ, as expressed in Equation 4.2. The lower bound Z_{lower} is the value of the optimal solution of the root node subproblem, which includes both present and future costs for that node. The upper bound Z_{upper} is the sum of the present costs related to the optimal solution of all subproblems, weighted by their corresponding probabilities [6], as in Equation 4.3.

$$\frac{Z_{upper} - Z_{lower}}{Z_{lower}} <= \delta \tag{4.2}$$

where:

Z_{upper}= Upper bound for the value of the optimal solution
Z_{lower}= Lower bound for the value of the optimal solution
δ = Pre-defined tolerance for the value of the optimal solution

$$Z_{upper} = \sum_{t \in T} \sum_{s \in \Omega_t} p_t^s z_t^s \tag{4.3}$$

where:

T = Number of stages
Ω_t= Set of all subproblems of stage t
p_t^s = Probability of the node (t, s)
z_t^s = Present cost of the subproblem related to node (t, s)

4.2.1 DDP variants regarding cut creation

The original version of the DDP approach [6] used the single-cut version on the algorithm, where the information on the Benders cuts for each stage scenario is grouped into a single recourse function for each node. The multi-cut version in [7] consists in creating individual piecewise linear approximations for the recourse function related to each descendant node of a given node, leading to a more refined approximation. This version is more costly in terms of memory requirements and CPU time to solve the subproblem for each node, but tends to decrease the number of DDP iterations until convergence. Such version could be implemented as well, and its advantage over the single-cut version may depend on the number of descendants and characteristics of the problem. In particular, in one of the asynchronous approaches proposed in this work, the multi-cut version may be attractive, as discussed later in section 5.1.

4.3 Stochastic Dual Dynamic Programming

One extension to DDP is a sampling-based method called SDDP, proposed in [55], which becomes necessary for problems with a huge scenario tree, where it is impossible to visit all nodes. Researchers have derived several variants of the SDDP algorithm like "Convergent Cutting-Plane and Partial-Sampling" (CUPPS, [12]), "Abridged Nested Decomposition" (AND, [22]), and "Reduced Sampling" (ReSA, [31]).

The SDDP algorithm also performs backward and forward passes iteratively, but instead of traversing the whole scenario tree, it generates scenario samples (with resampling at each iteration) until reaching a given stopping criterion, as shown in Figure 4.3. The SDDP iteration is represented in Pseudo Code 4.1.

```
1 stop = false
2
3 for iteration
4 from 1 to maxIteration
5 and not stop
6 and not convergenceReached do
```

```
7
8           reSampleScenariosTree()
9
10          //Forward Passes
11          for stage
12          from 1 to (stagesLenght - 1) do
13              for scenario
14              from 1 to scenariosLenght - 1 do
15                  solve(stage,scenario)
16                  communicateStateVariables(stage,scenario)
17              end
18          end
19
20          stop = stopCriteria()
21
22          if not stop then
23              //Backward Passes
24              for stage
25              from stagesLenght to 2 do
26                  for scenario
27                  from 1 to scenariosLenght - 1 do
28                      for aperture
29                      from 1 to apertureLenght - 1 do
30                          //Receive only if available
31                          receiveCuts(stage)
32                          solveAperture(stage,aperture)
33                          prepareCut(stage,aperture)
34                      end
35                      createCut(stage,scenario)
36                      communicateCuts()
37                  end
38              end
39          else
40      end
41  end
```

Pseudo Code 4.1: SDDP Iteration.

As in DDP, *forward* passes range from stage 1 to $T-1$. At the end of each forward pass, some values of the state variables are obtained where to evaluate and build a new approximation of the FCF of the problem, in the backward pass. It is important to note that, in the SDDP approach, the recourse function is given by stage, rather than by node as in DDP.

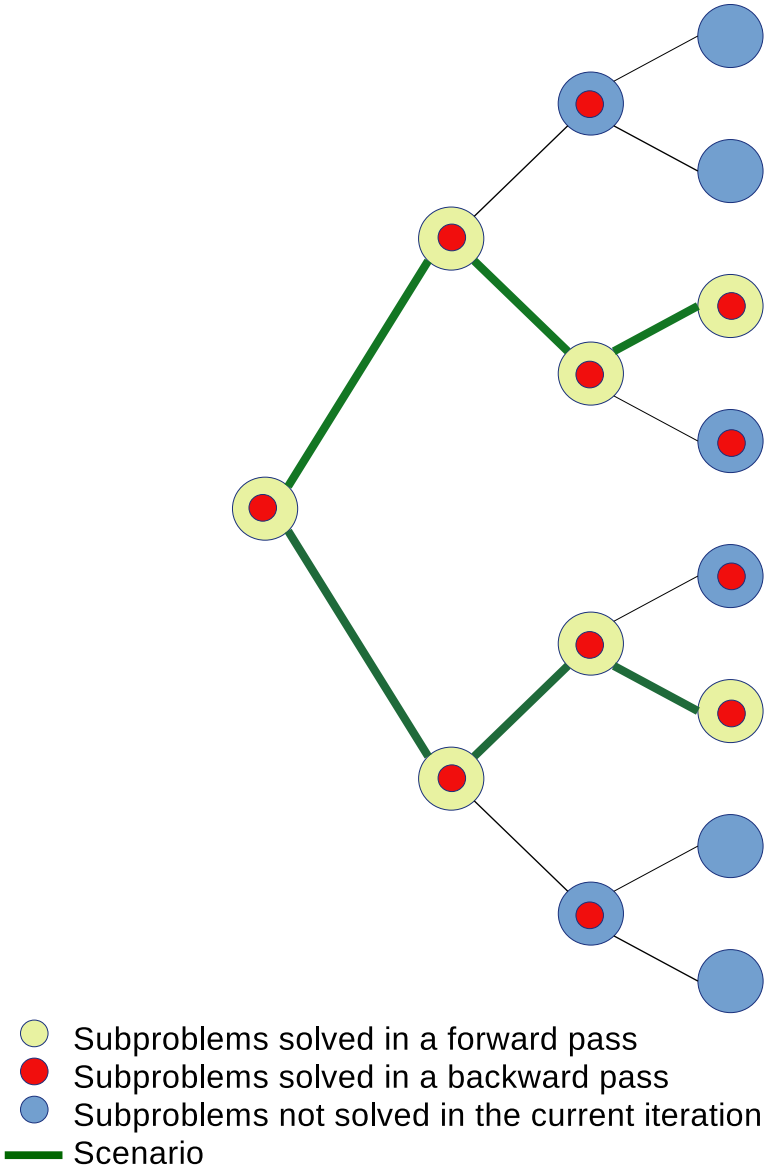

Figure 4.3: SDDP tree.

The optimal value of the solution of the first node is used as a lower bound to the overall problem, and the average sum of the present costs of all nodes, taking into account all scenario samples, is used as a sample value to obtain a statistical approximation for the upper bound, which is valid only in the risk-neutral version of SDDP (for risk-averse approaches, see [56] [39]).

Cut sharing is a valuable technique to improve SDDP performance [36]. When there is a stage-wise independence among the random variables for each stage, and also in some situations where dependency exists [36], samples must share Benders cuts to ensure convergence [44] [49].

4.3.1 Stopping criterion

When the SDDP algorithm was first presented in [55], it was stated that a confidence interval could be applied to Equation 4.2 to obtain a statistic stopping criterion in the risk-neutral case, as considered in this work, since it is possible to evaluate an upper bound based on a sample of scenarios. The work [22] first stated that finite convergence requires resampling through iterations, and later [40] proved an almost-sure convergence to SDDP algorithm considering the independence of sampled random variables. In the same direction, [57] extended such almost-sure convergence proof for the dependent case.

Many researchers presented alternative rules to stop the SDDP algorithm: in [33] it was presented a new version of the statistical convergence criterion based on the proximity between upper and lower bounds. The work [63] proposed a metric based on the stability of the lower bound that compares the lower bound of previous iterations until a relative difference on its variation reaches a value below a given tolerance for N consecutive SDDP iterations.

Another possibility is to evaluate the amount of cuts that have been built. This approach allows a fair comparison between methods that propose different numbers of scenarios along the forward passes [64]. The algorithm converges when a predetermined Benders cuts is reached.

Still regarding the construction of Benders cuts, in [10] it was presented a cut evaluation strategy as stopping criterion. The Benders cuts receive so called "benefit" values, and when such benefit is below a tolerance value, the algorithm reaches convergence. Finally, another rule associated with iterative algorithms is the maximum number of iterations, which was the stopping criterion adopted for this work.

4.4 Traditional Parallel SDDP Strategy

Parallel programs can significantly jeopardize the expected reduction in the overall running time of a program due to communication and processor idle time. Reducing this undesired time can be hard but is needed to successfully improve the program running time (some experience in this topic is provided in [70].

In [14] and [58] it was presented a so-called "standard" parallel scheme strategy applied to, in which scenario samples are solved independently, i.e., the forward scenarios are solved separately by different processes both in backward and forward passes. Another relevant characteristic of parallel processing applied to the SDDP algorithm is the existence of a synchronization point in each stage when samples share their cuts.

The load balance among processes can be dynamic or static. Dynamic load bal-

ance was exploited in [58], consisting of sending all state variables and Benders cuts to a so-called master node, which distributes data each iteration to all processors executing the program. This behavior implies always resetting the linear problems before solving and communicating data from/to master node, thus increasing processing and waiting time. On the other hand, static load balance does not suffer the increasing time by resetting linear problems but requires data updates before solving. Therefore, depending on the application, both approaches can be attractive.

This work considered a static load balance because of the small number of constraints to be updated and a limited number of Benders cuts added through SDDP iterations. Furthermore, the structural differences between the linear problems in each processor is small. However, depending on initial reservoir values and the values of inflows, the time to solve the subproblems for each stage scenario may vary a lot. The static load balance implies dividing the subproblems for all nodes among processors, and the assigned process becomes responsible for a given set of samples.

In the forward pass, each entire forward sample is solved separately by one process. Since it sends information of state variables for the next stage in the same sample and process, there is no extra time to communicate (Figure 4.4).

In the backward pass, even though the different processes independently solve the subproblems for all backward scenarios in the forward samples, there is a synchronization point in every stage to distribute all Benders cuts among the processes (Figure 4.5).

After forward and backward passes are performed in each SDDP iteration, the process that is responsible for the first stage calculates parameters to evaluate the stopping criterion (Section 4.3.1). In this step, depending on the adopted criterion, some communication may be needed. If the value of Z_{upper} is necessary, the information from all present costs has to be sent by all processes to the process responsible for the first stage [58].

Researchers have proposed later other parallel schemas: [30] relaxed the stage dependence to communicate cuts in the backward pass among processes, bringing some good results. The main idea was to wait a certain number of cuts to be added before moving on, thus reducing the idle time due to communication among samples. These assumptions heavily depend on the study case.

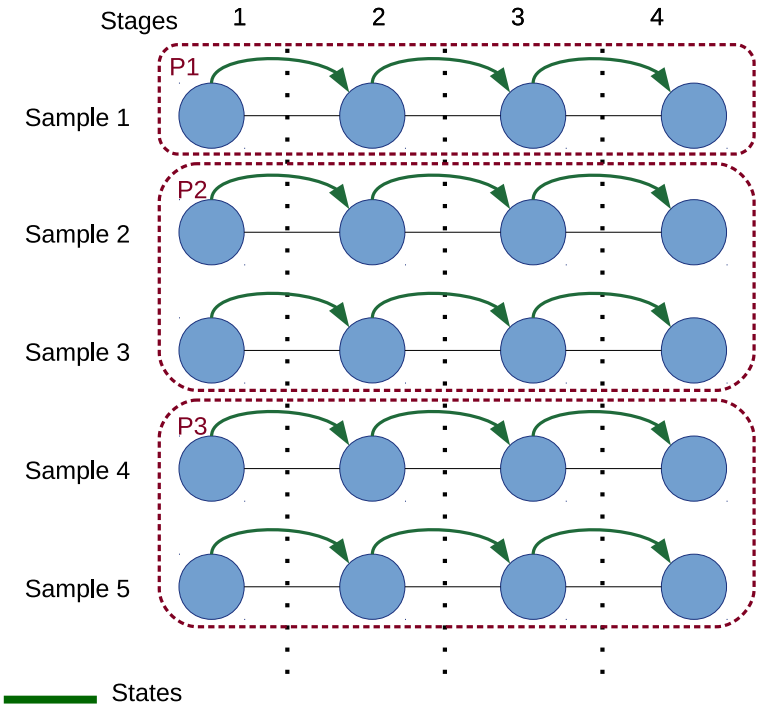

Figure 4.4: SDDP parallel forward.

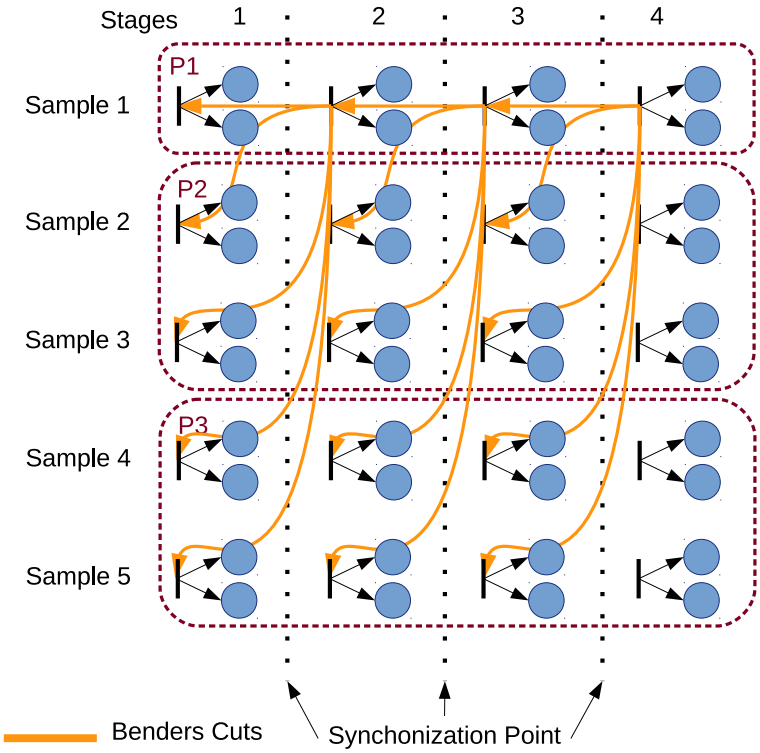

Figure 4.5: SDDP parallel backward.

Chapter 5

Proposed Asynchronous Stochastic Dual Dynamic Programming Method

In the previous chapter, it was shown that the standard SDDP approach has an intrinsic dependency between stages on backward and forward passes, and also a dependency among samples in the backward pass. In order to break this stage dependency, we propose an algorithm which independently performs the solution of all stages and computes backward and forward passes simultaneously. The method proposed here is an extension, to the sampling-based framework, of the successful ideas previously proposed in [62] and [9] for the Dual Dynamic Programming (DDP) method applied to deterministic and stochastic problems, respectively.

Instead of proceeding backward and forward passes as a standard SDDP "*iteration*", ASDDP iterates by proceeding a so-called"*step*." This distinction clarifies the main difference between the two methods. In the traditional SDDP approach, a single iteration of the backward pass transmits the information from the last time stage to the first stage through Benders cuts. In contrast, in the ASDDP algorithm, the information from each stage is only transmitted to its corresponding neighbor stages, therefore the information of the last stage, for example, arrives at the first stage only several steps later (this number is equal to the length of the time horizon).

Fig. 5.1 shows an ASDDP application with only one forward sample and four stages, each one with two backward scenarios. In each step of the ASDDP algorithm, the subproblem for all backward scenarios and stages are solved, and then resampling is performed for the next step. As a consequence, each step also creates a Benders cut related to each backward scenario, which is aggregated when the *single cut* variant of DDP is applied. It is noted that "Implicit forward passes" are formed through the steps, starting at the end of the N^{th} step, when N is the number of

stages.

