CIRRELT

CIRRELT-2019-49

An Asynchronous Parallel Benders Decomposition Method

Ragheb Rahmaniani
Teodor Gabriel Crainic
Michel Gendreau
Walter Rei

October 2019

Bureau de Montréal
Université de Montréal
C.P. 6128, succ. Centre-Ville
Montréal (Québec) H3C 3J7
Tél. : 1-514-343-7575
Télécopie : 1-514-343-7121

Bureau de Québec
Université Laval,
2325, rue de la Terrasse
Pavillon Palasis-Prince, local 2415
Québec (Québec) G1V 0A6
Tél. : 1-418-656-2073
Télécopie : 1-418-656-2624

An Asynchronous Parallel Benders Decomposition Method[ł]

Ragheb Rahmaniani[1,2], Teodor Gabriel Crainic[1,3,*], Michel Gendreau[1,4], Walter Rei[1,3]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT)

[2] Optimizaed Markets Inc., Pittsurgh, PA 15213, USA

[3] Department of Management and Technology, Université du Québec à Montréal, P.O. Box 8888, Station Centre-Ville, Montréal, Canada H3C 3P8

[4] Department of Mathematics and Industrial Engineering, Polytechnique Montréal, P.O. Box 6079, Station Centre-Ville, Montréal, Canada H3C 3A7

Abstract. Benders decomposition (BD) is one of the most popular solution algorithms for stochastic integer programs. The BD method decomposes stochastic problems into one master problem and multiple disjoint subproblems. It thus lends itself readily to parallelization. In almost all studies on the parallelization of this algorithm, the master problem remains idle until every subproblem is solved and vice versa. This can clearly result in having an extremely inefficient parallel algorithm due to excessive idle times. On the other hand, relaxing the synchronization requirement can yield a nonconvergent or less efficient algorithm that may even underperform when compared to the sequential version. Addressing these issues, we introduce an asynchronous parallel BD method. We show that the proposed algorithm converges to the global optima and suggest various acceleration strategies to enhance its performance. We conduct an extensive numerical study on benchmark instances from stochastic network design problems. The results indicate that our asynchronous algorithm reaches higher speedup rates compared to the conventional low-level parallel methods.

Keywords: Benders decomposition, branch-and-cut, stochastic integer programming, parallel computing.

Acknowledgements. Partial funding for this project has been provided by the Natural Sciences and Engineering Council of Canada (NSERC), through its Discovery Grant program and by the Fonds de recherche du Québec through its Team Grant program. We also gratefully acknowledge the support of Fonds de recherche du Québec through their strategic infrastructure grants. While working on this project, T. G. Crainic was also an adjunct professor in the Department of Computer Science and Operations Research of the Université de Montréal, and R. Rahmaniani was a PhD candidate at Polytechnique Montréal.

[ł]Revised version of the CIRRELT-2018-07

---

* Corresponding author: ragheb.rahmaniani@gmail.com

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2019

1. Introduction

Two-stage *stochastic integer programming* (SIP) offers a powerful tool to deal with uncertainties in planning problems where the distribution of uncertain parameters is assumed to be known and often characterized with a finite set of discrete scenarios (Birge and Louveaux, 1997). SIP models are, however, often very large in size and very difficult to solve due to the data uncertainty and their combinatorial and nonconvex nature (Ahmed, 2010). Therefore, their solvability constitutes one of the major research streams in the field.

Many efforts have been made to design various decomposition-based algorithms to break SIP models into more manageable pieces. Fortunately, in many cases the SIP models exhibit special structures that can effectively be exploited in various decomposition methods (Ruszczyński, 1997) such as Benders decomposition (BD) (Benders, 1962) also known as L-shaped method (Van Slyke and Wets, 1969), stochastic decomposition (Higle and Sen, 1991), nested decomposition (Archibald et al., 1999), subgradient decomposition (Sen, 1993), scenario decomposition (Rockafellar and Wets, 1991), disjunctive decomposition (Ntaimo, 2010), etc. Among these methods, BD has become a prominent methodology to address SIPs (Ruszczyński, 1997; Uryasev and Pardalos, 2013).

The BD method decomposes the two-stage SIP model into a *master problem* (MP) and a *subproblem* (SP) for each scenario. The algorithm then follows an iterative procedure where at each iteration, the MP is solved and its solution is fixed in each SP. Then, each SP is solved to generate an optimality or feasibility cut. The generated cuts are gathered and added to the MP and the same procedure is repeated until an optimal solution is found.

Solving the master problem at each iteration for SIP models can be very time-consuming. Therefore, the algorithm is commonly cast into a *branch-and-cut* (B&C) framework in order to avoid solving an integer problem from scratch at each iteration. Thus, a single branch-and-bound tree is built and the cuts are generated at the integer (and possibly some fractional) nodes (Rahmaniani et al., 2017a, 2018). Likewise, solving the SPs at each iteration usually corresponds to the most time-consuming part of the algorithm, because a large number of scenarios is often required to properly characterize the uncertain parameters. These SPs are disjoint and can be solved in parallel. Thus, *parallel computing* appears very promising to effectively accelerate solving the SIP problems when the BD method is used (Linderoth and Wright, 2003; Li, 2013).

Although parallel processing saves time, the processors at some point need to exchange information and consolidate the results in order to create work units for the next iteration. In the literature on parallel BD methods, these synchronization points are implemented in every iteration of the algorithm, that is, at each iteration the processor that solves the MP waits for the processors that solve the SPs and vice versa. The need for this frequent synchronization is due to the interdependency of the MP and SPs. This synchronized parallelization, however, causes having one or several idle processors at any given time, particularly when obtaining a master solution can take hours (Yang et al., 2016) or solving the SPs is slow (Linderoth and Wright, 2003). As a result, the efficiency of the parallel execution may decrease as the number of processors increases. It is thus important to design new parallelization schemes for the BD method to obtain a high-performing algorithm.

An evident strategy to design parallel BD methods, specially when the MP becomes the major computational bottleneck, is to implement the BD method in a B&C framework and possibly parallelize the branch-and-bound tree. In this regard, we are only aware of the study by Langer et al. (2013). However, even in such algorithms, the interdependency of the MP and SPs is not addressed and thus, the parallelization may still suffer from excessive idle times due to synchronizing each master processor with the slave processors.

In this article, we aim at developing effective parallelization strategies for the B&C implementation of the BD method. To avoid burdening this article, we limit ourselves to the case where the SPs are optimized in parallel on various *slave* processors and the MP is solved sequentially on a single *master* processor. Our goal is to introduce a parallel framework where the synchronization among the master and slave processors is not required. To the best of our knowledge, this is the first study to address this issue. Therefore, our study can also serve as a building block toward having more effective multi-master and multi-slave parallel Benders algorithms.

It is important to note that our parallelization framework is different from the existing parallel B&C algorithms. In the parallel B&C methods the main emphasis is on parallelizing the branch-and-bound tree and the cuts are generated to accelerate the convergence only (Ralphs et al., 2003; Crainic et al., 2006), while in the parallel BD methods, the main emphasis is on parallelizing the SPs and the cuts are *necessary* for convergence. Thus, the parallelization strategies of the B&C algorithms cannot easily be translated into parallel BD methods.