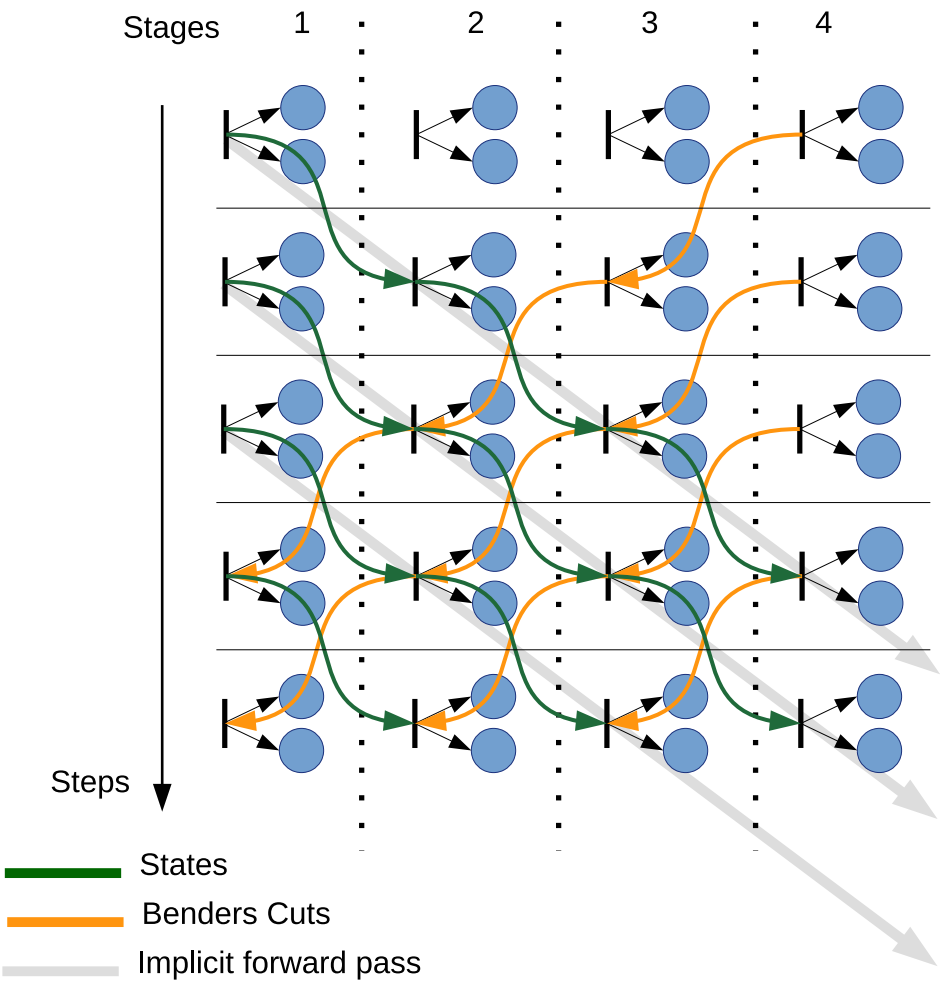

Figure 5.1: ASDDP solution process.

Pseudo Code 5.1 illustrates the solution process of the ASDDP algorithm through steps, where implicit backward (in orange) and forward (in green) passes are processed simultaneously. The algorithm stores Benders cuts and state variables for later communication.

Communication of state variables (Fig. 5.2) and Benders cuts (Fig. 5.3) occurs between neighbor stages, besides an additional communication among different samples if more than one forward sample per step is considered. In each individual sample, the subsequent stages receive state variables from previous stages as in a traditional forward pass of the SDDP algorithm. Also, previous stages receive Benders cuts computed by their subsequent stages from all samples as in a traditional SDDP backward pass.

The proposed asynchronous scheme aims to improve scalability by allowing an increase in the number of processors, especially in the forward pass, which is limited to the number of forward samples in the traditional SDDP algorithm.

```
for step
from 1 to maxStep
and not stopCriteria() do
    reSampleScenariosTree()

    //Backward/Forward Passes
    for stage
    from stagesLenght - 1 to 1 do
        for scenario
        from 1 to scenariosLenght - 1 do
            for aperture
            from 1 to apertureLenght - 1 do
                solveAperture(stage,aperture)
                prepareCut(stage,aperture)
            end
            createCut(stage,scenario)
            storeCuts(stage,scenario)
            storeStateVariables(stage,scenario)
        end
    end

    //Communicate cuts
    //and states variables
    communicateCuts()
    communicateStateVariables()
end
```

Pseudo Code 5.1: ASDDP Iteration

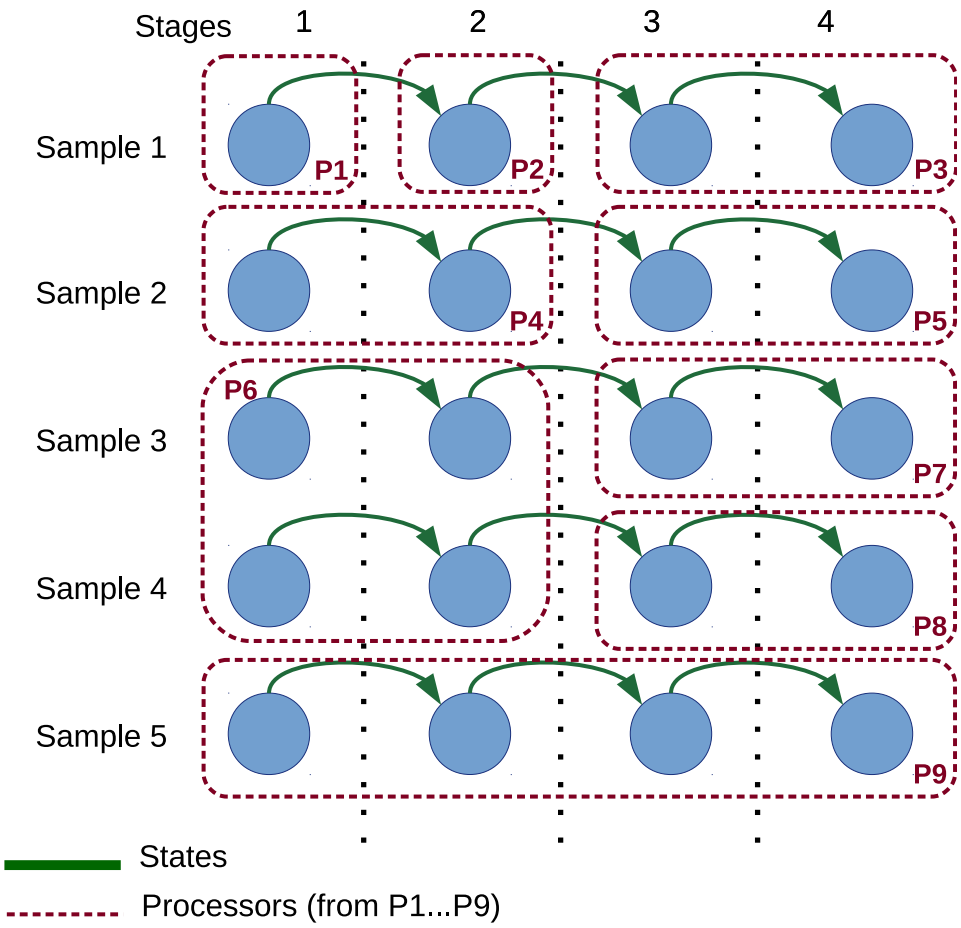

Figure 5.2: Communication of states in the ASDDP approach.

Moreover, the method reduces the idle time among processes in the backward pass, therefore decreasing the CPU time and increasing both efficiency and parallel performance, mainly when the number of backward scenarios is not high, as seen later in the results.

The algorithm synchronizes all information among processes at the end of each backward pass, i.e., a given process awaits both the Benders cuts and the states calculated at the current backward pass. Pseudo Code 5.1 illustrates this behavior, which guarantees that each process will perform the job related to each step in the same way, no matter how many processors are used to run the program. This behavior yields the property of reproducing the same results regardless of the number of processors, which may be required in certain applications, such as in the official operation planning of the Brazilian HTC problem. [45] [21]

A variant of this algorithm without a synchronization point is presented at Section 5.1. Both algorithms take into account the number of stages and scenarios to distribute subproblems to processors. Fig. 5.4 and Fig. 5.5 exemplify a possible division of a problem with two samples and four stages when four processors and

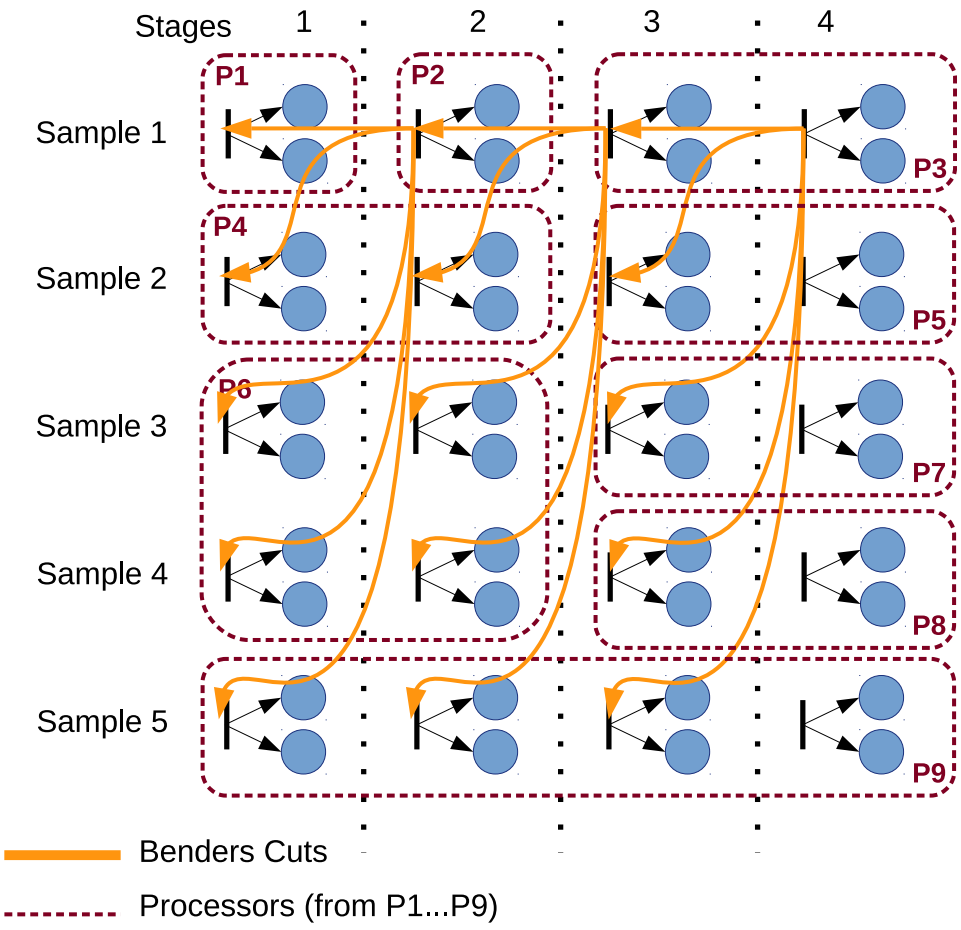

Figure 5.3: Communication of Benders cuts in the ASDDP approach.

eight processors are employed, respectively.

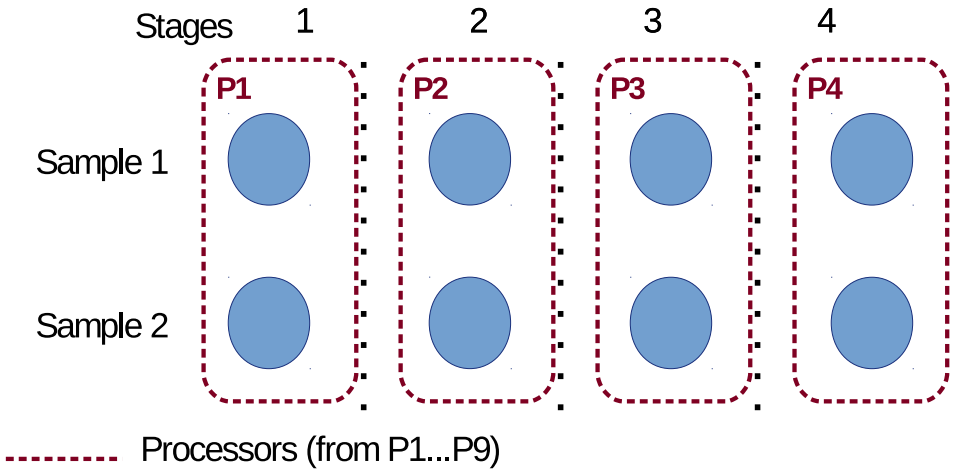

Figure 5.4: Division of an ASDDP parallel problems in 4 processors.

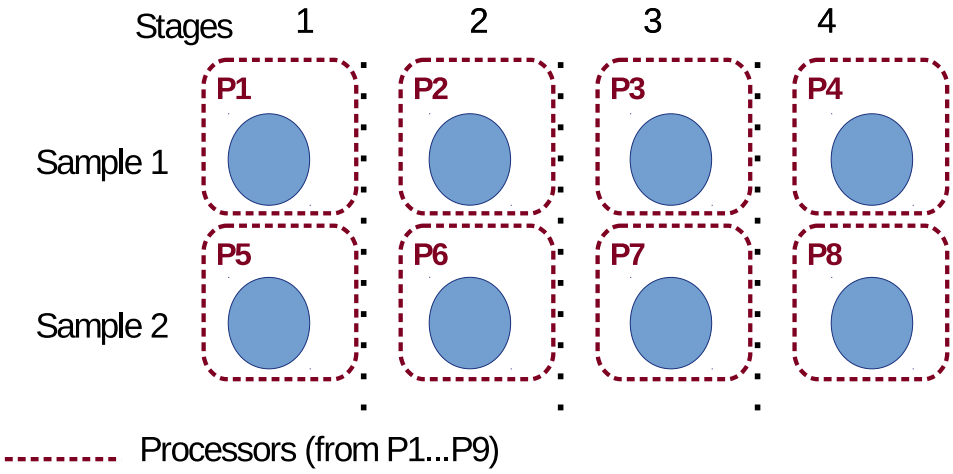

Figure 5.5: Division of an ASDDP parallel problem in 8 processors.

5.1 Variant without synchronization point

This approach, labelled as "totally" asynchronous ASDDP (or TASDDP) does not require a full synchronization before each step, and waits for any incoming data to a given processor to proceed to the next step. As a consequence, in this algorithm it is not guaranteed that all processes will be performing the same step of the overall TASSDP approach, and the results may vary according to the number of processors. Thus, TASDDP does not have an iteration as SDDP nor a step as ASDDP. The single cut version of DDP was implemented to this TASDDP variant, as described in the Pseudo Code 5.2. In [30], a similar approach was proposed in the backward iteration, such that a subproblem from a previous stage starts being solved without receiving Benders cuts from all backward scenarios of the current stage.

```
for step
from 1 to maxStep
and not stopCriteria() do
    reSampleScenariosTree()

    //Backward/Forward Passes
    for stage
    from stagesLenght - 1 to 1 do

        for scenario
        from 1 to scenariosLenght - 1 do
            for aperture
            from 1 to apertureLenght - 1 do
                //Receive only if available
                receiveCuts(stage)
                solveAperture(aperture)
                prepareCut(aperture)
            end
            createCut(stage,scenario)
            sendCut(stage,scenario)

            sendStateVariables(stage,scenario)
            receiveStateVariables(stage,scenario)
        end
    end

    //Waits only if not
    //received above
    waitAnyCut(stage)
end
```

Pseudo Code 5.2: TASDDP Single cut Iteration

```
1  for step
2  from 1 to maxStep
3  and not stopCriteria() do
4      reSampleScenariosTree()
5
6      //Backward/Forward Passes
7      for stage
8      from stagesLenght - 1 to 1 do
9          for scenario
10         from 1 to scenariosLenght - 1 do
11             for aperture
12             from 1 to apertureLenght - 1 do
13                 //Receive only if available
14                 receiveCuts(stage)
15                 solveAperture(stage,aperture)
16                 createCut(stage,aperture)
17                 sendCut(stage,aperture)
18             end
19         end
20         sendStateVariables(stage,scenario)
21         receiveStateVariables(stage,scenario)
22     end
23
24     //Waits only if not
25     //received above
26     waitAnyCut(stage)
27 end
```

Pseudo Code 5.3: TASDDP Multi cut Iteration

Since a synchronization point is not applied to TASDDP, in this variant the multi cut version of DDP may be attractive (see Pseudo Code 5.3) because a new cut can be sent to the previous stage as soon as it is generated. The other operations remain the same as in the single cut approach.

5.2 Upper bound estimation

The estimation of the upper bound proposed in [55] is based on the average costs of all stages and scenarios obtained through a Monte Carlo simulation as shown in Equation 5.1:

$$Z_{upper} = \frac{1}{N} \sum_{n=1}^{N} z_n \tag{5.1}$$

where:

N= Number of scenarios,
z_n= Total cost each n scenario

In that work, such estimation was applied to a set of fixed forward scenarios (samples). By contrast, this work applies resampling at each iteration, thus complicating the evaluation of the upper bound, since the new samples can be more optimistic or pessimistic on average, and therefore the upper bound value can oscillate.

The idea proposed in this work to overcome this oscillation effect is to store the latest M values of system operations along forward samples to calculate a moving average along the ASDDP iterations (Equation 5.2). As a result, in each ASDDP iteration an incoming operation cost for a new scenario replaces the oldest one in the list. Since the upper bound calculation occurs when an implicit forward is completed, the first upper bound computation is obtained only after T steps (see Figure 5.1).

$$Z'_{upper} = \frac{1}{M} \sum_{m=1}^{M} z^m_{upper} \tag{5.2}$$

where:

M = Number of stored upper bounds,
z^m_{upper}= Upper bound referent to the m^{th} item in list

Figure 5.6 illustrates the effect of moving average on ASDDP upper bound, and shows the SDDP upper bound without that tool.

However, the upper bound estimation for the TASDDP method is not as easily traceable as in ASDDP, therefore it is not used in the same way in that method. A possible upper bound estimation which suits all three methods (SDDP, ASDDP and

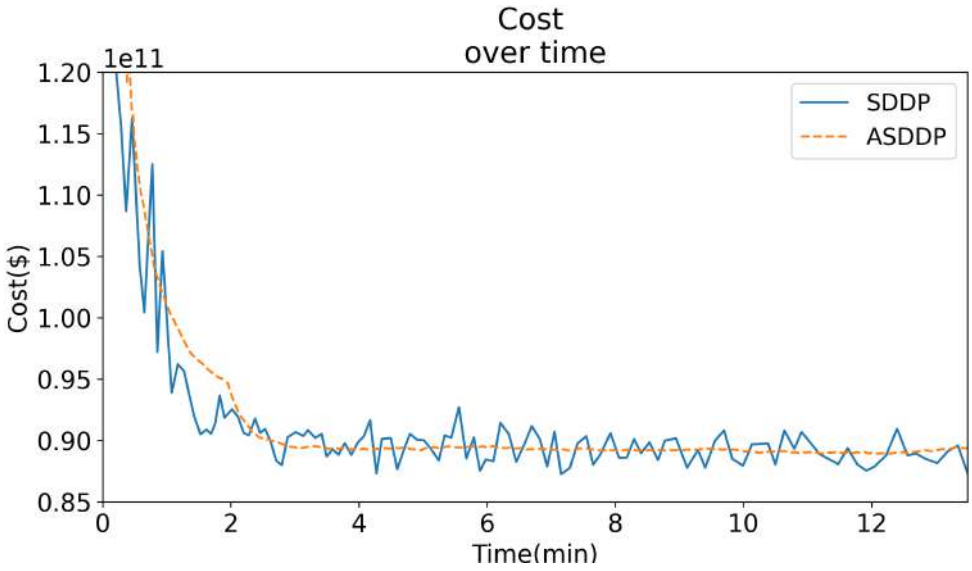

Figure 5.6: SDDP and ASDDP upper bound on resampling behavior.

TASDDP) is to perform a simulation of the policy with a large number of samples after the forward/backward passes (maybe after each k iterations), and to estimate the upper bound as the average cost among them [43].

5.3 Implementation aspects

Software Engineering is an essential subject that is related with every step in software development, since creation until deploy and future maintenance [68].

Large software frequently use the object-orientation (OO) paradigm, which is based on the use of classes and objects. A class is a model from a real-world entity and is composed of attributes and methods. An object is an instance of a class, which enables different combinations of attributes values. Classes can be pure or abstracts, and in the latter case cannot be instantiated directly, but only by an inheriting class.

The use of classes and objects allows the software to be reusable and maintainable, which are essential characteristics to large and long-range software. The term "Reusable" means to improve the capability of using a class in multiple parts of the code. Also, maintenance refers to the easiness on how to add, change, or read source code [68] [15].

Two other software features that ensure reusability and maintainability are "coupling" and "cohesion" [32] [48] [60]. Coupling defines the software arrangement on connecting parts, and cohesion is the grouping of these parts by a common subject.

The base of OO is on the following concepts [59]: abstraction, inheritance, polymorphism, and encapsulation, which are described below:

- *abstraction* means to create a model from the real-world problem;
- *inheritance* ensures the re-use of functions and data from another defined class, improving the approximation of the model to the real-world problem. The inherited class is called "base class" (or superclass), and the inheriting class is called a "subclass" (Fig. 5.7).
- *polymorphism* is an abstraction where an object can assume different behaviors on runtime. The inheriting class overrides methods or attributes as the model needs.
- *encapsulation* hides or shows class members from external access or inheriting classes as convenient, in order to protect parts of the code or only to show what is relevant. "Public", "protected", and "private" are the common encapsulation levels.

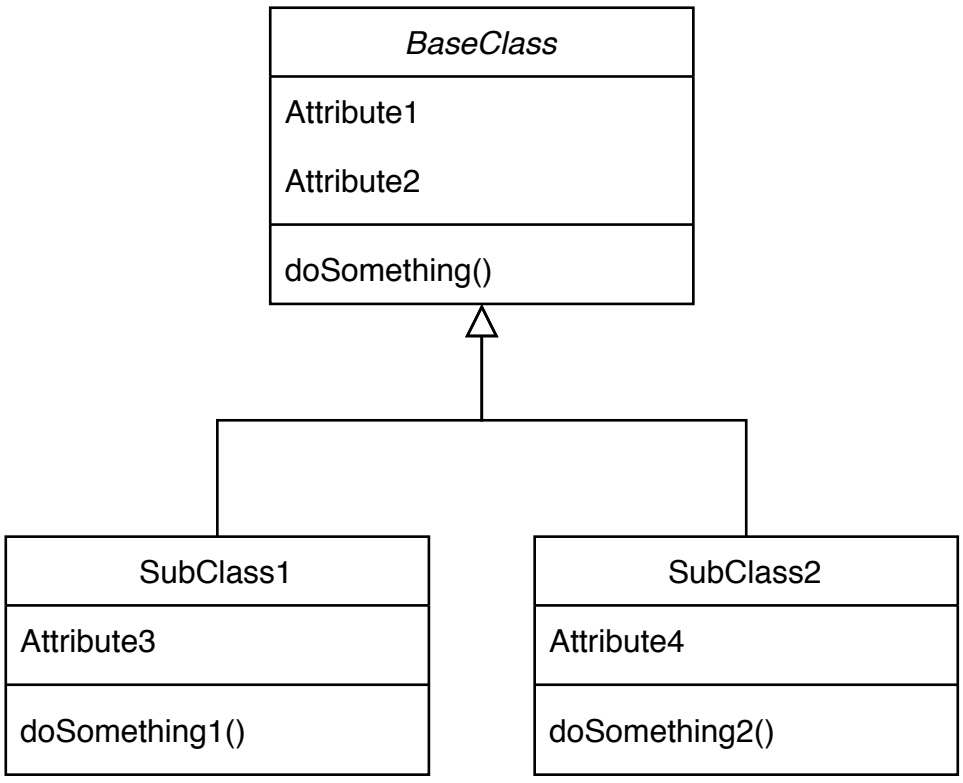

Figure 5.7: Illustration of class inheritance.

Software design patterns

Software evolves as humanity does [67], and one evidence is the development of "design patterns". Design patterns are standard solutions to similar problems that occur recurrently in different situations and models. Besides, the use of a design

pattern can improve the understanding of the source code by other software designers when reading it [51].

There exists a lot of well-known design patterns [26] [51] [1], and several fields of knowledge use these designs [54] [34] , as electrical knowledge, [23] [61].

This work used, in particular, two design patterns: “Factory” and “Strategy”, which are explained in the next sections.

Factory design pattern

The factory design pattern [26] uses factory methods to create objects without specifying which class will be the base of the created object. A base factory class (preferentially an interface) will contain methods to create objects, and subclasses that inherit this one will override methods to return the object. These methods, internally, will create the objects, and the caller class will not need to call class constructors, see Fig. 5.8 and Pseudo Code 5.4. Using the factory design pattern increases the decoupling in software, and consequently, its maintainability and reusability.

```
1  IFactory factory;
2
3  //Instance of the factory due to condition
4  if(condition)
5      factory = ConcreteFactory1();
6  else
7      factory = ConcreteFactory2();
8
9  //Create an object, independently of factory type
10 Type1 object = factory.createObject();
```

Pseudo Code 5.4: Factory design pattern use.

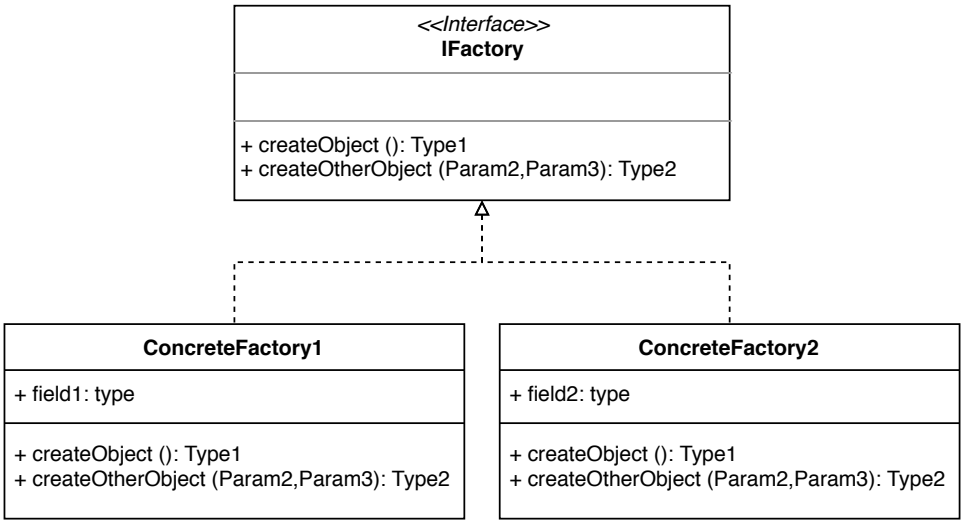

Figure 5.8: Factory design pattern.

Strategy design pattern

The strategy design pattern enables choosing a strategy behavior on run time [26]. It is strongly linked with reusability and maintainability and helps to improve source code decoupling. Such behavior, represented by a base class (preferentially an interface), is inherited by implementations, subclasses, and used by other classes that do not know this behavior (see Fig. 5.9 and Pseudo Code 5.5 to more details).

```
1  IStrategy strategyObj
2
3  //Instance of the strategy due to condition
4  if(condition)
5      strategyObj= Behavior1()
6  else
7      strategyObj= Behavior2()
8
9  //Create  an object to the class using the strategy
10 ClassUsingStrategy obj
11 //Apply the strategy, independently of its behavior
12 obj.strategy = strategyObj
13
14 obj.doStrategy()
```

Pseudo Code 5.5: Strategy design pattern use.

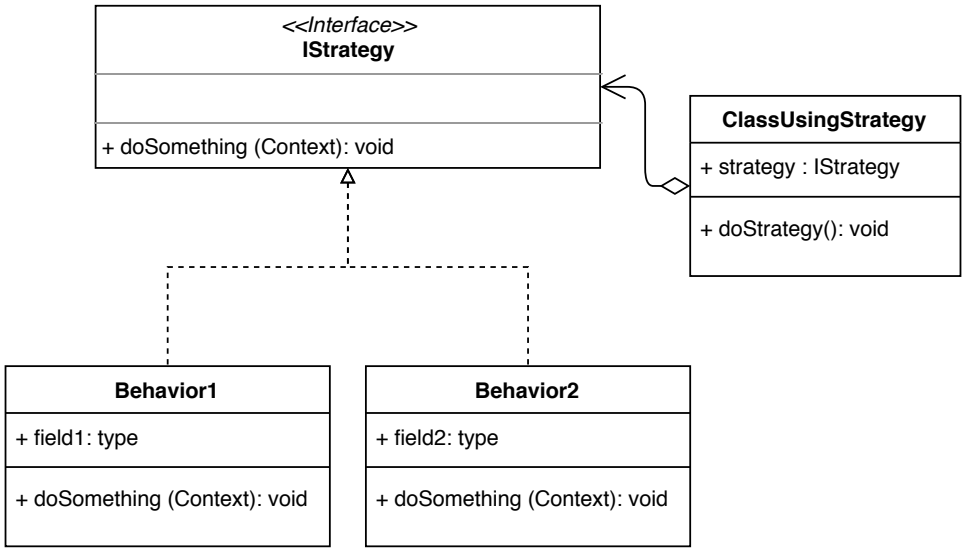

Figure 5.9: Strategy design pattern.

Design patterns on work

Regardless of the method applied to solve the HTC problem (as a single Linear problem, DDP, SDDP, and ASDDP), the variables and constraints are nearly the same. The difference resides on the presence of state variables and the position on the when verifying the stochastic value. For example, when creating the water balance constraint, if Benders decomposition is applied then a state variable should exist.

The phylosophy of the method is that it does not matter which variable or constraint is in the problem, so a factory design pattern suits this situation. When a method creates the individual problems, it iterates over a list of variables and constraints factories. Thus, this design allows using the solution method with any problem. Fig. 5.10 shows some examples of possibles variable and constraints factories applied to the HTC problem.

Moreover, in the solution method, strategy patterns were applied in different contexts such as: the overall problem and on backward/forward passes (see Fig. 5.11), in which a pre solve strategy was applied before solving any subproblem (to update constraints, for example), a post solve strategy applied after solving any subproblem (to store state variables, for example), and a store strategy to put variables and dual solutions at the end of the solution.

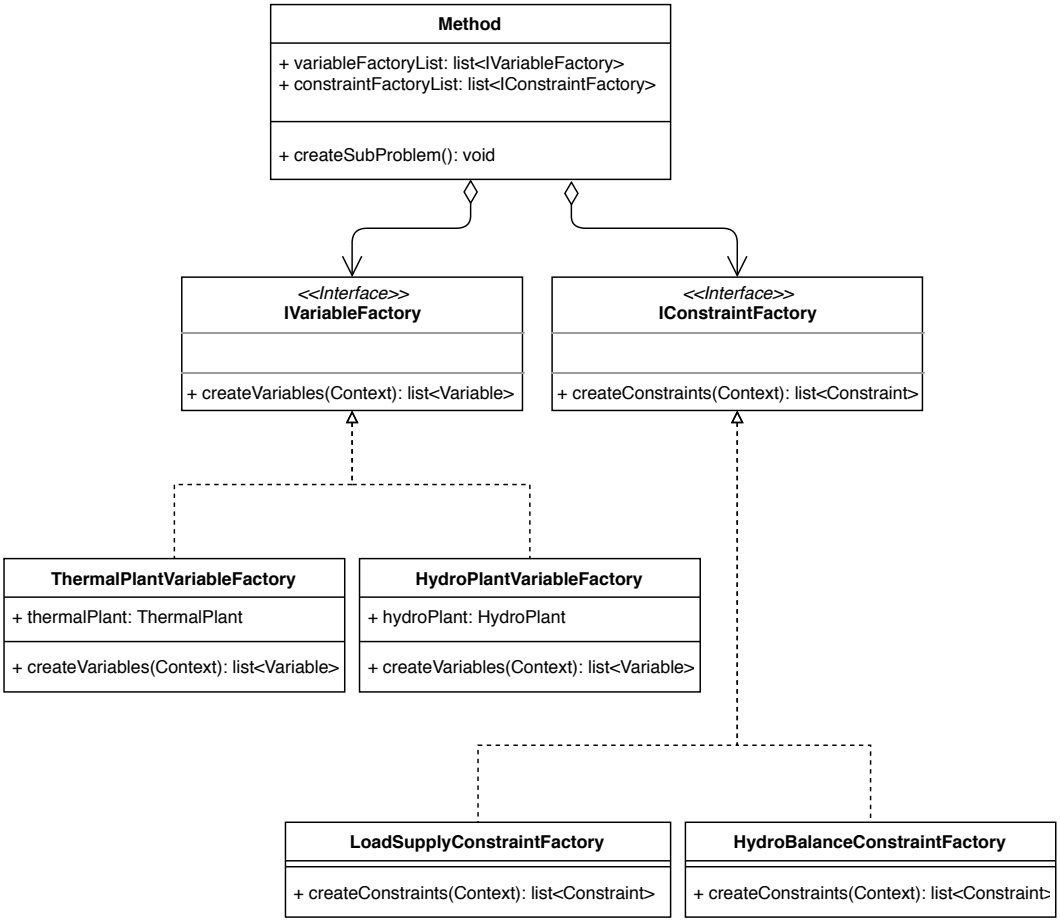

Figure 5.10: Variable and constraint factories examples.

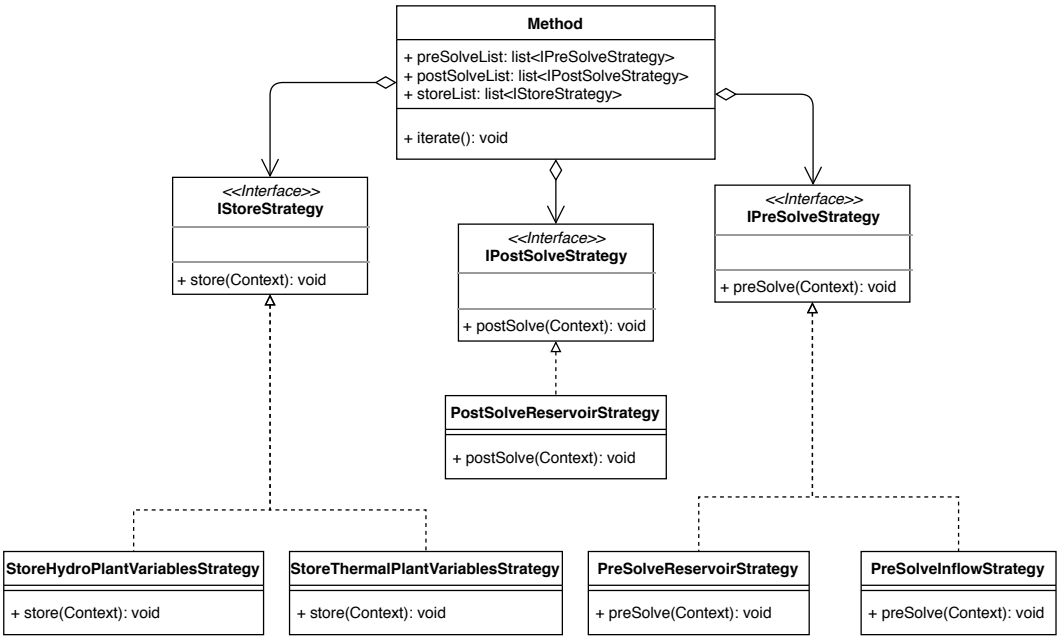

Figure 5.11: Solution methods strategies examples.