To achieve our goal, we relax the synchronization requirements among the master and slave processors. This means that the master processor may or may not wait for cuts to be generated by the slave processors at each node of the branch-and-bound tree. Although this significantly reduces the idleness of the master and slave processors, it results in an algorithm for which we are unable to prove global convergence. This happens because, in absence of synchronization, the necessary cuts at the integer nodes may not be applied at the right moment. We thus study this issue and show that with an appropriate B&C design the algorithm can converge. On the other hand, the asynchronous algorithm may execute a large amount of *redundant work* since the MP keeps branching on the master variables with a *partial* feedback from the SPs. This can cause serious efficiency problems such that the algorithm may even underperform when compared to the sequential variant. Therefore, we propose various acceleration techniques to obtain an efficient asynchronous parallel BD algorithm. The main contributions of this article are thus severalfold:

- Proposing an effective Benders-based *asynchronous* parallel B&C algorithm for two-stage SIP problems;
- Studying convergence properties and various variants of the algorithm;
- Discussing several factors that influence the design and performance of asynchronous parallel BD algorithms and proposing new strategies to accelerate the computational performance of our algorithm;

- Presenting extensive numerical results on benchmark instances to assess the proposed strategies and algorithms. We also provide guidelines on when to use different variants of the proposed parallelization technique and discuss various fruitful research directions.

The remainder of this article is organized as follows. In section 2, the problem of interest and the sequential BD algorithm are presented. In Section 3, we classify and review parallel BD methods. We present our asynchronous parallel algorithm and the proposed acceleration techniques in Sections 4 and 5. We discuss the implementation details and numerical results in Sections 6 and 7. Finally, conclusions and future remarks are summarized in the last section.

2. The Benders Decomposition Method

We first recall the two-stage stochastic problems of interest. Then, we present a sequential BD algorithm to solve it.

2.1. Two-Stage Stochastic Integer Programming

In two-stage stochastic programming, the uncertainty is observed only once and decisions are made before and after observing the uncertainty. The common practice is to assume that the probability distributions of random variables is characterized with a finite set of scenarios, each representing a possible realization of the random events with a known probability. Given the scenario set $\mathcal{S}$ and occurrence probability $\rho_s > 0$ for each $s \in \mathcal{S}$ such that $\sum_{s\in\mathcal{S}} \rho_s = 1$, a canonical representation of a two-stage stochastic program is:

$$z^* := \min_y \{f^\top y + \sum_{s\in\mathcal{S}} \rho_s Q(y,s) : \ By \geq b, \ y \in \mathcal{Y}\}, \tag{1}$$

where for each scenario $s \in \mathcal{S}$ the recourse problem $Q(y,s)$ is defined as:

$$Q(y,s) = \min_x \{c_s^\top x : W_s x \geq h_s - T_s y, \ x \in \mathcal{X}\}, \tag{2}$$

with $f \in \mathbb{R}^n$, $B \in \mathbb{R}^{k\times n}$, $b \in \mathbb{R}^k$, $c_s \in \mathbb{R}^m$, $W_s \in \mathbb{R}^{l\times m}$, $h_s \in \mathbb{R}^l$, $T_s \in \mathbb{R}^{l\times n}$. Here, $\mathcal{X} \subseteq \mathbb{R}^m$ and $\mathcal{Y} \subseteq \mathbb{R}^n$ are nonempty closed sets which define the nature of the x and y decision variables in terms of sign and integrality restrictions. In this article we make the conventional assumption that program (1) is bounded, $\mathcal{Y} \equiv \{0,1\}^n$, and $\mathcal{X} \equiv \mathbb{R}^m_+$. Note that any problem with non-binary integer first-stage variables can be approximated with a binary problem to a desired precision and without increasing the problem size by too much (Zou et al., 2017). In this program, y represents the first-stage decisions and x represents the second-stage (or recourse) decisions. We thus seek a feasible solution that minimizes the first-stage cost $f^\top y$ plus the expected recourse cost.

2.2. Sequential Benders Decomposition Method

For a tentative value of the first-stage variables $\bar{y}$, the recourse problem $Q(\bar{y}, s)$ is a continuous linear program. Given a dual variable α associated with the constraint $W_s x \geq h_s - T_s \bar{y}$, the dual of $Q(\bar{y}, s)$ is:

$$(SP(\bar{y}, s)) \quad Q(\bar{y}, s) = \max_{\alpha} \{(h_s - T_s \bar{y})^\top \alpha : \ {W_s}^\top \alpha \leq c_s, \ \alpha \in \mathbb{R}^l_+\}. \tag{3}$$

The above program is either unbounded or feasible and bounded. In the former case, the $\bar{y}$ solution is infeasible and thus there exists a direction of unboundedness $r_{q,s}$, $q \in F_s$ that satisfies $(h_s - T_s y)^\top r_{q,s} > 0$, where F_s is the set of extreme rays of (3). To assure the feasibility of the y solutions, we need to forbid the directions of unboundedness through imposing $(h_s - T_s y)^\top r_{q,s} \leq 0, \ q \in F_s$, on the y variables, which gives:

$$z^* = \min_{y \in \mathbb{Z}^n_+} \{f^\top y + \sum_{s \in \mathcal{S}} \rho_s Q(y, s) : \ By \geq b, \ (h_s - T_s y)^\top r_{q,s} \leq 0 \ \forall s \in \mathcal{S}, q \in F_s\}. \tag{4}$$

In the latter case, the optimal solution of the SP is one of its extreme points $\alpha_{e,s}, \ e \in E_s$, where E^s is the set of extreme points of (3). We can thus rewrite program (4) in an extensive form by interchanging $Q(y, s)$ with its dual, i.e., $SP(y, s)$:

$$\min_{y \in Y} \big\{ f^\top y + \sum_{s \in S} \rho_s \max_{e \in E_s} \{(h_s - T_s y)^\top \alpha_{e,s}\} : \ (h_s - T_s y)^\top r_{q,s} \leq 0 \quad s \in S, q \in F_s \big\}, \tag{5}$$

where $Y = \{y : By \geq b, y \in \mathbb{Z}^n_+\}$. Capturing value of the inner maximization in a single variable θ_s for every $s \in \mathcal{S}$, we can obtain the following equivalent reformulation of (1), called Benders *master problem* (MP):

$$MP(E_1, ..., E_{|\mathcal{S}|}; F_1, ..., F_{|\mathcal{S}|}) := \min_{y \in Y} f^\top y + \sum_{s \in S} \rho_s \theta_s \tag{6}$$

$$(h_s - T_s y)^\top \alpha_{e,s} \leq \theta_s \qquad s \in S, e \in E_s \tag{7}$$

$$(h_s - T_s y)^\top r_{q,s} \leq 0 \qquad s \in S, q \in F_s \tag{8}$$

Enumerating all the extreme points E_s and extreme rays F_s for each SP $s \in S$ is computationally burdensome and unnecessary. Thus, Benders (1962) suggested a delayed constraint generation technique to extract the *optimality* (7) and *feasibility* (8) cuts on the fly.

In the classical BD method, the MP is solved from scratch at each iteration. However, nowadays, this method is usually implemented in a B&C framework to avoid solving a MP at each iteration (Naoum-Sawaya and Elhedhli, 2013; Rahmaniani et al., 2017b). Moreover, the LP relaxation of the MP is often solved first to quickly tighten the root node (McDaniel and Devine, 1977). Algorithm 1 presents the pseudo-code of this algorithm.

The algorithm is applied in two phases. It first starts with solving the LP relaxation of the MP, which initially includes no optimality and feasibility cuts (lines 1 and 2). Thus, it sequentially solves the MP and SPs to generate optimality or feasibility cuts until a stopping condition (e.g., time limit, maximum number of iterations, or optimality gap) is activated .