Chapter 6

Numerical Results

In order to assess the performance of the proposed ASDDP and TASDDP methods, different sets of tests for the mid and long terms hydrothermal coordination problems were applied to these algorithms, as well as to the traditional SDDP algorithm.

The first test aims to provide a consistency analysis, using a small study case whose solution can be found by solving the overall multi-stage stochastic optimization problem as a single linear program, therefore validating the implementation of the stochastic programming methods applied or proposed in this work.

The second analysis consists in a performance evaluation test, where the lower bound (LB) and average operation cost (AOC) were tracked over time to make a proper comparison among the methods. In these tests the amount of used cores was varied, the CPU time along the iteration/steps of the traditional parallel SDDP and the proposed methods were measured, and the objective values reached by methods were compared.

Finally, a sensitivity analysis of the results was presented, regarding the values of some execution parameters and changes in the hydro inflows data in order to assess the robustness of the methods.

6.1 Hardware and software specifications

The language used to develop the source code was C++ with large use of *boost* library [8]. To solve linear problems the linear programming solver package *Clp COIN-OR* [13] was used. The communication between processes used the MPICH library (version 2) [50].

Since the software was created from scratch, all mentioned methods (DDP, SDDP, ASDDP and TASDDP) were implemented using the same computational features and sharing as much as possible the same classes, in order to guarantee a fair comparison of their performances. Moreover, all implementations using serial and parallel processing. The source codes were included to the context of the

"Libs" project, from CEPEL (Centro de Pesquisas de Energia Eletrica), whose aim is to unify all official programs in a single object-oriented software, written in C++ language.

The implemented SDDP algorithm can be improved by using sophisticated techniques well known in literature, such as cut selection [16] and a more efficient parallel schema as presented in [58]. Also, since the ASDDP and and TASDDP algorithms can be improved as well in both cited aspects to SDDP, we believe that the comparison was fair enough to show the result differences between the presented algorithms.

The implemented SDDP, ASDDP and TASDDP algorithms used as reservoir storage values at the end of the backward passes (for the next time step) one of the results of the backward scenario for the current time step, selected randomly. This value was also used to calculate the forward passes on SDDP.

Deficit cost was not implemented, and to simulate it was inserted a thermal plant with a cost much higher than all others resources.

The tests were performed on a cluster of computers with the following configuration: CentOS Linux release 7.1.1503 operating system, 12 cores, 24 threads - AMD Opteron(TM) Processor 6238 1,530 GHZ and 98GB RAM Memory.

6.2 Study cases

As mentioned previously, the SDDP, ASDDP and TASDDP algorithms were compared using two different stochastic HTC problems, in order to trace their behavior on mid and large scenario trees. Both problems have the same power system configuration, with 46 thermal plants and 84 hydro plants, 44 of which have reservoirs with regularization capacity.

The small study case contains 7 stages with 2 inflow possibilities each, leading to a total number of 64 scenarios at the end of the horizon [9], which allows the application of the DDP solving strategy, in order to traverse the entire scenario tree and to be able to obtain the value of the optimal solution for comparison purposes. All tests considered the single cut variant for those methods.

The mid case extends the smaller case to 24 stages with 5 inflow possibilities each. Such a large scenario tree is more suitable for the sampling-based SDDP-related methods.

Finally, in the long term case the scenario tree is extended to 120 stages with 20 inflow possibilities each, leading to a huge final number of scenarios, which cannot be handled by deterministic-like solving strategies such as DDP and forces the application of the sampling-based SDDP method.

6.3 Consistency test

The main purpose of this test is to validate the correctness of the implementation of the methods and to show that they are able to reach the optimal value of the stochastic program. Initially, the small case was solved as a single LP and the optimal solution was obtained. Then, the SDDP, ASDDP and TASDDP methods were applied with a single processor and one forward sample in each iteration/step with resampling, until the (already known) optimal value was reached.

Figure 6.1 shows the evolution of the lower bounds of SDDP, ASDDP and TASDDP algorithms over time. The horizontal line represents the optimal solution obtained by solving the problem as a single LP. Since all algorithms have reached the optimal value, we conclude that the methods have been implemented properly.

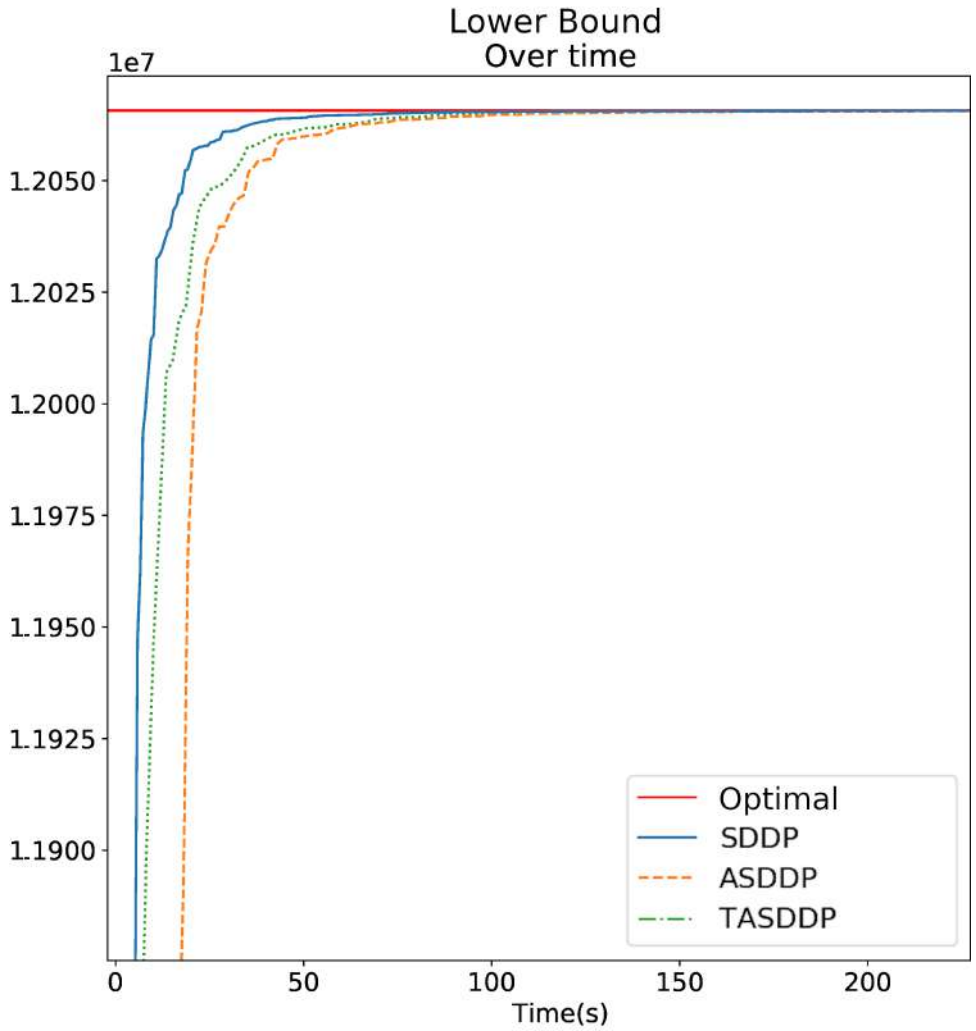

Figure 6.1: Consistency test.

6.4 Performance evaluation tests

The performance evaluation tests aim to analyze the behavior of lower bound and the average operation cost of the traditional parallel SDDP and the proposed ASDDP and TASDDDP methods. The tests varied the number of processors as well as number of samples accordingly, as explained later.

In order to test the parallel processing scalability, the well known speedup and efficiency metrics were used. Since neither Amdahl's [2] nor Gustafson's [28] (Section 3.5) points of view suits the problem, an intermediate criterion was adopted. The problem size is the same to all runs, but the work load is modified when the number of processors is increased. Due to the nature of the SDDP method, for this algorithm the number of samples is directly linked to the number of processors, therefore the work load increases when the number of processors increases. For the ASDDP and TASDDP methods, it is not necessary to increase the number of samples, since a same sample can be divided among several processors, each one solving different stages. Therefore, the speedup and efficiency were calculated with the following metrics: to calculate speedup a chosen algorithm is used as reference for the serial time Ts to be compared with the parallel time Tp for P processors, by the relation $Speedup = Ts/Tp$; the efficiency metric is set as $Efficiency = Speedup/P$.

Some alternatives could be employed for the serial time Ts to compare the SDDP, ASDDP and TASDDP methods: (i) each algorithm itself with a number of samples equivalent to the parallel processing run with P processors; (ii) the serial run of the SDDP method with P processors; (iii) the best serial run among all methods, using only one sample. The last choice was used to allow a fair comparison among the methods, since the other two could lead to very high super linear speedups, because the number of subproblems to be solved enlarges with increment of P.

6.4.1 Mid/Short term problem

The mid term case contains 24 stages and the same characteristics listed in section 6.2. This case was executed with 1, 2, 4, 12, 24, 48 and 96 processors to evaluate speedup and efficiency. The number of forward samples was chosen to maximize the use of processors, thus allowing a fair comparison. Therefore, it was made equal to the number of processors in the SDDP algorithm, and for the ASDDP or TASDDP approaches we selected the number of samples that best suits the amount of available processors, thus solving one subproblem for each processor. Due to the variation of the solution time of each subproblem, the CPU time of the TASDDP method presents a higher sensitivity to the number of samples.

Figures 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8 show the evolution of lower bound and the average operation cost over time.

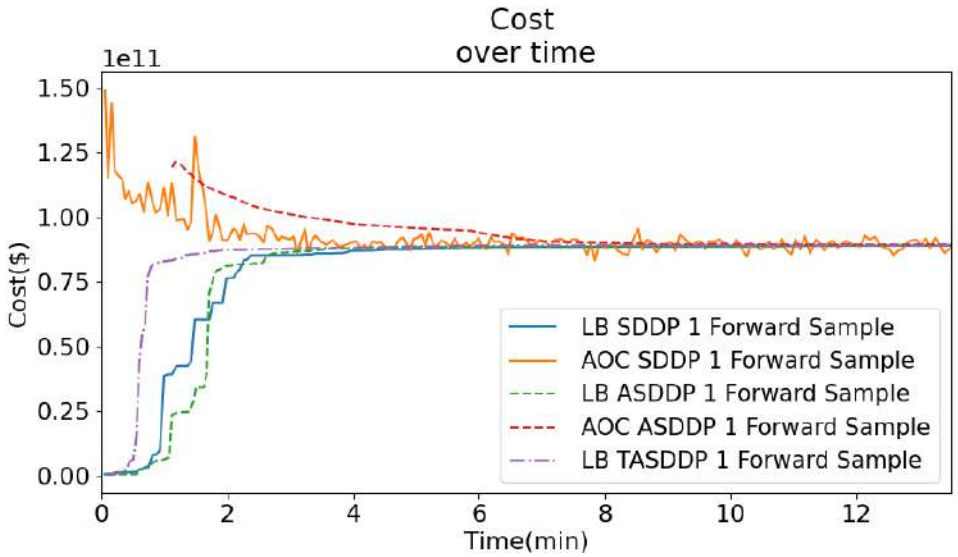

Figure 6.2: Evaluation of lower bound and the average operation cost over time for the mid term case with 1 processor.

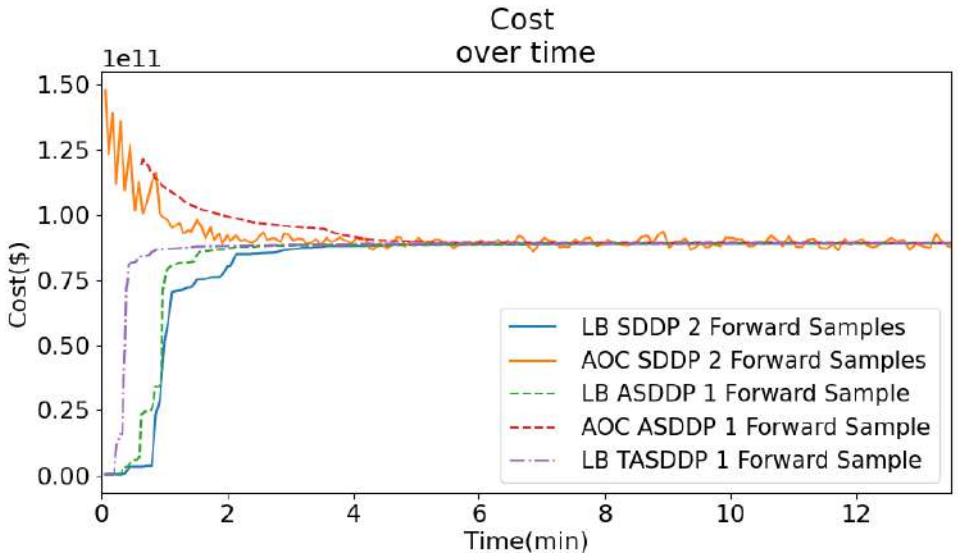

Figure 6.3: Evaluation of lower bound and the average operation cost over time for the mid term case with 2 processors.

According to the evaluation charts, the TASDDP method was the first to stabilize the lower bound value over time when using parallel environments, and this effect is due to its asynchronous nature. In the same parallel environment the TASDDP method is followed by the ASDDP method.

All charts with the average operation cost shows an oscillation both to SDDP and ASDDP methods, as expected. This behavior occurs because resampling can select forward scenarios that are better or worse than the previous ones. ASDDP oscillates less than than SDDP, due to the moving average applied to the average operation cost computation.

An empirical value for the lower bound was adopted to calculate speedup and

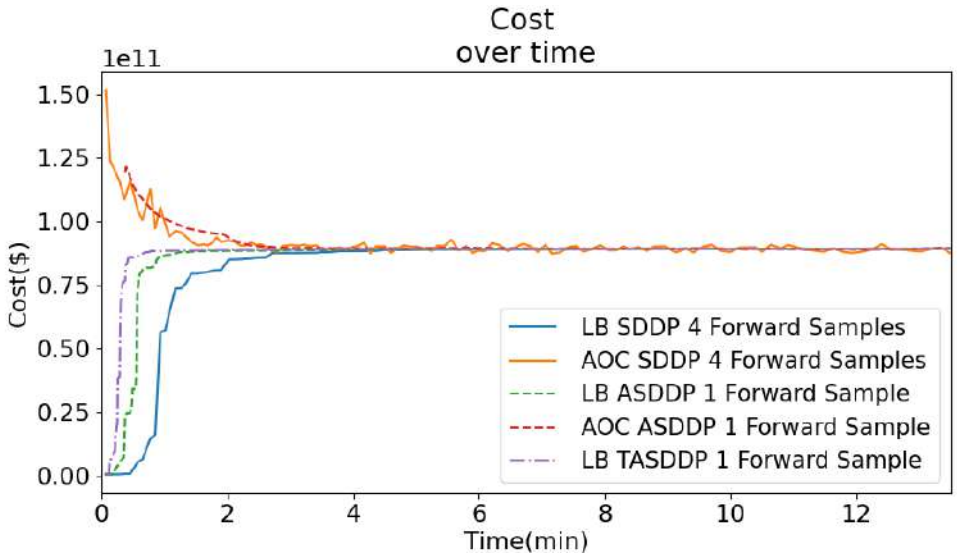

Figure 6.4: Evaluation of lower bound and the average operation cost over time for the mid term case with 4 processors.

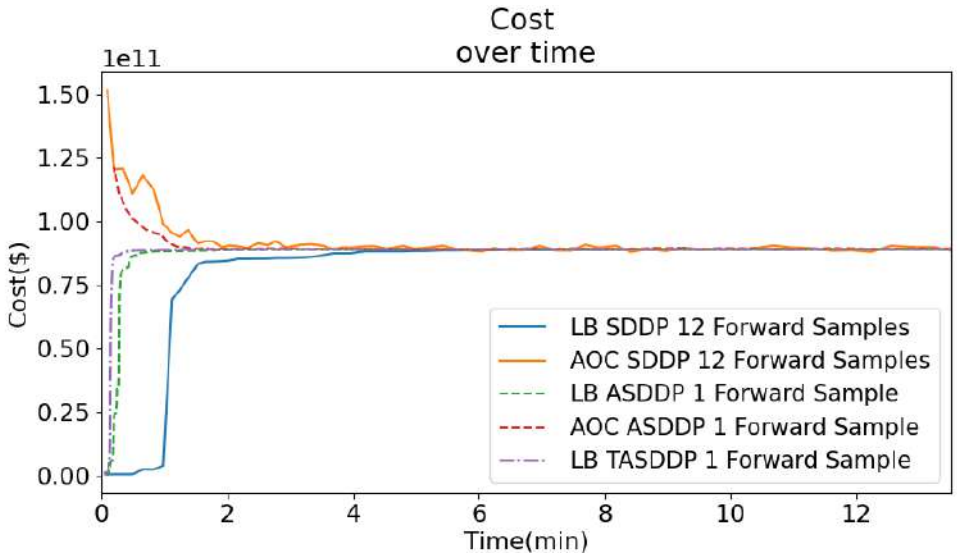

Figure 6.5: Evaluation of lower bound and the average operation cost over time for the mid term case with 12 processors.

efficient of the mid term case, which was based on a value that was reached by all executed cases. Figures 6.9 and 6.10 show the speedup and efficiency, respectively, where it can be seen that the proposed ASDDP and TASDDP methods remain with a good scalability until reaching 96 processors.

Figure 6.2 shows the lower bound and the average operation cost for all methods with 1 processor. Although the lower bound stabilizes in close values for all methods, the average operation cost from SDDP is far better than the others. This is expected, since an explicit forward pass is performed by SDDP since its first iteration, thus the initial storages in the reservoirs for each stage are always sound, which does not occur in the first $T-1$ iterations of the ASDDP and TASDDP methods.

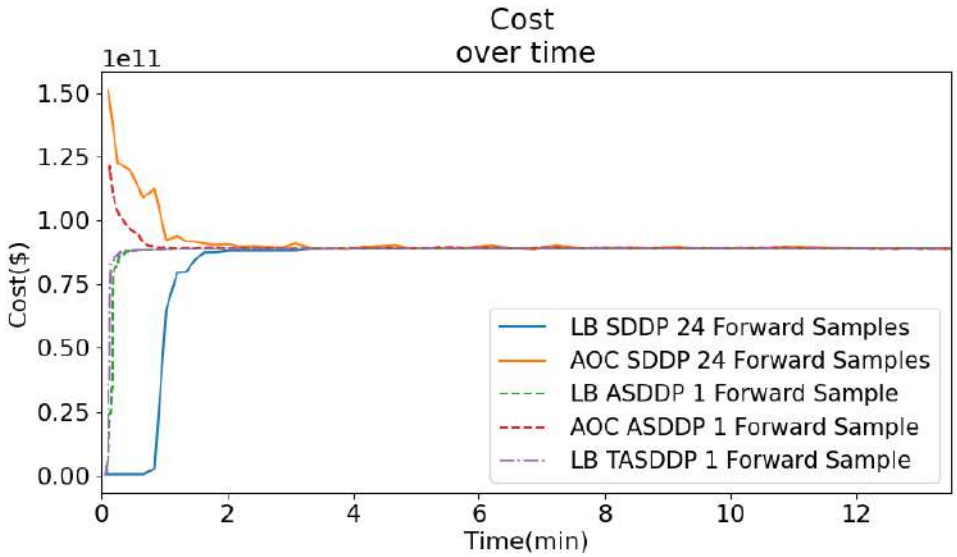

Figure 6.6: Evaluation of lower bound and the average operation cost over time for the mid term case with 24 processors.

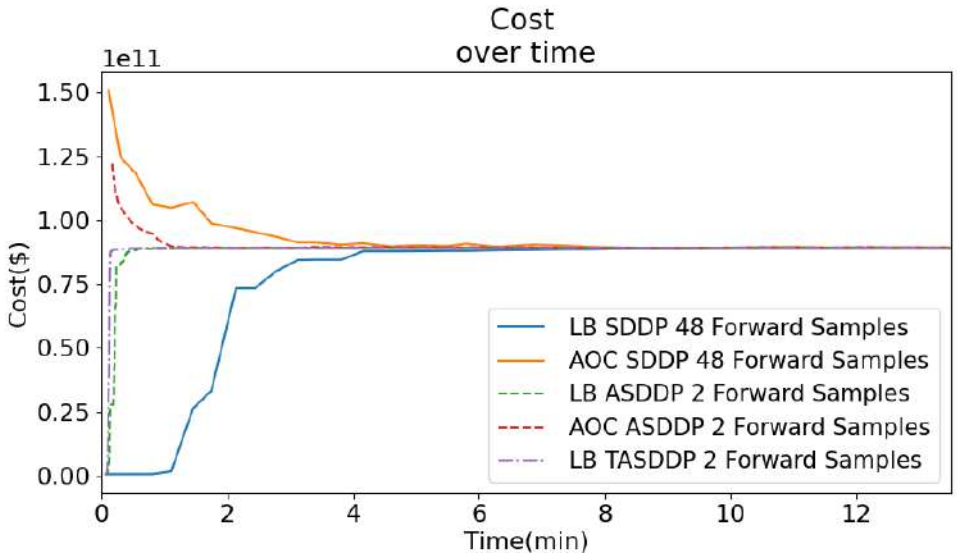

Figure 6.7: Evaluation of lower bound and the average operation cost over time for the mid term case with 48 processors.

Figures 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8 show that lower bound and the average operation cost of the ASDDP method improve as the number of processors is increased, taking advantage of a better scalability, mainly in the TASDDP approach. By contrast, the time of SDDP grows with an increasing number of processors, due to the fact that work load is directly linked to it, i.e., the number of subproblems to be solved in each SDDP iteration grows with the number of processors. In other words, the conceptual advantages of increasing the number of forward scenarios in the SDDP method (as more processors are employed) do not pay off the increase in the CPU time due to the larger amount of work load in each iteration.

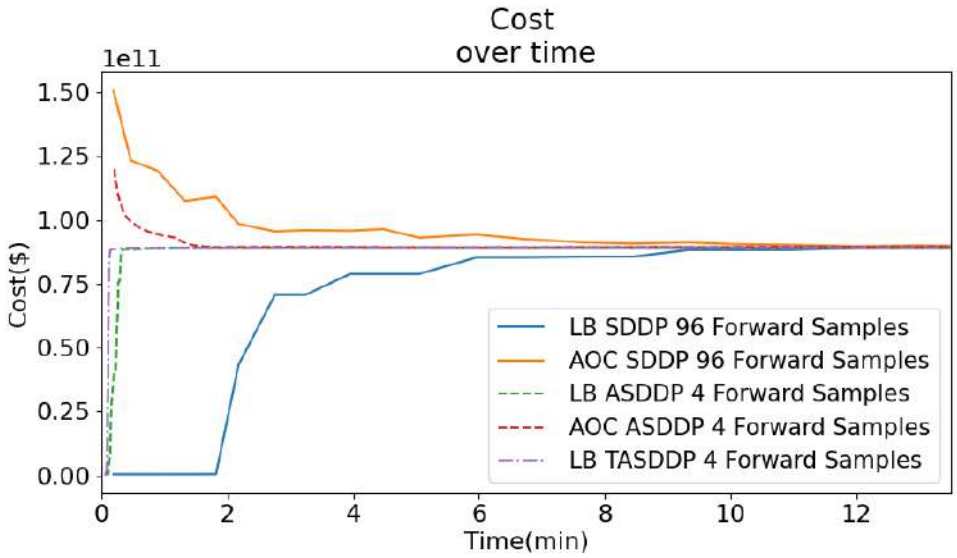

Figure 6.8: Evaluation of lower bound and the average operation cost over time for the mid term case with 96 processors.

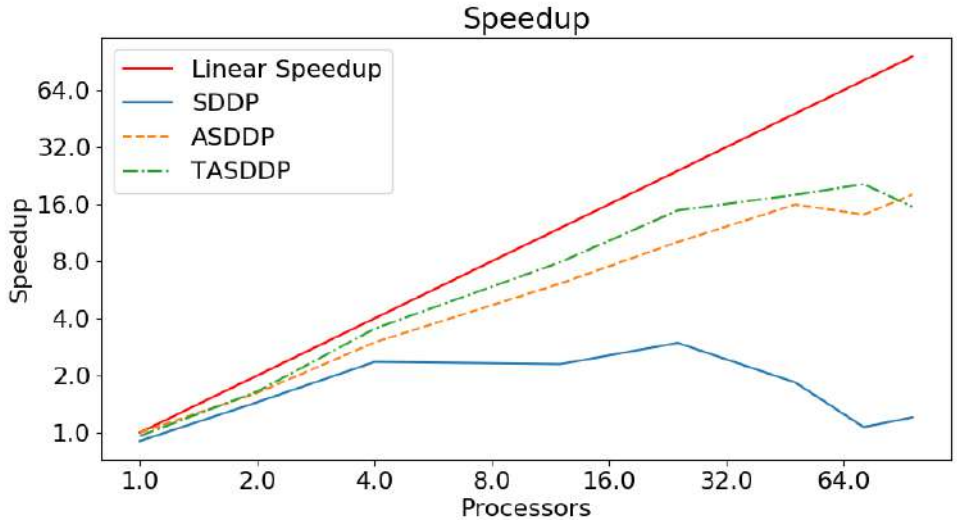

Figure 6.9: Speedup of mid term case with log2 scale.

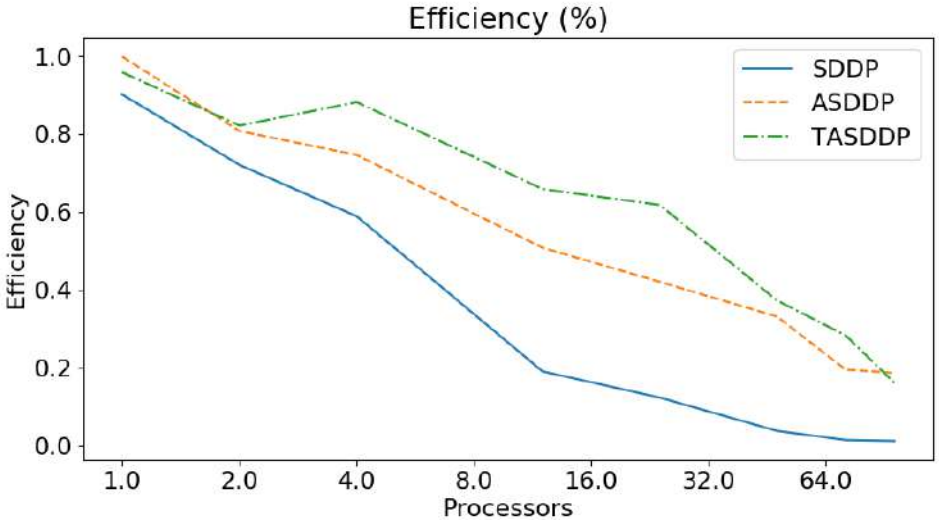

Figure 6.10: Efficiency of mid term case with log2 scale.

6.4.2 Larger case: long term planning problem

The long term test case was applied with 1, 2, 4, 12, 30, 60, 120 and 240 processors. We did not concern in applying a proper stopping criterion because the main objective was to compare the evolution of the algorithms along time. Since we did not know in advance the value of the optimal solution, it was assessed after the test cases were executed, by examining the evaluation charts. The number of forward samples for the SDDP algorithm was again set equal to the number of processors, whereas for the ASDDP and TASDDP methods it was set to one, since the number of processors is usually lower than the number of stages of this test case. For the run with 240 processors we applied 2 forward samples.

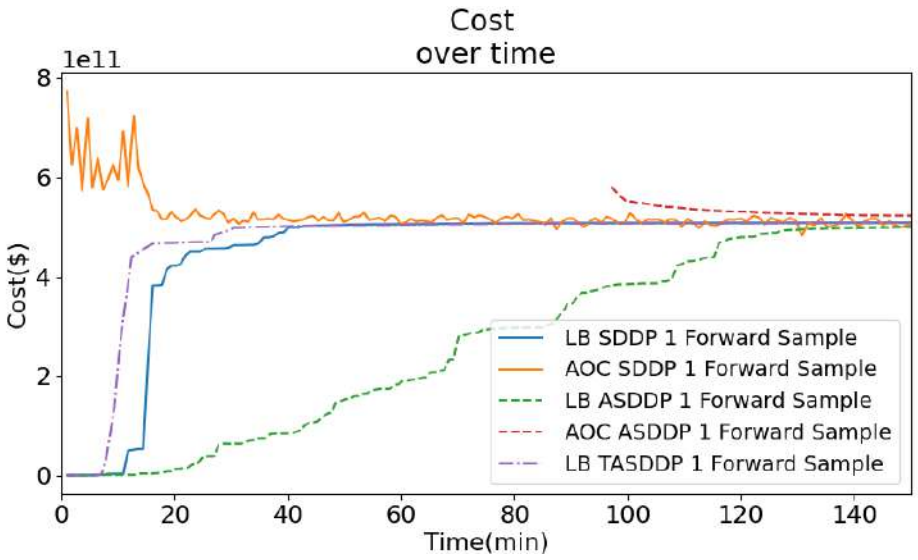

Figure 6.11: Evaluation of lower bound and the average operation cost over time for the long term case with 1 processor.

As in the mid term case, Figure 6.11 shows that the SDDP average operation cost stabilizes faster than the others algorithms for the serial run (1 processor) and the lower bound stabilizes faster than ASDDP and slower than TASDDP. This is explained by the fact that, in principle, the asynchronous algorithms are more suitable only under parallel processing environments, and TASDDP with 1 processor is exactly the same as in SDDP. Figures 6.12 and 6.13 shows ASDDP improving and SDDP losing performance, and TASDDP becoming faster. Later, Figures 6.14, 6.15, 6.16, 6.17 and 6.18 present the opposite behavior, as the ASDDP and TASDDP lower bound stabilizes faster than SDDP. This behaviour is observed in both long term and mid term test cases.

Figures 6.19 and 6.20 show that the proposed ASDDP and TASDDP methods remains with a good scalability until reaching 120 processors.

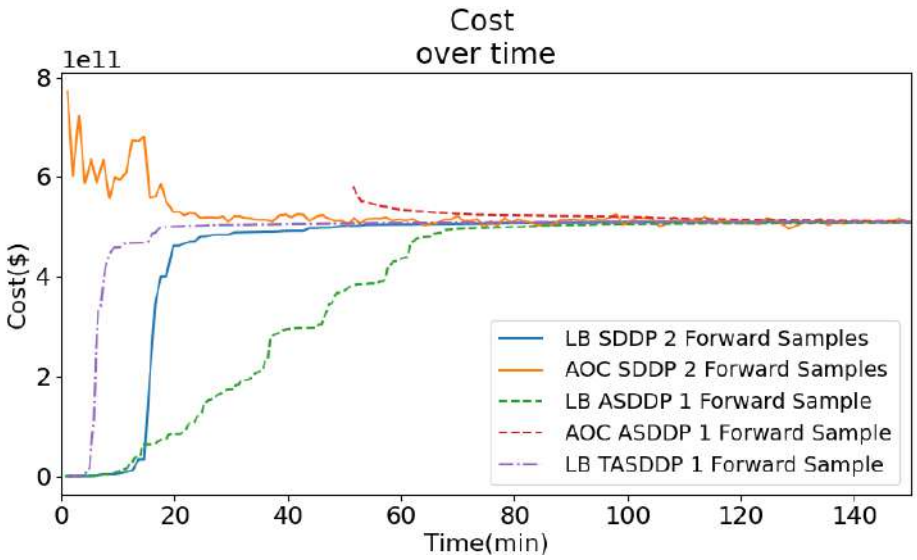

Figure 6.12: Evaluation of lower bound and the average operation cost over time for the long term case with 2 processors.

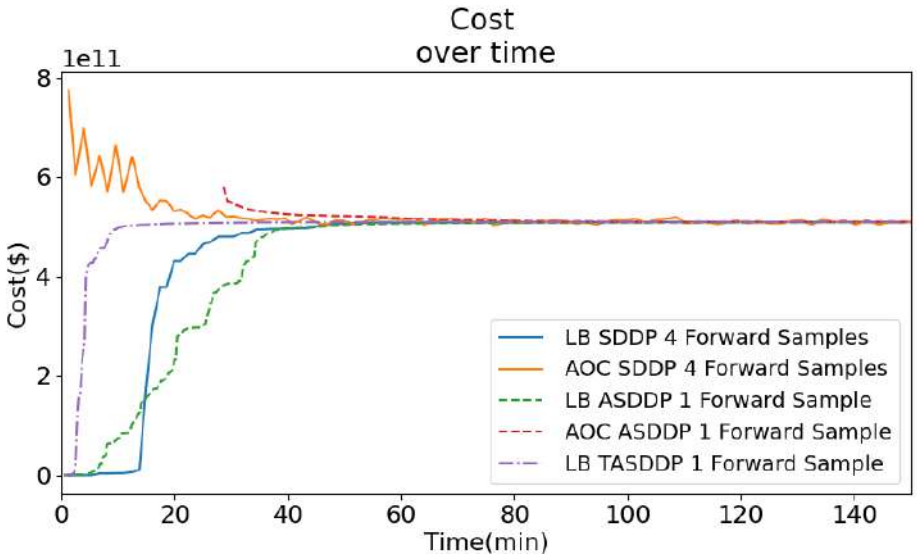

Figure 6.13: Evaluation of lower bound and the average operation cost over time for the long term case with 4 processors.