Algorithm 1 : The sequential Branch-and-Benders-cut method

1: Create the MP which is the LP relaxation of program (6)-(8) with $E_s = \emptyset$ and $F_s = \emptyset$, $\forall s \in \mathcal{S}$
2: Iteratively solve the MP and add cuts until a stopping criteria is satisfied
3: Add the obtained MP into tree $\mathcal{L}$, set $UB = \infty$
4: **while** *no stopping condition is met* **do**
5: Select node l from $\mathcal{L}$
6: Solve this node to get an optimal solution $\bar{y}$ with objective value of ϑ^*
7: **if** *node was infeasible* **or** $\vartheta^* \geq UB$ **then**
8: Prune the node and go to line 4
9: **if** $\bar{y}$ *is integer* **then**
10: **while** *a violating cut can be found* **and** *the* $\bar{y}$ *is integer* **do**
11: **for** $s \in \mathcal{S}$ **do**
12: Solve $SP(\bar{y}, s)$
13: **if** $SP(\bar{y}, s)$ *was infeasible* **then**
14: Extract the extreme ray r_s
15: **else**
16: Extract the optimal extreme point e_s
17: Add the cuts obtained in lines 14 and 16 to the master formulation
18: Solve this node again to get an optimal solution $\bar{y}$ with objective value of ϑ^*
19: **if** *node was infeasible* **or** $\vartheta^* \geq UB$ **then**
20: Prune the node and go to line 4
21: **if** $\bar{y}$ *is integer* **then**
22: Set $UB = \min\{UB, \vartheta^*\}$, prune the node, and go to line 4
23: Choose a fractional variable from $\bar{y}$ to branch
24: Create two nodes and add them to $\mathcal{L}$
25: Remove l from $\mathcal{L}$.

In the second phase, a branch-and-bound tree is created (line 3), where the root node is the MP with the cuts generated in the first phase. At each iteration, the algorithm selects and solves a node from the pool $\mathcal{L}$ (lines 5 and 6). When the obtained solution is fractional and the node cannot be pruned (line 7), a branching occurs to create and add two new nodes to the pool (lines 23 and 24). When the node cannot be pruned (line 7) and its solution is integer (line 9), the algorithm iteratively adds cuts until it can either prune that node, the node solution becomes fractional, or no more violated cut can be found (lines 10 to 20). This process (lines 5 to 25) repeats until a stopping condition is satisfied or the node pool $\mathcal{L}$ becomes empty. Note that in line 22, the upper bound is updated, i.e., the potential incumbent is accepted if $\bar{y}$ is an integer solution that satisfies all the Benders cuts and the associated cost is lower than the current incumbent value.

Our developments in this article are based on Algorithm 1, which we refer to as the *Branch-and-Benders-cut* (B&BC) method. More specifically, our goal is to generate the cuts

in parallel and avoid the inner while-loop so as to improve the overall efficiency. Various acceleration techniques are proposed to boost the BD algorithm. To avoid burdening this article, we refer the reader to Rahmaniani et al. (2017a) for a complete overview of the classical acceleration techniques. In the next section, we present an extensive review of the parallel BD methods.

3. Parallelization Strategies and Previous Work

The BD method lends itself readily to parallelization as it creates a MP and many disjoint SPs. Thus, efforts have been made to take advantage of parallel computing to accelerate this method. To the best of our knowledge, the existing parallel variants of this method follow the master-slave parallel programming model. We classify and review such methods in this part.

3.1. Master-slave parallelization strategies

Almost all the existing parallel BD methods can be summarized as follows: *The MP is assigned to a processor, the "master", which also coordinates other processors, the "slaves", which solve the SPs. At each iteration, the solution obtained from solving the MP is broadcast to the slave processors. They then return the objective values and the cuts obtained from solving the SPs to the master. The cuts are added to the MP model and the same procedure repeats.* Such master-slave parallelization schemes are known as *low-level parallelism* as they do not modify the algorithmic logic or the search space (Crainic and Toulouse, 1998).

In many cases, the number of the processors is less than the number of SPs and some SPs may be more time consuming than others. Therefore, it is important to take into account how work units are allocated to the processors to avoid having idle processors.

To create work units for the next iteration, information must be exchanged among the processors which necessitates some sort of communication to share information. The type of communications strongly influences the design of parallel BD algorithms. For example, if communications are synchronized, the only difference between the parallel algorithm and the sequential one lies in solving the SPs in parallel. Whereas, in asynchronous communications, the processors are less interdependent and many new factors must be taken into account while designing the parallelization framework. For example, after broadcasting each master solution, the master processor may or may not wait for cuts before branching. Also, some of the SPs associated with the current master solution may remain unsolved because the master processor may generate a new solution before the slave processors evaluate all the current SPs. Likewise, the slave processors may generate many cuts (associated to the current and/or previous solutions) while the master processor is solving the MP. Therefore, it is important to decide when to solve the MP, which solution to use in generating cuts, which SP to solve now, which cut to apply to the MP, and so on. We thus define a three-dimension taxonomy, depicted in Figure 1, that captures all these factors.

- ***Communication***: defines whether the processors at each iteration stop to communicate (*synchronous*) or not (*asynchronous*);

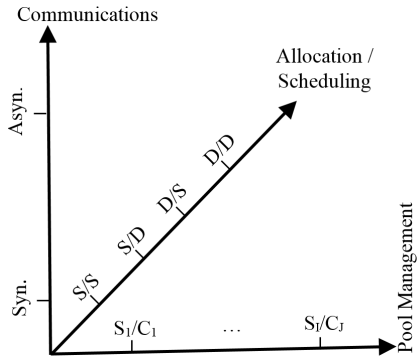

Figure 1: Taxonomy of the master-slave parallel Benders decomposition methods

- ***Allocation/Scheduling***: determines how the SPs and the MP are assigned to the processors and when they are solved. These decisions can be made either *dynamically* (D) or *statically* (S). In the former case, the decisions regarding when and where to solve each problem are taken during the course of the algorithm. In the latter case, the decisions are made beforehand. Thus, S/D, for example, means static allocation and dynamic scheduling;
- ***Pool Management*** implies the strategies used to manage the pool of solutions (denoted by $S_1, ..., S_I$) and the pool of cuts (denoted by $C_1, ..., C_J$), where I and J are the number of possible strategies to manage each pool, respectively.

A combination of various alternatives for these components yields a parallel BD algorithm. Proper strategies in each dimension need to be defined such that the overall *idle times* and amount of *redundant work* are minimized.

3.2. Previous work

Most of the parallel BD algorithms are developed for stochastic problems as the decomposition creates multiple disjoint SPs (Ariyawansa and Hudson, 1991; Wolf and Koberstein, 2013; Tarvin et al., 2016). Although parallelization of the BD method for SIP models seems natural, the number of existing parallel variants of this method is very limited. This is mainly due to the interdependency of the MP and the SPs, i.e., the MP needs feedback from the SPs before being able to execute its next iteration and vice versa. The literature discusses some of the strategies and algorithmic challenges arising from parallelizing the BD method.

Dantzig et al. (1991) considered a dynamic work allocation strategy in which the next idle processor gets the next SP based on a first-in-first-out strategy until all the SPs are solved and the MP can be recomputed with the new cuts. The efficiency of this parallel algorithm did not exceed 60% even on machines with 64 processors. Similarly, Li (2013) observed that dynamic work allocation is superior to static work allocation, because it reduces idle times and also it saves executing extra work. For example, if a SP is infeasible, the evaluation of the SPs remaining in the queue will be terminated and the feasibility cut generation scheme will be launched.

Nielsen and Zenios (1997) exploited the structural similarities of the SPs by applying an interior point algorithm on a fine-grained parallel machine. They decided to sequentially solve the MP to optimality from scratch at each iteration because it could easily be handled by means of an interior point method. Vladimirou (1998) implemented a partial-cut aggregation strategy to reduce the communication overheads.