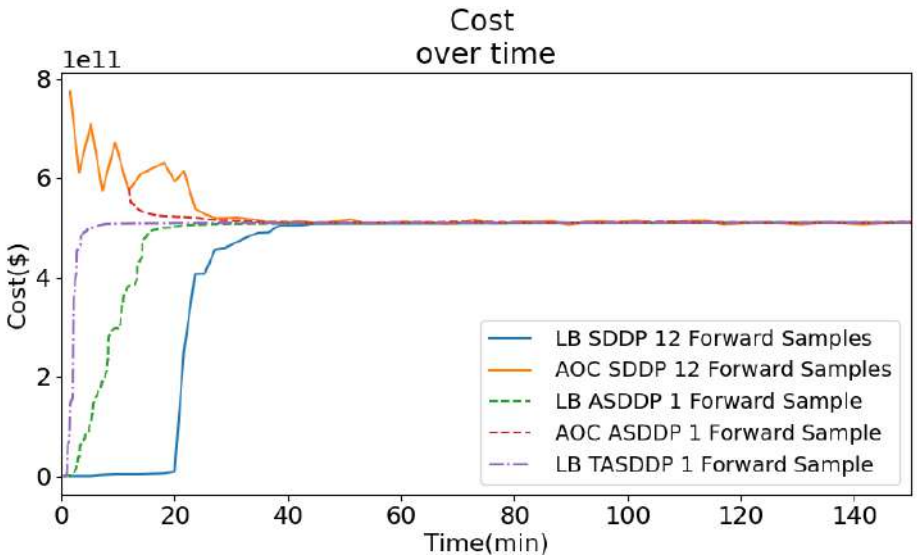

Figure 6.14: Evaluation of lower bound and the average operation cost over time for the long term case with 12 processors.

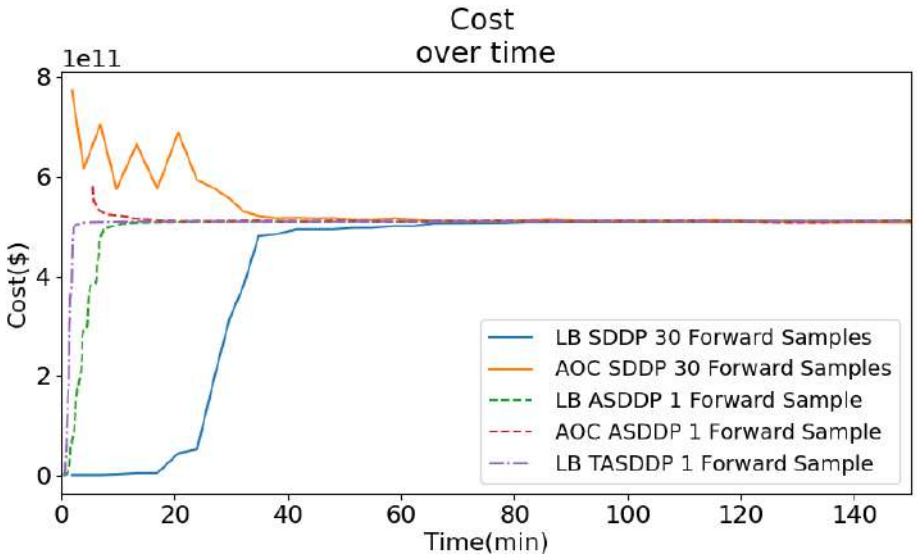

Figure 6.15: Evaluation of lower bound and the average operation cost over time for the long term case with 30 processors.

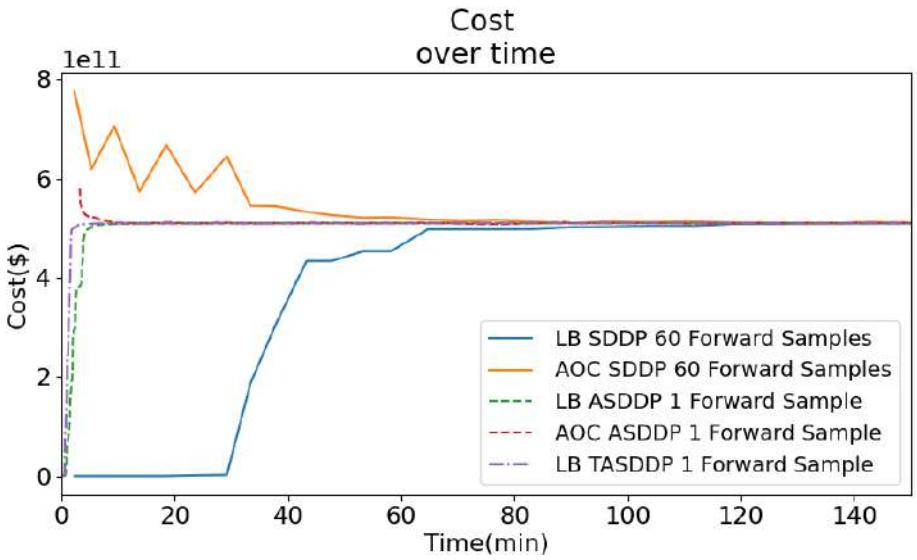

Figure 6.16: Evaluation of lower bound and the average operation cost over time for long term case with 60 processors.

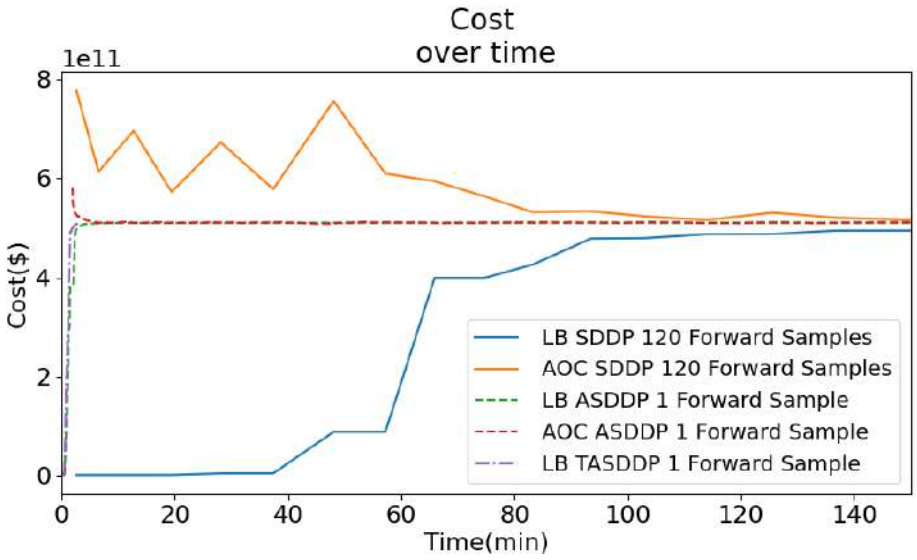

Figure 6.17: Evaluation of lower bound and the average operation cost over time for long term case with 120 processors.

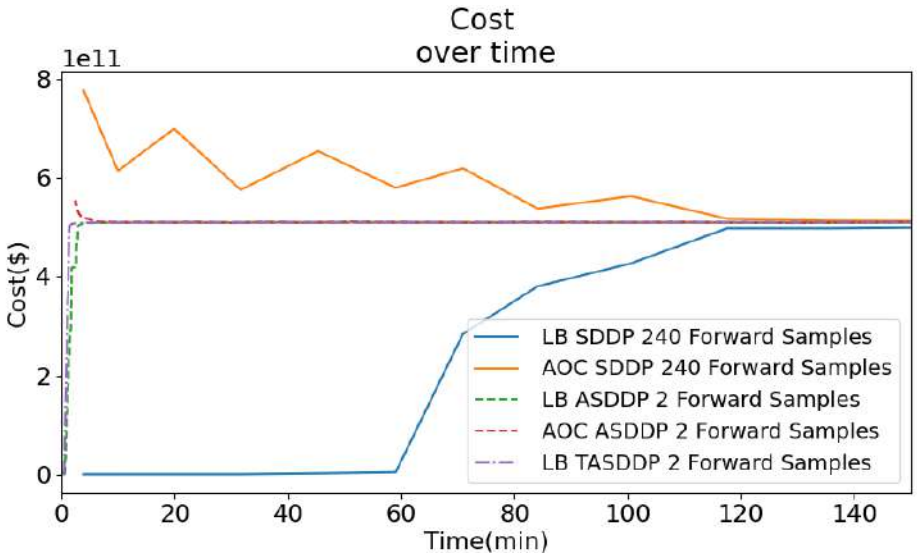

Figure 6.18: Evaluation of lower bound and the average operation cost over time for long term case with 240 processors.

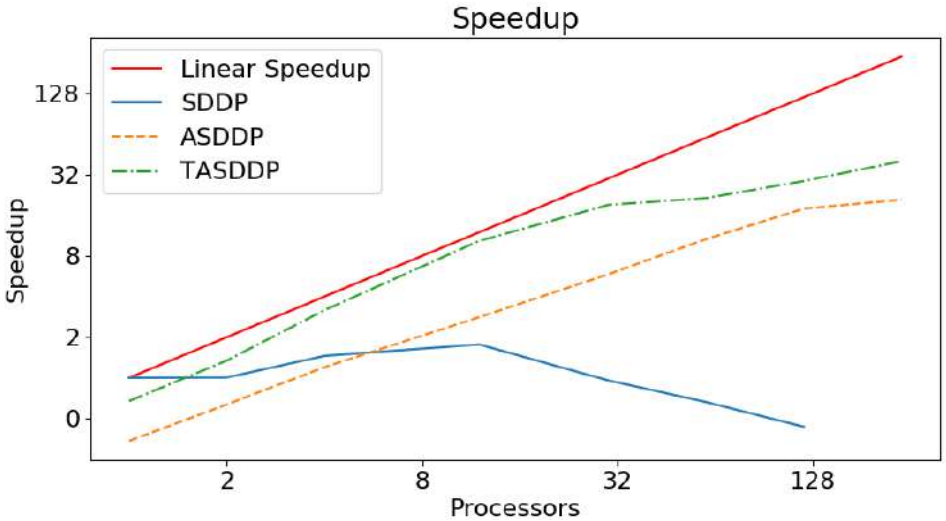

Figure 6.19: Speedup of long term case with log2 scale.

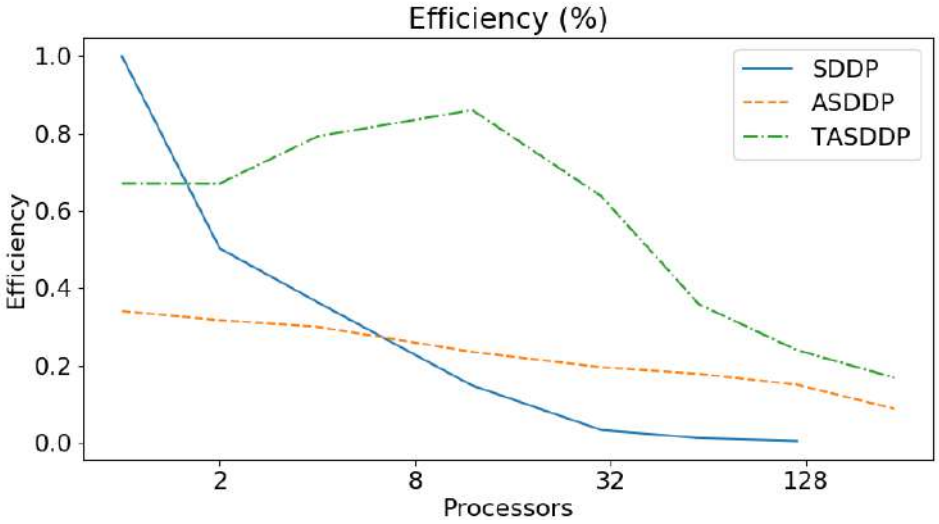

Figure 6.20: Efficiency of long term case with log2 scale.

6.5 Sensitivity analysis

Some aspects of the tree structure (number of backward scenarios) and input data (hydro inflows to the hydro plants) were modified in order to better evaluate the algorithms behavior.

6.5.1 Modifying the number of backward scenarios

Both mid and long terms test cases were modified regarding the number of backward scenarios: for the mid term case this was changed from 5 to 2 and 10 at each stage and for the long term case we reduced from 20 to 2 and 10. The number of processors to execute the problem takes into account the number of stages: 24 processors for the mid term case and 120 processors for the long-term case.

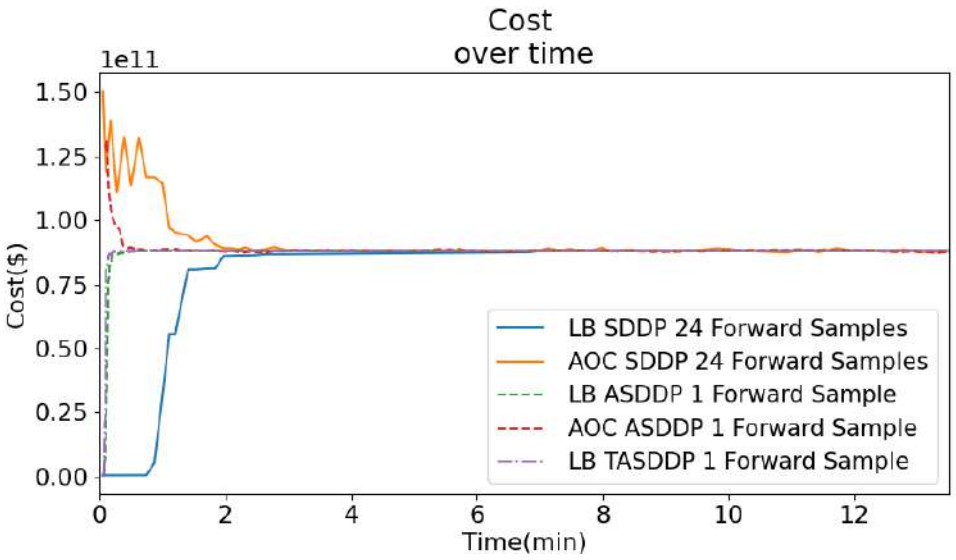

Figure 6.21: Evaluation of lower bound and the average operation cost over time for the mid term problem with 2 backward scenarios per stage and 24 processors.

Figures 6.21 and 6.22 show simulations with 2 and 10 backward scenarios for each stage, respectively. It can be seen that these simulations have the same behavior as the ones shown in Section 6.4 with 5 backward scenarios per stage.

Figures 6.23 and 6.24 show the same analysis for 2 and 10 backward scenarios of each stage for the long term case, where also the same behavior as in Section 6.4 (for the the case with 20 backward scenarios) is observed.

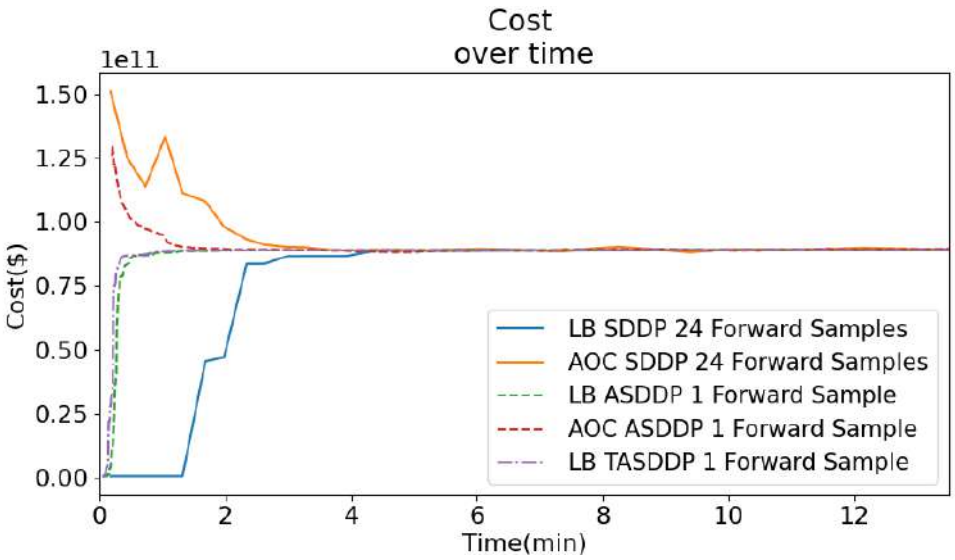

Figure 6.22: Evaluation of lower bound and the average operation cost over time for the mid term problem with 10 backward scenarios per stage and 24 processors.