The decomposition has been modified in some studies to better suit the parallelization scheme. Dempster and Thompson (1998) proposed a parallel nested algorithm for multi-stage stochastic programs. The authors used stage aggregation techniques to increase the size of the nodes and therefore the time spent on calculations relative to the time spent communicating between processors. In a similar spirit, Chermakani (2015) observed that when the number of SPs is considerably larger than the number of available processors, so that some SPs must be solved sequentially, it may be better to aggregate some of them. Latorre et al. (2009) modified the decomposition scheme for multi-stage stochastic programs such that the subsets of nodes assigned to each SP may overlap. This, unlike former studies, allows to noticeably reduce the dependency among SPs at each iteration, because they can be solved at the same time.

All the reviewed studies implement synchronized parallelism. Moritsch et al. (2001) proposed a prototype for an asynchronous nested optimization algorithm. However, no specific strategy or numerical results were presented. Linderoth and Wright (2003) implemented an asynchronous communication scheme in which the MP is re-optimized as soon as a portion of the cuts are generated. Testing this algorithm on LP stochastic programs with up to 10^7 scenarios, the authors observed a time reduction up to 83.5% on a computational grid compared to the sequential variant.

The speedup rates of the low-level parallelizations are often limited. Pacqueau et al. (2012) observed that solving SPs accounts for more than 70% of the total time requirement, while they merely observed a speedup ratio up to 45% with 4 processors. Furthermore, low-level parallelism improves the efficiency when the MP solving does not dominate the solution process. Yang et al. (2016) observed that the benefit of parallelism fades away with the scale of the problem because the computational effort is dominated by solving the MP. To cope with this issue, Langer et al. (2013) proposed to parallelize the branch-and-bound tree where each processor solves a partition of the tree. To further improve the scalability, the authors also proposed to solve the SPs in parallel, where a FIFO queue is used in picking and solving the SPs. Moreover, in this parallelization, each node after evaluation is added to a waiting list until all its cuts are generated. Langer et al. (2013) observed that a better performance in solving aircraft allocation problem can be obtained when the master processors share the cuts.

Some of the common knowledge for the sequential algorithm may not apply to its parallel variants. For example, Wolf and Koberstein (2013) pointed out that the single-cut version benefits from parallelization more than the multi-cut version because more SPs need to be solved in the former case. They also observed that the single-cut method may outperform the multi-cut variant because it requires less cuts in general, although it executes a larger number of iterations.

To conclude this part, we summarize the literature in Table 1. We observe that the

Table 1: Summary of the Benders-based parallel algorithms

Reference	Problem class		Implementation		Communication		Work allocation		Scheduling		Pool management	
	LP	MIP	Classic	B&C	Syn.	Asyn.	Dynamic	static	Dynamic	Static	Solution	Cut
Chermakani (2015)	+		+		+			+		+		
Dantzig et al. (1991)	+		+		+		+			+		
Dempster and Thompson (1998)	+		+		+			+		+		
Langer et al. (2013)		+		+	+		+			+		+
Latorre et al. (2009)		+	+		+		+			+		
Li (2013)		+	+		+		+			+		
Linderoth and Wright (2003)	+		+			+		+	+		+	+
Moritsch et al. (2001)	+		+			+		+	+		+	
Nielsen and Zenios (1997)	+		+		+			+		+		
Pacqueau et al. (2012)		+	+					+		+		
Vladimirou (1998)	+		+		+			+		+		
Wolf and Koberstein (2013)	+	+	+		+		+			+		
Yang et al. (2016)		+	+		+			+		+		
Mateo et al. (2018)	+		+		+			+		+		

literature on parallel BD algorithms is sparse. Few studies consider integer programs and only one has implemented the parallel algorithm in the B&C format. Moreover, only two studies consider asynchronous communications and these are developed for linear continuous problems. Also, we realized that synchronized parallelism is suitable when the MP can be solved quickly and the SPs constitute the major computational bottleneck of the algorithm.

Moreover, none of these studies addresses the interdependency of the MP and SPs, particularly for SIPs and when the BD method is casted into a B&C framework. Thus, the reviewed strategies are not directly applicable to develop a truly efficient parallelization of the BD method.

4. Asynchronous Parallel Benders Decomposition Algorithm

The synchronization requirement between master and slave processors increases the overheads due to the excessive idle times. This is particularly evident when solving any of the problems (MP or SP) is noticeably more time consuming than the others.

Our strategy to deal with this issue is to obtain an asynchronous algorithm by relaxing the interdependency among the master and slave processors. To do so, the master processor is required to wait only for $100\gamma\%$ of the cuts from slave processors before executing its next iteration, where $0 \leq \gamma \leq 1$. However, for any $\gamma < 1$, the B&BC method may fail to converge and also, it may increase the amount of redundant work such that the parallel algorithm might underperform when compared to the sequential algorithm. In this section, we address these issues to obtain an effective and convergent asynchronous parallel Benders algorithm . In this regard, we detail the considerations and propose modifications to Algorithm 1 and conclude the section with a complete pseudo-code of the proposed asynchronous algorithm.

4.1. Convergence

To ensure convergence of the B&BC method, the inner while-loop in Algorithm 1 cannot be stopped prematurely. Otherwise, an incumbent solution might be accepted in line 22 which does not necessarily satisfy all the Benders cuts and, thus, the optimal value might be underestimated. For this reason, straightforward asynchronism may compromise the convergence as it applies only a subset of the cuts at each iteration of the while-loop. To

cope with this issue, we need to keep track of the evaluated SPs for each master solution and update the incumbent value only when all the SPs are evaluated and the obtained bound is lower than the current incumbent value.

In the B&BC, the incumbent value is actively used to prune nodes. In the asynchronous algorithm, depending on the γ value, potential incumbent solutions are partially evaluated when their associated node is being processed. This creates a delay between finding an incumbent solution and obtaining its true value. Thus, to ensure that we are not exploring some nodes which might have been pruned if we had the true incumbent value at the right moment, we use the best-first search strategy in exploring the tree.

Asynchronous algorithms raise another issue when the integer node solution remains unchanged, i.e., the $\bar{y}$ solution in lines 9 and 21 are the same, after applying $\gamma\%$ of the cuts. In this case, we cannot prune the node based on the integrality rule at line 22 unless we are sure that the $\bar{y}$ satisfies all the Benders cuts. An effective remedy for this issue is to make use of combinatorial cuts of the form:

$$\sum_{i\in\{1,..,n\}:\bar{y}_i=0} y_i + \sum_{i\in\{1,..,n\}:\bar{y}_i=1} (1-y_i) \geq 1, \tag{9}$$

to forbid regeneration of the current integer solution $\bar{y}$. This cut family is particularly helpful in the context of the asynchronous algorithm because it eliminates the current solution by (1) making the current node infeasible, (2) generating another integer solution, or (3) leading to the generation of a new fractional solution. In the first case, the node can be pruned by the infeasibility rule. This does not affect the convergence because no other feasible solution can be extracted from that node. The second case is in fact a desirable situation as it may yield a better incumbent value. This case, however, indicates that we need to add the combinatorial cuts within the inner while-loop at line 17. In the third case, the node will be added to the pool of active nodes, which does not affect the convergence properties. As a result, we can practically set $\gamma = 0$, i.e., the master processor does not need to wait for any cut and maintains the convergence.

4.2. Communications

In our parallel algorithm, each processor continuously executes its assigned work units (e.g., branching on the master variables or solving a SP) and shares the generated information. The information sharing must take place without any waiting in order to minimize the idle times. To do so, we make use of asynchronous message passing communications where each processor has its own buffer, i.e., local memory. This allows each processor to write its shareable information onto a buffer without waiting for the other processor(s) (i.e., the receiver) to actually receive the message. Thus, after sending the message, the processor returns immediately and continues executing the next work unit. Similarly, to avoid waiting in receiving information, each processor checks its buffer to see if there is any new message recorded by another processor. If there is new information in the buffer, the processor reads and processes it (which also cleans the buffer) and continues with the next step. If there is no information to read from the buffer, the processor continues with executing the next work unit without waiting.

At some point, however, a processor may need to wait for new information before being able to execute the next step. This waiting for a slave processor can only happen when there are no more SPs to evaluate. For the master processor waiting is controlled by the γ parameter, which we will further elaborate in Section 4.5.2.

4.3. Work Allocation Strategies

Two main strategies can be used in assigning SPs (i.e., work units) to slave processors. The first strategy revolves around dynamically assigning the SPs to slave processors, i.e., give the next SP to the next available processor. In the second strategy, each SP is assigned to a specific slave processor decided a priori.

Broadly speaking, static work allocation is preferred over the dynamic work allocation because the SPs obtained from SIP models usually have the same level of difficulty, i.e., this assures the workload is balanced. Moreover, static work allocation allows for a more effective use of the re-optimization tools of the black box solvers. In addition, when one uses dynamic work allocation within the asynchronous algorithm, one needs to have an additional processor to merely coordinate the slave processors, while it could be used to do more useful works. We thus adopt the static work allocation. We assign an almost equal number of SPs to each processor in order to avoid the additional overheads and complexities associated with the load balancing. To further improve the re-optimization capabilities, we also try to assign similar SPs to the same processor. The similarity of two SPs is measured as the Euclidean distance among the random parameters. Note that, grouping and assigning SPs to the slave processors is done by the master processor once at the beginning of the algorithm.

4.4. Pool Management Strategies

In our asynchronous algorithm, each processor continuously generates and shares information. The master processor keeps generating first-stage solutions and the slave processors keep evaluating the SPs to obtain cuts and bounds. Therefore, when the master processor is reading the buffer there might be many cuts in the memory. These cuts can be associated to different solutions and different SPs. Likewise, the master processor may share new solutions with the slave processors before they finish evaluating the SPs associated with the current solution. Therefore, each slave processor must have a pool to keep the generated master solutions, called *solution pool*, and the master processor must have a pool that contains the generated cuts, called *cut pool*. Properly managing these pools is of crucial importance in performance of the asynchronous algorithm.

4.4.1. Solution Pool Management

Each slave processor has a pool containing one or several master solutions. All the SPs assigned to the slave processor must be solved for every solution in this pool. The order of selecting these solutions is important. For instance, the cuts associated with a more recent solution might be more effective in improving the lower bound, while the objective value obtained from an older solution might be even more important in updating the incumbent value. Therefore, the slave processor needs to decide which solution to choose at the current step and then decide to evaluate which one of its associated unevaluated SPs. The latter is

a scheduling decision which we address in Section 4.5.1. We suggest the following strategies to select solutions from the solution pool:

S1: choose solutions randomly;

S1: choose solutions based on the *first-in-first-out* (FIFO) rule;

S3: choose solutions based on the *last-in-first-out* (LIFO) rule.

In all these strategies, we always give a lower priority to infeasible solutions. This means that if a SP associated to a solution is infeasible (i.e., the dual is unbounded) that solution will not be selected unless there is no other solution in the pool. This is because an infeasible solution does not yield an incumbent value and it has little impact on the lower bound. Last but not least, we use the *cut improvement* notion to identify the solutions which are no longer required to be evaluated, see Rei et al. (2009) for more information.

4.4.2. Cut Pool Management

The slave processors may generate many cuts, which are not all worth adding to or keeping in the master formulation. We thus define some strategies to effectively manage the cuts generated by the slave processors. In this regard, the master processor reads and stores all the cuts from the buffer. Then, it selects an appropriate subset of them to be added to the MP, removes some useless cuts from the master formulation, and cleans up the cut pool. Note that removing cuts is a safe procedure, i.e., it does not affect convergence, since they can be regenerated if needed.

A cut from the cut pool is added to the master formulation if its relative violation (the absolute violation of the cut at the current solution divided by the 2-norm of the cut coefficients) is at least δ, otherwise it is discarded. Moreover, if there are more than one cut for a recourse variable $\theta_s,\ s \in \mathcal{S}$, in the pool, we only add the one that violates the current solution the most and keep the rest in the pool.

Cut removal can computationally be expensive as it changes the simplex basis and slows re-optimization. Thus, we execute the following routine after termination of the first phase, i.e., line 2 of Algorithm 1, to identify the dominated cuts. At iteration t, the hyperplane $(h_s - T_s y)^\top \alpha_s^t$ is generated to primarily bound the recourse problem s at solution $\bar{y}^t$. Following a heuristic notion, if there is another generated cut $(h_s - T_s y)^\top \alpha_s^j$, $j \neq t$, that bounds the recourse variable θ_s tighter at $\bar{y}^t$, i.e., $h_s^\top(\alpha_s^j - \alpha_s^t) + (\alpha_s^t - \alpha_s^j)^\top T_s \bar{y}^t \geq 0$, it flags the possibility of removing cut $(h_s - T_s y)^\top \alpha_s^t$ without deteriorating the approximated value function for SP $s \in \mathcal{S}$. We apply the same procedure for the feasibility cuts. Note that executing this test for all $y \in Y$ would yield an exact method to identify the dominated cuts (Pfeiffer et al., 2012). In the branching phase, i.e., lines 4 to 25, our algorithm shall remove any cut that might become slack.

4.5. Scheduling Strategies

In this section, we address two important questions which largely affect our asynchronous algorithm: 1) when to solve the MP, 2) which SP to solve next.

4.5.1. Sequencing the SPs

Once the slave processor choses the next master solution to evaluate, it also needs to decide which one of the associated SPs to evaluate next. Notice that the sequence of choosing a master solution and an associated SP is repeated at each step, as after solving a SP, new information can be obtained and, thus, the algorithm can make better decisions in selecting the next master solution. In this regard, we suggest the following strategies:

SP1: Randomly choose one of the unevaluated SPs associated with the chosen master solution;

SP2: We may not need to solve all SPs associated to an infeasible master solution. Thus, in this strategy, we assign a *criticality* counter to each SP, which increases by one each time the SP becomes infeasible. Then, a SP with the highest criticality value is selected.

SP3: This strategy is a hybrid of SP1 and SP2, where SPs with higher criticality values have a higher chance to be selected.

4.5.2. Scheduling the MP

A key feature of the asynchronous algorithm lies in being independent of the slave processors to return their evaluation before it executes the next step. Recall that we control the waiting portion of the master processor with the γ parameter. However, the cuts that the master processor reads from the buffer can be associated with the current and/or previous solutions. Therefore, we need to be more specific when saying "the master processor waits only for γ% of the cuts". We thus highlight the following three strategies:

MP1: The master processor waits until $\gamma|\mathcal{S}|$ cuts (associated with any master solution) are added to the master formulation;

MP2: The master processor waits until $\gamma|\mathcal{S}|$ cuts associated with the current master solution are added to the master formulation;

MP3: Master processor waits until $\gamma|\mathcal{S}|$ cuts (associated with any master solution) that are violated at the current solution are added to the master formulation or no more unevaluated solution exists.

We will examine these three strategies with different values of γ, which needs to be computationally tuned.

4.6. Overall Algorithm

We present the overall framework of the proposed asynchronous parallel BD method. Similar to Algorithm 1, the proposed method starts from the LP phase for which its pseudo code is presented in Algorithm 2.