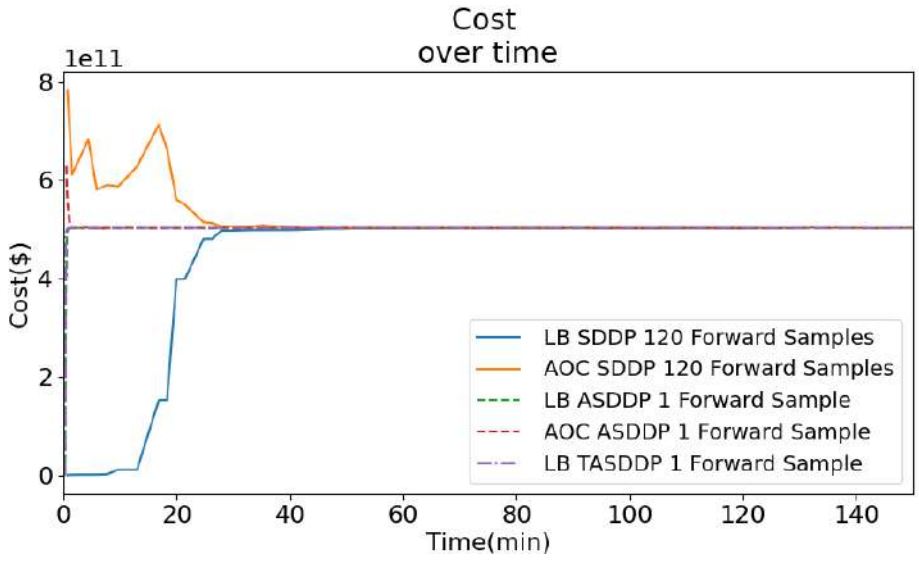

Figure 6.23: Evaluation of lower bound and the average operation cost over time for the long term problem with 2 backward scenarios per stage and 120 processors.

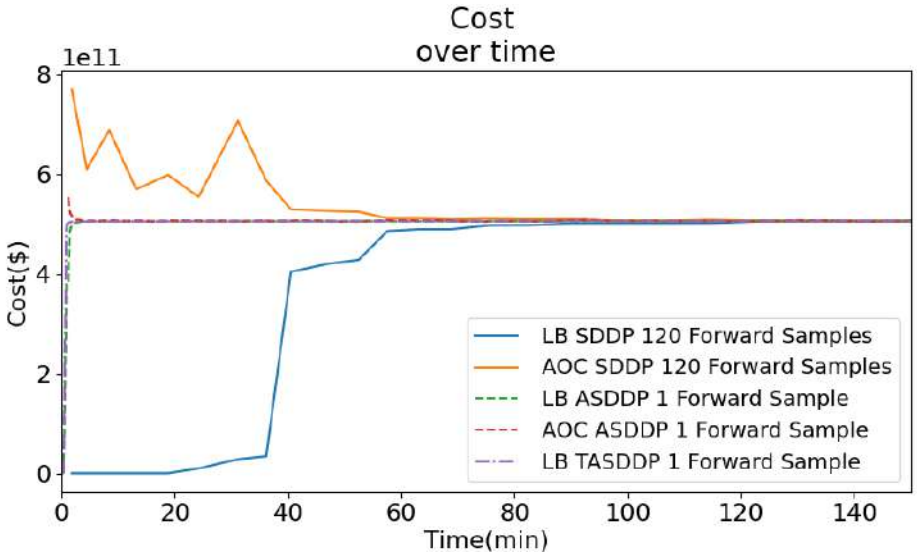

Figure 6.24: Evaluation of lower bound and the average operation cost over time for the long term problem with 10 backward scenarios per stage and 120 processors.

6.5.2 Modifying the number of forward scenarios

Since in the traditional parallel SDDP method the increase in the number of processors forces an increase in the number of forward scenarios, and the method was compared to the proposed ASDDP and TASDDP methods with the same number of processors (thus also forcing an increase in the number of forward samples for the other methods when the number of processors is greater than T), the results of this analysis have already been presented in Section 6.4. It can be noted that SDDP improves the convergence rate over iterations with the inclusion of new forward samples, but the time is also affected, as well as the algorithm performance. As expected, the convergence rate also improves on adding new forward samples for the ASDDP and and TASDDP methods, mostly for TASDDP, which does not need to wait all incoming data and thus has some gain with the inclusion of new Benders cuts.

6.5.3 Modification of inflow scenarios

All tests presented previously were executed forcing a set of very low inflow scenarios to evaluate the performance of the algorithms under more critical situations. In this session we asses the impact of inflow variability in the algorithms by modifying the inflow data, as explained below.

Long Term Average - percentage

It is common to use historical inflows as a reference to generate possible scenarios to the problem. One known measure is the so-called "long term average", which we refer to in the sequel as MLT, according to its Portuguese term "Média de Longo Termo". This is the average value for each month, taken from a large number of past years (since 1931 for most plants).

One possible variation is to consider, for example, empirical values in a range of 20% around the MLT. In this case we obtained the results shown in Figure 6.25 for the mid term case and in Figure 6.26 for the long term case. There are some differences in this behaviour as compared to the ones shown in Section 6.4: as expected, the inflows scenarios are better in this test, and both lower bound and the average operation cost reach stability earlier as compared to the previous test case with a critical situation, in all run tests to both mid and long term cases. All algorithms perform better but the relative time between SDDP and proposed algorithms (ASDDP and TASDDP) to reach this stability decreases. It seems that the algorithms performance is very sensitive to changes in the inflow scenarios.

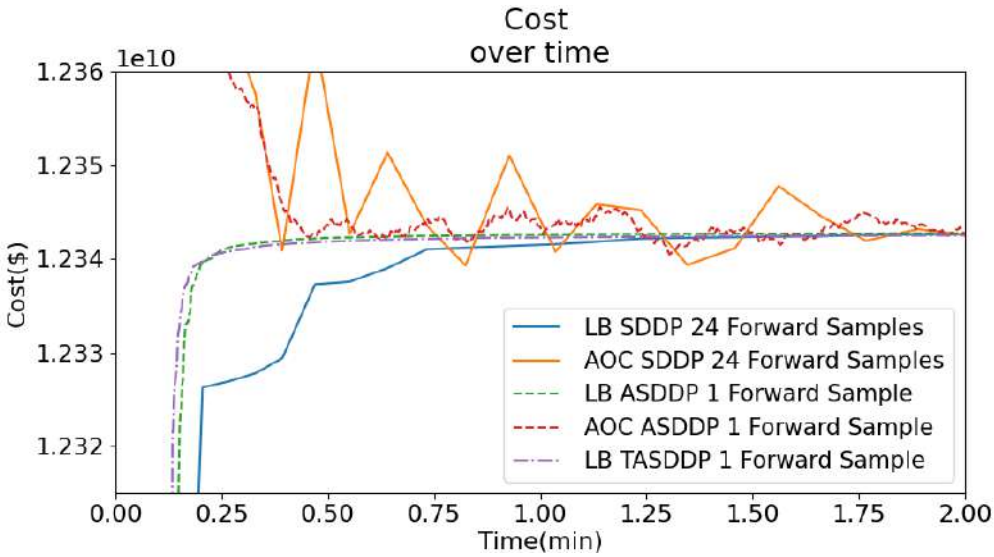

Figure 6.25: Sensitivity analysis for the mid term problem with 5 backward scenarios per stage and a variation of 20% around the MLT inflows.

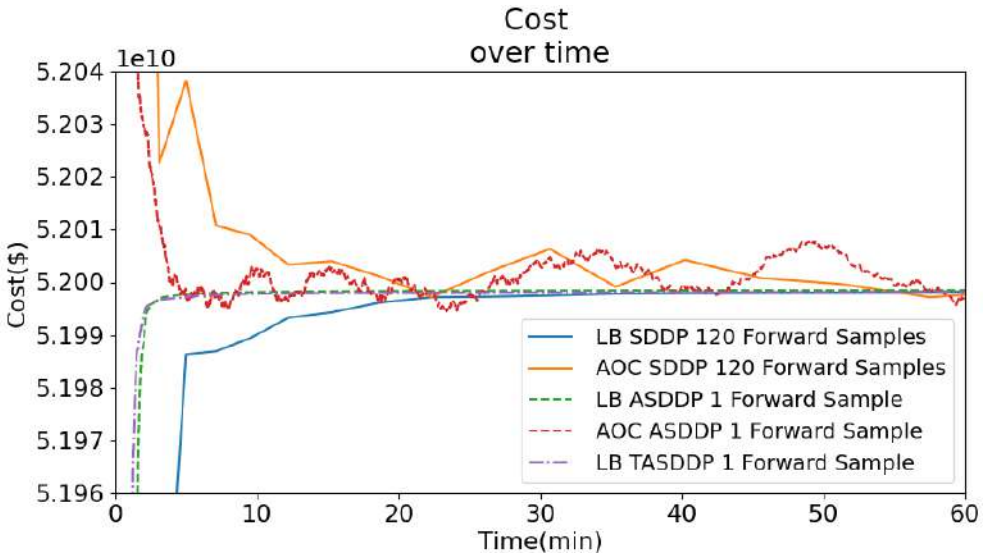

Figure 6.26: Sensitivity analysis for the long term problem with 20 backward scenarios per stage and a variation of 20% around the MLT inflows.

Long Term Average with log normal distribution

Another variation considered a more realistic case, using a log normal distribution [11] fitting parameters with MLT data [69]. The results are shown in Figure 6.27 for the mid term case and in Figure 6.28 for the long term case.

This test showed that there is a slight difference between using log normal distribution and using a range around 20% of MLT to model the hydro inflows. The average operation cost seems more difficult to stabilize, and lower bound stabilizes almost at same time for both mid and long term cases.

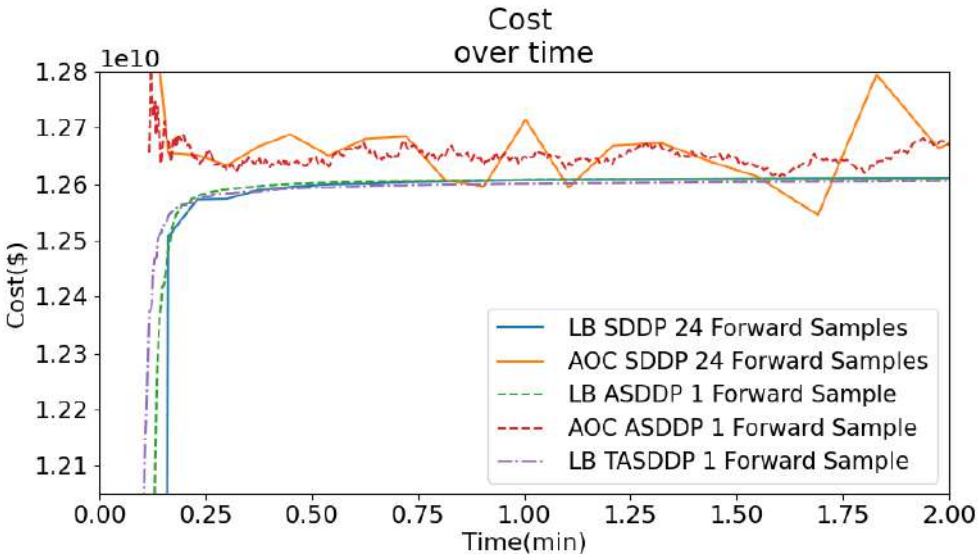

Figure 6.27: Sensitivity analysis for the mid term problem with 5 backward scenarios per stage and a log-normal distribution with average values equal to the MLT inflows.

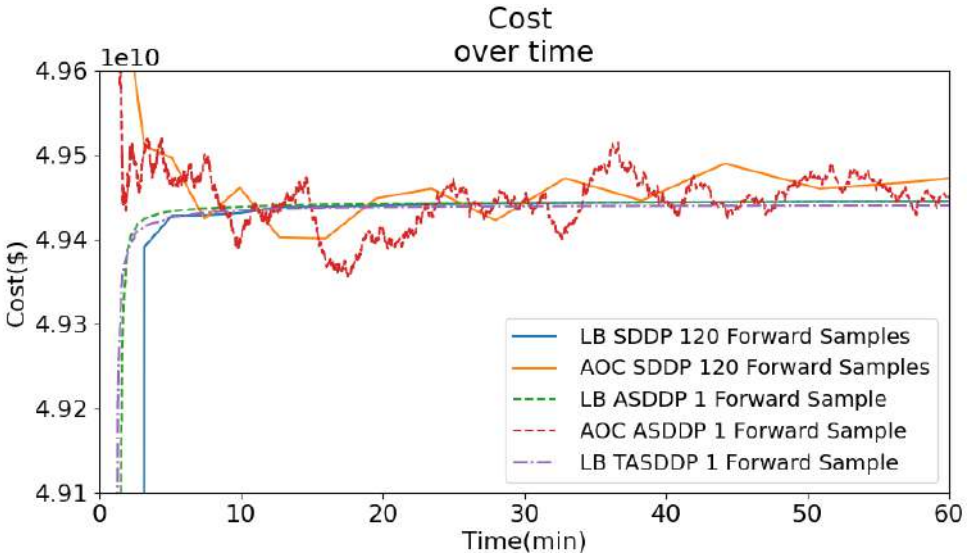

Figure 6.28: Sensitivity analysis for the long term problem with 20 backward scenarios per stage and a log-normal distribution with average values equal to the MLT inflows.

Chapter 7

Conclusions

The power generation planning is a difficult task due both to the modeling aspects of the large-scale stochastic programming problem and the difficult to solve it, mainly because of the large uncertainty in the natural inflows to the hydro plants. Even though in principle such optimization problem could be solved by its equivalent deterministic formulation, the problem can quickly become intractable, with a prohibitive increase in the computational time. This limitation can be overcome by applying decomposition methods like Dual Dynamic Programming (DDP) or sampling-based methods as Stochastic Dual Dynamic Programming (SDDP) to make better use of the computational resources.

This work proposes an asynchronous version of the SDDP algorithm that aims to overcome the drawback of the existing synchronization point between stages in the traditional parallel implementation of this method, with an application for the long term hydrothermal coordination (HTC) problem.

The proposed asynchronous SDDP approach performs backward and forward passes simultaneously and iterates by the so-called "steps" rather than full iterations, as described in Chapter 5. The work describes the proposed SDDP asynchronous approach called as Asynchronous Stochastic Dual Dynamic Programming algorithm (ASDDP) with a synchronization point between steps, and a variant called Totally Asynchronous ASDDP algorithm (TASDDP) that does not perform a full synchronization. ASDDP solves all subprobblems related to the forward and backward pass at each step for each stage and then communicates Benders cuts and state variables to the neighborhood stages. By contrast, the TASDDP variant proceeds to the next step as soon as a new cut from the subsequent stage or a new state variable from the previous stage has arrived. The results show that both approaches were capable of reducing the computational time to reach lower and upper bounds similar to those obtained with the traditional SDDP method, and have a high scalability regarding the number of processors that are employed.

In addition, the implementation of this work was made under an object-oriented

(OO) approach, to improve maintainability and reusability of the source code. The OO paradigm enables to reuse the same code regardless of the strategy applied to solve the problem. i.e.., whether a single Linear Problem for the equivalent-deterministic formulation, DDP, SDDP, or both variants of the proposed ASDDP approach is used, without the need to make specific treatments for each different method.

Considering the parallel programming implementation, other parallel models can be explored to improve scalability and better use of existing resources. A hybrid model using OpenMP and MPI [17] could be used to improve the memory use of cores on same node.

The performance of the proposed algorithms can still be improved in the future by including some features already presented in the literature, such as (i) the inclusion of a cut selection or elimination strategy [16], (ii) or the use of a multi-cut approach to build the Benders cut, in order to evaluate its impact in the convergence rate. Besides, tests with a more complex system are important to investigate whether the nice properties of the algorithm are maintained for different problem size and data. In addition, alternative upper bound estimation techniques can be employed to verify the quality of the operating policy [64].

Finally, in relation with the HTC problem itself, some improvements should be tested too. Due to the variation on the results by modifying the hydro inflows data, one could also assess the impact of the solving strategy when the hydrological trend is taken into account as additional state variables, as has already been considered in a number of works that employ the traditional SDDP parallel strategy, [42], or even more accurate representations of the stochastic process regarding the water inflows.

Bibliography

[1] ALEXANDRESCU, A., 2001, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional. ISBN: 0201704315.

[2] AMDAHL, G. M., 1967, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485, New York, NY, USA, april. ACM.

[3] ANON, 1993, "MPI: A message passing interface", *Proceedings of the Supercomputing Conference*, pp. 878–883. doi: <10.1109/SUPERC.1993.1263546>.

[4] AVELEDA, A. D. A., 2003, *"Utilização de sistemas de alto desempenho no processamento de sinais na análise de problemas de vibrações induzidas por desprendimento de vórtices em estruturas offshore"*. Ph.D. Thesis, COPPE/UFRJ.