Algorithm 2 first initiates the LP MP, groups the SPs, and assigns each cluster of SPs to a slave processor. The master processor iteratively solves the MP and adds cuts until

Algorithm 2 : LP phase of the asynchronous parallel Benders method

Create the MP which is the LP relaxation of program (6)-(8) with $E_s = \emptyset$ and $F_s = \emptyset$, $\forall s \in \mathcal{S}$
Group scenarios and assign each cluster to a slave processor (see Section 4.3)
$LB = -\infty$, $UB = \infty$, and $t = 0$
while *no stopping condition is met* **do**
    Solve the MP to get an optimal solution $\bar{y}$
    Broadcast the $\bar{y}$ solution and set $t \leftarrow t + 1$
    Wait until the appropriate cuts are read from the buffer (see Section 4.5.2)
    Chose and add cuts to the MP (see Section 4.4.2)
    Update the UB (if possible) and set LB equal to the current MP's objective value.
Clean the dominated cuts (see Section 4.4.2).

a stopping condition is met. After solving the MP, the master solution is broadcast to all other processors provided that the solution is not the same as the previous iteration (note that the MP can generate the same solution if the applied cut subset is not violated by the current solution). After broadcasting the current solution, the master processor returns immediately and checks for the information shared by the slave processors. After retrieving the appropriate information, it adds the appropriate cuts to the master formulation, updates the bounds, and repeats the same procedure.

Every slave processor executes the same process but independently (i.e., there is no communication among the slave processors). The pseudo code of this process is illustrated in Algorithm 3.

Algorithm 3 The process executed on each slave processor

Receive the set of assigned SPs; Set $\mathcal{P} = \emptyset$.
Create the dual SP formulation(s)
while *no termination signal is received* **do**
    Retrieve each available $\bar{y}$ solution in the buffer and set $\mathcal{P} = \mathcal{P} \cup \bar{y}$, if any
    if $\mathcal{P} = \emptyset$ **then**
        Wait until a new master solution is broadcast
    Select a master solution $\bar{y}$ from the pool $\mathcal{P}$
    Select an unevaluated SP associated with $\bar{y}$
    Update and solve the dual formulation for this SP
    Send a message to the master processor with the cut and bound information
    if all SPs associated with $\bar{y}$ are evaluated **then**
        Remove $\bar{y}$ from the pool, i.e., $\mathcal{P} = \mathcal{P} \backslash \bar{y}$

In Algorithm 3, each slave processor receives its unique set of SPs from the master processor. At each step, the slave processor checks if a new master solution is broadcast. If so, it retrieves and stores the solution in a pool. If the solution pool is empty, it waits until a new one is broadcast by the master processor. Next, it chooses a solution from the

pool based on the appropriate *solution management* strategy. Then, it selects (based on the appropriate *solving SPs* strategy) one of the unevaluated SPs associated with the chosen solution and evaluates it. Finally, the processor sends a non-blocking message to the master processor containing the generated information (i.e., cut and bound).

In the second phase, the slave processors keep executing the same procedure, but the master processor starts branching on the master variables. The pseudo code of the overall algorithm is presented in Algorithm 4.

Algorithm 4 : The asynchronous parallel Benders method

1: Let the obtained MP from Algorithm 2 be the root node of tree $\mathcal{L}$, set $UB = \infty$, and let ϵ_{opt} to be the optimality tolerance.
2: **while** *no stopping condition is met* **do**
3: Read all the messages in the buffer
4: Add the appropriate cuts to the formulation, if any
5: Update the UB, if possible
6: Select node l from $\mathcal{L}$
7: Solve node l to get an optimal solution $\bar{y}$ with objective value of ϑ^*
8: **if** *node l is infeasible* **or** $\vartheta^* \geq UB$ **then**
9: Prune the node (i.e., $\mathcal{L} = \mathcal{L} \backslash \updownarrow$) and go to line 2
10: **if** *$\bar{y}$ is integer* **then**
11: Broadcast $\bar{y}$
12: Wait for cuts according to the selected scheduling strategy (see Section 4.5.2)
13: Add the new cuts, if any
14: **if** *no cut violates $\bar{y}$* **and** *not all cuts associated with $\bar{y}$ are generated* **then**
15: Add a combinatorial cut
16: Solve this node again and get the objective value ϑ^*
17: **if** *node was infeasible* **or** $\vartheta^* \geq UB$ **then**
18: Prune the node and go to line 2
19: **if** *the current node solution is the same as $\bar{y}$* **then**
20: **if** *all cuts associated with $\bar{y}$ are added to the MP* **then**
21: Prune node l, and go to line 2
22: **else if** *current node solution is integer* **then**
23: Set $\bar{y}$ equal to the current node solution and go to line 11
24: Remove node l from $\mathcal{L}$ and choose a fractional variable from $\bar{y}$ for branching
25: Create two nodes and add them to $\mathcal{L}$.
26: Broadcast the termination signal.

At each step, the master processor reads the new information from the buffer before choosing and evaluating an open node.

After selecting a node from the pool $\mathcal{L}$, if it cannot be pruned and its solution is integral, the solution will be sent to the slave processors. Then, the master processor may or may not stay idle for feedback from the slave processors. Unless all the cuts are added to the MP, it

will add a complementary combinatorial cut to the formulation to ensure convergence, lines 14 and 15. If upon adding the new cuts, the master solution changes but remains integral, we repeat the same procedure, line 22 and 23. In any case if the node solution is fractional, branching on the fractional variables occurs and the resulting child nodes will be added to the set of active nodes $\mathcal{L}$, lines 24 and 25. Finally, if one of the stopping criteria is satisfied, the master processor broadcasts the termination signal and exits.

5. Acceleration Strategies

The performance of the BD method vastly depends on the feedback it receives from the SPs at each iteration. The asynchronous parallel BD method may entail a larger number of iterations and a considerable amount of redundant work as it executes the next iteration with *partial feedback* on its current solution. As a result, the lower bound may progress slowly since only a subset of the cuts are applied to the MP at each iteration. Also, the asynchronism may increase the computational cost of each iteration since the MP grows large at a faster pace.

The proof of convergence may also be delayed due to the unavailability of the function value of the master solutions for all SPs. To overcome these drawbacks, we propose a number of acceleration strategies. Note, one may use many of the classical acceleration techniques to improve the performance of the asynchronous algorithm. We discuss such techniques in Section 6.2. In this Section, we limit ourselves to more generic and novel ones which are specific to the (asynchronous) parallel implementation of the algorithm.

5.1. Cut Aggregation

When the number of SPs is larger than the number of the first-stage variables, it is not numerically the best strategy to add to the MP a cut per SP (Trukhanov et al., 2010). Thus, we group the SPs into $|\mathcal{D}|$ clusters using the *k-mean++* algorithm (Arthur and Vassilvitskii, 2007), where $\mathcal{D}$ is the set of clusters with cardinality $|\mathcal{D}|$. Note that number of the SPs in each cluster may not be equal. In the synchronized and sequential BD algorithms, one can define a single recourse variable for each cluster and aggregate the generated cuts in that cluster into a single cut. However, this procedure is not applicable to the asynchronous algorithm, since it may only solve a subset of the SPs in each cluster.

To alleviate this issue, we define a recourse variable for each SP. Then, we add a cut of the form: $\sum_{s\in\hat{\mathcal{S}}_d}\rho_s\theta_s \geq \sum_{s\in\hat{\mathcal{S}}_d}\rho_s\big(h_s - T_s y\big)^\top\alpha_s$ for cluster $d \in \mathcal{D}$, where $\hat{\mathcal{S}}_d$ indicates the set of evaluated SPs in this cluster at the current iteration. Note that, we can update this inequality in the following iterations when the remaining SPs in cluster d are evaluated. However, this process requires modifying the MP, which is not computationally efficient. We thus aggregate the cuts for the current solution separately from those associated with the previous iterations.

5.2. Creating Artificial SPs

The master processor does not wait for all the SPs associated with the current solution to be solved. Thus, having good cuts that can represent the unevaluated SPs is important

to improve the efficiency of the asynchronous algorithm. To this end, we extend the idea presented by Crainic et al. (2016) in order to create artificial scenarios. The SP associated with each artificial scenario will then be solved to bound the recourse variables for a set of SPs. Our developments require the following assumption.

Assumption 1. *We assume that the problem has a fixed recourse property and the objective coefficients are deterministic, i.e.,* $W_s = W$ *and* $c_s = c$, $\forall s \in \mathcal{S}$.

We cluster the SPs into $|\mathcal{G}|$ groups according to the similarity measure (Crainic et al., 2016). Note that the cardinality of clusters may not be equal. Then, to generate an artificial SP for cluster $g \in \mathcal{G}$, we set $h_g = \sum_{s\in\mathcal{S}_g} \beta_s h_s$ and $T_g = \sum_{s\in\mathcal{S}_g} \beta_s T_s$, where $\mathcal{S}_g$ is the set of scenarios in cluster $g \in \mathcal{G}$ with $\mathcal{S} = \cup_{g\in\mathcal{G}}\mathcal{S}_g$, and β_s is the weight associated with scenario $s \in \mathcal{S}_g$ such that $\sum_{s\in\mathcal{S}_g} \beta_s = 1$.

Proposition 1. *Any extreme point* α^g *and extreme ray* r^g *of the artificial subproblem* $g \in \mathcal{G}$ *gives a valid optimality cut* $\sum_{s\in\mathcal{S}_g} \beta_s\theta_s \geq (h_g - T_g y)^\top \alpha^g$ *or a feasibility cut* $0 \geq (h_g - T_g y)^\top r^g$.

Proof 1. *Proof. See Appendix A.*

An important issue in deriving the artificial cuts lies in setting the β weights. The following theorem suggests an optimal way to obtain these weights.

Theorem 1. *The maximum bound improvement by the artificial scenario* $g \in \mathcal{G}$ *is attained when the convex combination weight* β_s *for scenario* $s \in \mathcal{S}_g$ *is* $\dfrac{\rho_s}{\sum_{s\in\mathcal{S}_g} \rho_s}$.

Proof 2. *Proof. See Appendix C.*

Note that we make use of the artificial scenarios to bound the recourse variable of those SPs which remain unevaluated at the present iteration. The following corollary suggests a strategy to further tighten the associated cuts.

Corollary 1. *Let* $\bar{\mathcal{S}}_g$ *be the set of evaluated SPs in cluster* $g \in \mathcal{G}$ *at the current iteration. Then solving the artificial SP using a smaller set* $\mathcal{S}_g \backslash \bar{\mathcal{S}}_g$ *yields a tighter cut for the remaining SPs. This follows from the aggregation step in the proof of Proposition 1.*

From Corollary 1, we observe that there is no need to apply the scenario creation strategy within the synchronized or sequential algorithms.

5.3. Cut Propagation

We suggest the use of bunching techniques (Birge and Louveaux, 1997) to propagate the generated cuts in order to derive additional approximate cuts for the set of unevaluated SPs. Assume that Assumption 1 holds. Thus, given an optimality cut $\theta_s \geq (h_s - T_s y)^\top \alpha_s$ associated with the SP $s \in \mathcal{S}$, we can generate a valid optimality cut $\theta_{s'} \geq (h_{s'} - T_{s'} y)^\top \alpha_s$ for SP $s' \in \mathcal{S} : s' \neq s$ without solving SP s'. This simply follows from the fact that under the considered assumptions, the dual polytope is identical for all SPs.

It is not computationally viable to propagate any dual solution due to time consuming calculations and handling requirements. We thus generate propagated cuts only for those recourse variables whose associated SP remains unevaluated at the current iteration. Also, if a dual value yields a cut which is not violated by the current solution, it will not be used in the propagation procedure. Finally, for SP s' we generate a propagated cut using the solution from SP s if the latter has the greatest *dominance* value over the former in the set of evaluated SPs at the current iteration. Note that the dominance value of SP s over SP s' is calculated as the total number of the elements $j \in \{1, ..., l\}$ for which $h_s^j \geq h_{s'}^j$ and ${T_s^j}^\top y \geq {T_{s'}^j}^\top y$ for every $y \in Y$ (Crainic et al., 2016).

5.4. Upper Bound from Fractional Points

The upper bounds obtained from fractional master solutions (e.g., line 2 of Algorithm 1) are not valid for the original problem. Due to the high importance of the upper bound value in pruning nodes of the branch-and-bound tree, we suggest a practical strategy to extract valid incumbent values from fractional solutions. This strategy requires the following assumption.

Assumption 2. *If $f_i \geq 0$ and for a given feasible $\bar{y}$, $\hat{y}$ is also feasible if $\hat{y}_i \geq \bar{y}_i$ for all $i \in \{1, ..., n\}$, then $Q(\bar{y}, s) \geq Q(\hat{y}, s), s \in \mathcal{S}$.*

Proposition 2. *Given a feasible fractional master solution $\bar{y}$, i.e., $\bar{y} \in Y \bigcap_{s\in\mathcal{S}}\{W_s x \geq h_s - T_s\bar{y}$, for some $x \in \mathbb{R}_+^m\}$ and the associated recourse costs ν_s^* for every $s \in \mathcal{S}$, under assumption 2 a global upper bound can be obtained from $f^\top\lceil\bar{y}\rceil + \sum_{s\in\mathcal{S}} \rho_s\nu_s^*$, where $\lceil\rceil$ is a roundup function.*

Proof 3. *Proof. See Appendix B.*

Note that if Assumption 2 does not hold for a given problem, the overall algorithm still converges to an optimal solution without the results of Proposition 2.

5.5. Complementary Cut Generation

Conventionally, generating cuts at fractional nodes of the B&BC algorithms, except for the root node, is ignored. This is, to a very large extent, due to the excessive time required to generate and handle them (Botton et al., 2013). However, we have more computational power at our disposal in a parallel B&BC method. Thus, we suggest strategies to generate and handle cuts at fractional nodes of the branch-and-bound tree. We refer to them as *cut generation* strategies.

We propose to generate cuts for the first Γ fractional nodes, where $0 < \Gamma \leq \infty$ and 0 being the root node. There are two possibilities at each fractional node: (i) generate at most one round of cuts per node or (ii) call upon the cut generation module as long as the local bound at the current node improves by more than $\tau\%$.

Another cut generation idea is to round a fractional solution in order to derive an integer one, which can be used to obtain valid upper bounds and cuts. The rounding can be done in many different ways. We suggest the following techniques:

i- Round the fractional values to the closest integer point;

ii- Round upward/downward any value that its fractional part is greater/smaller than λ, where $\lambda \in (0, 0.5)$;

iii- Solve a restricted MILP program to round the fractional values, see Appendix D for more information.

5.6. Hybrid Parallelization

We observed that when solving the MP and SPs is quick, re-optimizing the MP with partial feedback from the slave processors can yield efficiency drawbacks. This is particularly the case during the first phase of our asynchronous parallel algorithm. For this reason, we propose to use the synchronized algorithm to solve the LP relaxation of the problem (i.e., set $\gamma = 1$ in line 7 of Algorithm 2) and explore the branch-and-bound tree with the asynchronous algorithm. We refer to this strategy as *hybrid parallelism*.

6. Implementation Details

We solve all LP and MILP problems using IBM ILOG CPLEX 12.7.1. All programs are coded in C++ environment. The code is compiled with g++ 4.8.1 performed on Intel Xeon E7-8837 CPUs running at 2.67GHz with 64GB memory under a Linux operating system. We used the *open mpi* 1.8.6 to manage the communications. The B&C algorithm was also implemented using CPLEX's legacy callable libraries. We solve the extensive formulation with CPLEX's default setting, and turn off presolve features for our Benders-based algorithms.