[5] BENDERS, J. F., 1962, "Partitioning procedures for solving mixed-variables programming problems", *Numerische Mathematik*, v. 4, n. 1, pp. 238–252. doi: <10.1007/BF01386316>. Availability: <http://dx.doi.org/10.1007/BF01386316>.

[6] BIRGE, J. R., 1985, "Decomposition and Partitioning Methods for Multistage Stochastic Linear Programs", *Operations Research*, v. 33, n. 5, pp. 989–1007. doi: <10.1287/opre.33.5.989>.

[7] BIRGE, J. R., LOUVEAUX, F. V., 1988, "A multicut algorithm for two-stage stochastic linear programs", *European Journal of Operational Research*, v. 34, n. 3, pp. 384 – 392. doi: <http://dx.doi.org/10.1016/0377-2217(88)90159-2>.

[8] BOOST, 2020. "Boost". Availability: <https://www.boost.org/>.

[9] BRANDAO, L. C., DINIZ, A. L., SIMONETTI, L., 2018, "Accelerating Dual Dynamic Programming for Stochastic Hydrothermal Coordination Problems", *2018 Power Systems Computation Conference (PSCC)*, pp. 1–7.

[10] BRANDI, R. B. S., MARCATO, A. L. M., DIAS, B. H., et al., 2017. "A Convergence Criterion for Stochastic Dual Dynamic Programming: Application to the Long-Term Operation Planning Problem". doi: <10.1109/TPWRS.2017.2787462>.

[11] CHARBENEAU, R. J., 1978, "Comparison of the two- and three-parameter log normal distributions used in streamflow synthesis", *Water Resources Research*, v. 14, n. 1 (feb), pp. 149–150. doi: <10.1029/WR014i001p00149>.

[12] CHEN, Z. L., POWELL, W. B., 1999, "Convergent cutting-plane and partial-sampling algorithm for multistage stochastic linear programs with recourse", *Journal of Optimization Theory and Applications*, v. 102, n. 3, pp. 497–524. doi: <10.1023/A:1022641805263>.

[13] CLP TEAM, 2020. "Clp Coin OR". Availability: <https://github.com/coin-or/Clp>.

[14] DA SILVA, E., FINARDI, E. C., 2003, "Parallel processing applied to the planning of hydrothermal systems", *IEEE Transactions on Parallel and Distributed Systems*, v. 14, n. 8, pp. 721–729. doi: <10.1109/TPDS.2003.1225052>.

[15] DANTAS, F., 2011, "Reuse vs. maintainability: Revealing the impact of composition code properties", *Proceedings - International Conference on Software Engineering*, pp. 1082–1085. doi: <10.1145/1985793.1986001>.

[16] DE MATOS, V. L., PHILPOTT, A. B., FINARDI, E. C., 2015, "Improving the performance of Stochastic Dual Dynamic Programming", *Journal of Computational and Applied Mathematics*, v. 290, pp. 196 – 208. doi: <http://dx.doi.org/10.1016/j.cam.2015.04.048>.

[17] DIAS, B. H., TOMIM, M. A., MARCATO, A. L. M., et al., 2013, "Parallel computing applied to the stochastic dynamic programming for long term operation planning of hydrothermal power systems", *European Journal of Operational Research*, v. 229, n. 1, pp. 212–222. doi: <10.1016/j.ejor.2013.02.024>.

[18] DIAZ, J., MUÑOZ-CARO, C., NIÑO, A., 2012, "A survey of parallel programming models and tools in the multi and many-core era", *IEEE Transac-*

tions on Parallel and Distributed Systems, v. 23, n. 8, pp. 1369–1386. doi: <10.1109/TPDS.2011.308>.

[19] DINIZ, A. L., MACEIRA, M. E. P., 2008, "A Four-Dimensional Model of Hydro Generation for the Short-Term Hydrothermal Dispatch Problem Considering Head and Spillage Effects", *IEEE Transactions on Power Systems*, v. 23, n. 3 (Aug), pp. 1298–1308. doi: <10.1109/TPWRS.2008.922253>.

[20] DINIZ, A. L., SANTOS, T. N., 2013, "An Efficient Parallel Decomposition Approach for Stochastic Dual Dynamic Programming". In: *2013 ICSP - International Conference on Stochastic Programming*, july.

[21] DINIZ, A. L., COSTA, F. D. S., MACEIRA, M. E., et al., 2018, "Short/Mid-Term Hydrothermal Dispatch and Spot Pricing for Large-Scale Systems-the Case of Brazil". In: *2018 Power Systems Computation Conference (PSCC)*, pp. 1–7, June.

[22] DONOHUE, C. J., BIRGE, J. R., 2006, "The Abridged Nested Decomposition Method for Multistage Stochastic Linear Programs with Relatively Complete Recourse", *Algorithmic Operations Research*, v. 1, n. 1, pp. 20–30.

[23] DZAFIC, I., GLAVIC, M., TESNJAK, S., 2004, "A component-based power system model-driven architecture", *IEEE Transactions on Power Systems*, v. 19, n. 4 (Nov), pp. 2109–2110. doi: <10.1109/TPWRS.2004.836178>.

[24] FLYNN, M. J., 1966, "Very High-speed Computing Systems", *Proceedings of the IEEE*, v. 54, n. 12, pp. 1901–1909.

[25] FOSSO, O. B., GJELSVIK, A., HAUGSTAD, A., et al., 1999, "Generation scheduling in a deregulated system. The Norwegian case", *IEEE Transactions on Power Systems*, v. 14, n. 1 (Feb), pp. 75–81. doi: <10.1109/59.744487>.

[26] GAMMA, E., HELM, R., JOHNSON, R., et al., 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. ISBN: 0201633612.

[27] GIL, E., BUSTOS, J., RUDNICK, H., 2003, "Short-term hydrothermal generation scheduling model using a genetic algorithm", *IEEE Transactions on Power Systems*, v. 18, n. 4 (Nov), pp. 1256–1264. doi: <10.1109/TPWRS.2003.819877>.

[28] GUSTAFSON, J. L., 1988, "Reevaluating amdahl's law", *Communications of the ACM*, v. 31, n. 5, pp. 532–533. doi: <10.1145/42411.42415>.

[29] HAFEEZ, M., ASGHAR, S., MALIK, U. A., et al., 2011, "Survey of MPI Implementations". In: Cherifi, H., Zain, J. M., El-Qawasmeh, E. (Eds.), *Digital Information and Communication Technology and Its Applications*, pp. 206–220, Berlin, Heidelberg, june. Springer Berlin Heidelberg. ISBN: 978-3-642-22027-2.

[30] HELSETH, A., BRAATEN, H., 2015, "Efficient parallelization of the stochastic dual dynamic programming algorithm applied to hydropower scheduling", *Energies*, v. 8, n. 12, pp. 14287–14297. doi: <10.3390/en81212431>.

[31] HINDSBERGER, M., 2014, "ReSa: A method for solving multistage stochastic linear programs", *Journal of Applied Operational Research*, v. 6, n. 1, pp. 2–15.

[32] HITZ, M., MONTAZERI, B., 1995, "Measuring Coupling and Cohesion In Object-Oriented Systems", *Angewandte Informatik*, v. 50, pp. 1–10.

[33] HOMEM-DE MELLO, T., DE MATOS, V. L., FINARDI, E. C., 2011, "Sampling strategies and stopping criteria for stochastic dual dynamic programming: a case study in long-term hydrothermal scheduling", *Energy Systems*, v. 2, n. 1 (mar), pp. 1–31. doi: <10.1007/s12667-011-0024-y>.

[34] HUANG YONGJIE, LIU QIANG, GUI XUN, 2011, "Application of design pattern in MBIT ground software of flight control system". In: *2011 3rd International Conference on Computer Research and Development*, v. 2, pp. 80–84, March.

[35] HWANG, K., 1992, *Advanced Computer Architecture: Parallelism,Scalability,Programmability*. McGraw-Hill Higher Education. ISBN: 0070316228.

[36] INFANGER, G., MORTON, D. P., 1996, "Cut sharing for multistage stochastic linear programs with interstage dependency", *Mathematical Programming*, v. 75, n. 2 (nov), pp. 241–256. doi: <10.1007/BF02592154>.

[37] KAI HWANG, 1987, "Advanced parallel processing with supercomputer architectures", *Proceedings of the IEEE*, v. 75, n. 10 (Oct), pp. 1348–1379. doi: <10.1109/PROC.1987.13894>.

[38] KHRONOS GROUP, 2019. "OpenCL". Availability: <https://www.khronos.org/opencl/>.

[39] KOZMÍK, V., MORTON, D. P., 2015, "Evaluating policies in risk-averse multi-stage stochastic programming", *Mathematical Programming*, v. 152, n. 1-2, pp. 275–300. doi: <10.1007/s10107-014-0787-8>.

[40] LINOWSKY, K., PHILPOTT, A. B., 2005, "On the convergence of sampling-based decomposition algorithms for multistage stochastic programs", *Journal of Optimization Theory and Applications*, v. 125, n. 2, pp. 349–366. doi: <10.1007/s10957-004-1842-z>.

[41] MACEDONIA, M., 2003, "The GPU enters computing's mainstream", *Computer*, v. 36, n. 10 (oct), pp. 106–108. doi: <10.1109/MC.2003.1236476>.

[42] MACEIRA, M. E. P., BEZERRA, C. V., 1997, "Stochastic streamflow model for hydroelectric systems", *5th Int. Conf. on Probabilistic Methods Applied to Power Systems-PMAPS*.

[43] MACEIRA, M. E. P., PENNA, D. D. J., DINIZ, A. L., et al., 2018, "Twenty Years of Application of Stochastic Dual Dynamic Programming in Official and Agent Studies in Brazil-Main Features and Improvements on the NEWAVE Model". In: *2018 Power Systems Computation Conference (PSCC)*, pp. 1–7, June.

[44] MACEIRA, M., 1993, *Programação Dinâmica Dual Estocástica Aplicada ao Planejamento da Operação Energética de Sistemas Hidrotérmicos com Representação do Processo Estocástico de Afluências por Modelos Auto-Regressivos Periódicos*. Relatório Técnico 237/93, CEPEL.

[45] MACEIRA, M., TERRY, L., F.S.COSTA, et al., 2002, "Chain of optimization models for setting the energy dispatch and spot price in the Brazilian system", *Power System Computation Conference*, , n. June, pp. 24–28.

[46] MACEIRA, M., DUARTE, V., PENNA, D., et al., 2008, "Ten years of application of stochastic dual dynamic programming in official and agent studies in Brazil - Description of the NEWAVE program". In: *16th Power System Computation Conference*, july.

[47] MACEIRA, M., DUARTE, V., PENNA, D., et al., 2011, "An approach to consider hydraulic coupled systems in the construction of equivalent reservoir model in hydrothermal operation planning", (01).

[48] MIJAČ, M., STAPIĆ, Z., 2015, "Reusability Metrics of Software Components : Survey Reusability Metrics of Software Components : Survey", *26th Central European Conference on Information and Intelligent Systems (CECIIS 2015)*, , n. September. doi: <10.13140/RG.2.1.3611.4642>.

[49] MORTON, D. P., 1996, "An enhanced decomposition algorithm for multistage stochastic hydroelectric scheduling", *Annals of Operations Research*, v. 64, n. 1, pp. 211–235. doi: <10.1007/BF02187647>. Availability: <http://dx.doi.org/10.1007/BF02187647>.

[50] MPICH, 2020. "Mpich". Availability: <https://www.mpich.org/>.

[51] MU, H., JIANG, S., 2011, "Design patterns in software development", *ICSESS 2011 - Proceedings: 2011 IEEE 2nd International Conference on Software Engineering and Service Science*, pp. 322–325. doi: <10.1109/ICSESS.2011.5982228>.

[52] MUNKE, A., WERNER, J., REHM, W., 1996, "Parallel Programming Styles: A Brief Overview". In: *Computational Physics: Selected Methods Simple Exercises Serious Applications*, pp. 354–374, Berlin, Heidelberg, Springer Berlin Heidelberg. ISBN: 978-3-642-85238-1.

[53] NVIDIA CORPORATION, 2019. "CUDA Zone". Availability: <https://developer.nvidia.com/cuda-zone>.

[54] PANDE, A., GUPTA, M., TRIPATHI, A. K., 2010, "Design pattern mining for GIS application using graph matching techniques". In: *2010 3rd International Conference on Computer Science and Information Technology*, v. 3, pp. 477–482, July.

[55] PEREIRA, M. V. F., PINTO, L. M. V. G., 1991, "Multi-stage stochastic optimization applied to energy planning", *Mathematical Programming*, v. 52, n. 1, pp. 359–375. doi: <10.1007/BF01582895>.

[56] PHILPOTT, A. B., DE MATOS, V. L., 2012, "Dynamic sampling algorithms for multi-stage stochastic programs with risk aversion", *European Journal of Operational Research*. doi: <10.1016/j.ejor.2011.10.056>.

[57] PHILPOTT, A., GUAN, Z., 2008, "On the convergence of stochastic dual dynamic programming and related methods", *Operations Research Letters*, v. 36, n. 4, pp. 450 – 455. doi: <https://doi.org/10.1016/j.orl.2008.01.013>.

[58] PINTO, R. J., BORGES, C. T., MACEIRA, M. E. P., 2013, "An Efficient Parallel Algorithm for Large Scale Hydrothermal System Operation Planning", *IEEE Transactions on Power Systems*, v. 28, n. 4 (Nov), pp. 4888–4896. doi: <10.1109/TPWRS.2012.2236654>.

[59] PRESSMAN, R., 2009, *Software Engineering: A Practitioner's Approach*. 7 ed. USA, McGraw-Hill, Inc. ISBN: 0073375977.

[60] PRIYALAKSHMI, G., LATHA, R., 2018, "Evaluation of Software Reusability Based on Coupling and Cohesion", *International Journal of Software Engineering and Knowledge Engineering*, v. 28, n. 10, pp. 1455–1485. doi: <10.1142/S0218194018500420>.

[61] QIU, W., ZOU, W., SUN, Y., 2012, "Design patterns applied in power system analysis software package", *Proceedings of the 2012 International Conference on Industrial Control and Electronics Engineering, ICICEE 2012*, pp. 836–840. doi: <10.1109/ICICEE.2012.222>.

[62] SANTOS, T. N., DINIZ, A. L., BORGES, C. L. T., 2017, "A New Nested Benders Decomposition Strategy for Parallel Processing Applied to the Hydrothermal Scheduling Problem", *IEEE Transactions on Smart Grid*, v. 8, n. 3 (May), pp. 1504–1512. doi: <10.1109/TSG.2016.2593402>.

[63] SHAPIRO, A., TEKAYA, W., DA COSTA, J. P., et al., 2013, "Risk neutral and risk averse Stochastic Dual Dynamic Programming method", *European Journal of Operational Research*, v. 224, n. 2 (Jan), pp. 375–391. doi: <10.1016/j.ejor.2012.08.022>.

[64] SHAPIRO, A., TEKAYA, W., SOARES, M. P., et al., 2013, "Worst-Case-Expectation Approach to Optimization Under Uncertainty", *Operations Research*, v. 61, n. 6, pp. 1435–1449. doi: <10.1287/opre.2013.1229>.

[65] SHAWWASH, Z. K., SIU, T. K., DENIS RUSSELL, S. O., 2000, "The B.C. hydro short term hydro scheduling optimization model", *IEEE Transactions on Power Systems*, v. 15, n. 3, pp. 1125–1131. doi: <10.1109/59.871743>.

[66] SLYKE, R. M. V., WETS, R., 1969, "L-Shaped Linear Programs with Applications to Optimal Control and Stochastic Programming", *SIAM Journal on Applied Mathematics*, v. 17, n. 4, pp. 638–663. doi: <10.1137/0117061>.

[67] SOLINAS, M., ANTONELLI, L., 2013, "Software evolution and design patterns", *IEEE Latin America Transactions*, v. 11, n. 1, pp. 347–352. doi: <10.1109/TLA.2013.6502828>.

[68] SOMMERVILLE, I., MELNIKOFF, S. S. S., ARAKAKI, R., et al., 2008, *Engenharia de software*. ADDISON WESLEY BRA. ISBN: 9788588639287.

[69] STEDINGER, J. R., 1980, "Fitting log normal distributions to hydrologic data", *Water Resources Research*, v. 16, n. 3 (jun), pp. 481–490. doi: <10.1029/WR016i003p00481>.

[70] WARLAND, G., HENDEN, A. L., MO, B., 2016, "Use of Parallel Processing in Applications for Hydro Power Scheduling - Current Status and Future Challenges", *Energy Procedia*, v. 87, n. 1876, pp. 157–164. doi: <10.1016/j.egypro.2015.12.346>.

[71] WILKINSON, B., ALLEN, M., 2004, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2Nd Edition)*. Upper Saddle River, NJ, USA, Prentice-Hall, Inc. ISBN: 0131405632.