6.1. Test Instances

To test our method, we address the well-known *Multi-Commodity Capacitated Fixed-charge Network Design Problem* with *Stochastic Demand* (MCFNDSD). This problem naturally appears in many applications (Klibi et al., 2010) and it is notoriously hard to solve (Costa, 2005; Crainic et al., 2011). The complete details of this problem are given in Appendix E. To conduct the numerical tests, we have used the **R** instances which are widely used in the literature, e.g., Chouman et al. (2017); Rahmaniani et al. (2017b); Crainic et al. (2016, 2011); Boland et al. (2016). These instances have up to 64 scenarios. To generate a larger number of scenarios (i.e., 1000 scenarios), we have followed a procedure similar to the one used by Boland et al. (2016). For the numerical assessment of the strategies, we have considered a subset of the instances: r04-r10 with correlation of 0.2 and cost/capacity ratio of 1, 3, and 9 which yields for 21 instances. This subset of the **R** family corresponds to the instances most commonly tackled in the literature (Rahmaniani et al., 2017b; Crainic et al., 2016). In the following part of the computational experiments, where we study the performance of our method versus alternative methods, we have considered a larger number of instances, i.e., r04-r11 accounting for 200 instances where each instance has 1000 scenarios. The description of these instances is given in Appendix F.

6.2. Classical Acceleration Strategies

To further accelerate the presented B&BC algorithms, we also incorporated some of the best known classical acceleration strategies. We have thus implemented the following techniques in all of our algorithms: (i) warm start strategy, (ii) valid inequalities for the MP, (iii) valid inequalities for each SP, and (iv) Pareto-optimal cuts. The complete details of these strategies can be found in Rahmaniani et al. (2017b). Finally, as local branching was shown to effectively accelerate the BD method (Rei et al., 2009), we turn on the local branching of CPLEX in the second phase of our B&BC algorithms.

6.3. Implementation of the Asynchronous Algorithm

We discussed various search strategies whose combination gives a very large number of algorithms to test. Presenting the numerical results for all of them is clearly beyond the scope of this article. For this reason, we provide some insights for those strategies which we do not present numerical results for.

With respect to *solution management* strategies of Section 4.4.1, we observed that LIFO outperforms both the FIFO and random strategies. The main reason is that the Benders method, as a "dual algorithm", is very sensitive to the feedback on its current solution. In both the FIFO and random selection strategies, "older" solutions are usually selected. As a result, the generated cuts are dominated or they have a very limited impact on the current MP. Thus, the MP generates a solution which is not very different (in terms of quality) from the previous iteration(s) and the lower bound progresses very slowly. Furthermore, in both strategies the upper bound improves at a slower pace compared to LIFO, although they might update the upper bound more frequently at the initial iterations. This is because the FIFO and random selection strategies need a much larger number of iterations to actually find a high quality feasible solution that gives a tight bound. As a result, we only consider the LIFO as the solution selection strategy in the following.

With respect to *solving SPs*, we realized that the random selection of the SPs yields a better performance. This is because the random selection of the SPs ensures diversity of the cuts applied to the MP. In the ordering based strategy, i.e., SP2, the order of solving the SPs reaches a stationary state after few iterations. Thus, the algorithm rarely bounds the recourse variables of the SPs with the least priority.Hence, the lower bound progresses slowly. Notice that each slave processor selects a master solution based on the LIFO strategy and not every SP associated with that solution might be solved before the master processor broadcasts a new solution. Finally, the random selection based on the criticality weights tends to perform better than a pure random strategy. This is because higher priority is given to the indicator SPs and the diversity in the cuts applied to the MP is maintained. Therefore, we consider this strategy, i.e., SP3, when selecting SPs for evaluation.

Finally, we consider the proposed strategies in Section 4.5.2. In this regard, we make use of MP2, i.e., waiting until $\gamma|S|$ cuts associated with the current solution are added to the MP. This is because we can use the combinatorial cuts that are violated by any integer master solution. We thus avoid the additional waiting times as required in the MP3 strategy. Moreover, we observed that applying cuts associated with the current solution is

important, due to the same reasons that justified the FIFO strategy. The MP1 strategy does not guarantee that any of such cuts will be added to the MP.

6.4. Stopping Criteria and Search Parameters

In solving each stochastic instance, we have set the stopping optimality gap at 1%. The total time limit is set at 2 hours. To solve the LP relaxation of the problem at the root node, we have considered half of the maximum running time limit. The parallel variants are run on 5 processors unless otherwise specified. One of the processors solves the MP and the rest are assigned to solving the SPs. In the cut generation strategy, τ and λ, δ values are set to 0.5%, 10^{-1}, and 10^{-3}, respectively.

7. Computational Results

We present the numerical assessments of the proposed parallelization strategies in two steps. We first study different versions of our parallel B&BC algorithms to evaluate the limitations and impact of the proposed acceleration techniques. The second part of the analysis is devoted to test the speedup and scalability of our parallel algorithms. Finally, we conduct a comparison between our exact algorithms and CPLEX to benchmark their performance.

7.1. Synchronized Parallel Algorithm

We investigated the impact of the proposed acceleration strategies, namely the cut management, cut aggregation and cut generation in the context of the synchronous parallel B&BC algorithm. Note that we activated the proposed strategies one by one to observe their cumulative impact over the performance. Thus, the basic algorithm in each section is the best variant obtained from the previous subsection.

7.1.1. Cut Management

We first study the impact of the cut management strategy. We compare the method proposed in Section 4.4.2 to that commonly used in the literature, where cuts with high slack value are removed, see, e.g., Pacqueau et al. (2012). In Table 2, the average running time in seconds, optimality gap in percentages, and number of removed cuts, shown with #Cut, are reported for each cut management strategy.

Table 2: Numerical results for various cut management strategies

	NoCutManagement		**SlackCutManagement**			**DominanceCutManagement**		
	Time(ss)	Gap(%)	Time(ss)	Gap(%)	#Cut	Time(ss)	Gap(%)	#Cut
r04	2422.60	1.00	2424.57	1.26	151.00	1802.97	0.36	1616.00
r05	1133.64	0.47	517.22	0.47	164.33	500.64	0.47	942.67
r06	3270.45	1.76	2656.59	2.06	602.33	2601.43	2.12	22.00
r07	2437.16	3.32	2439.03	2.37	13.67	2141.70	2.06	1599.33
r08	2534.89	3.36	2513.26	3.56	50.67	2315.18	2.40	292.00
r09	4898.53	4.01	4737.94	4.24	163.67	4780.70	4.25	868.67
r10	4977.60	7.77	4927.27	5.25	645.00	4931.94	6.23	108.00
Ave.	**3096.41**	**3.10**	**2887.98**	**2.75**	**255.81**	**2724.94**	**2.56**	**778.38**

It is important to note that none of the methods deteriorates the LP bound at the root node. We observe from Table 2 that cleaning up the useless cuts yields a positive impact on the performance. The slack-based strategy keeps many of the useless cuts in the MP while the proposed method removes a much larger number of them. Thus, for small and medium instances, we observe a clear advantage of the proposed method. However, for larger instances, the impact of removing cuts on the run time is less significant because the algorithm reaches the time limit. We here note that most of the removed cuts are associated with early iterations. Finally, we observe that there is no direct relation between the number of the cuts removed and the performance of the B&BC method. This is probably due to the heuristic nature of both strategies.

7.1.2. Cut Aggregation

We ran the algorithm with 11 different cluster sizes in order to study the impact of various cut aggregation levels on the performance. The comparative results in terms of total running time are depicted in Figure 2. Note that the value on each column gives the average optimality gap in percentage.

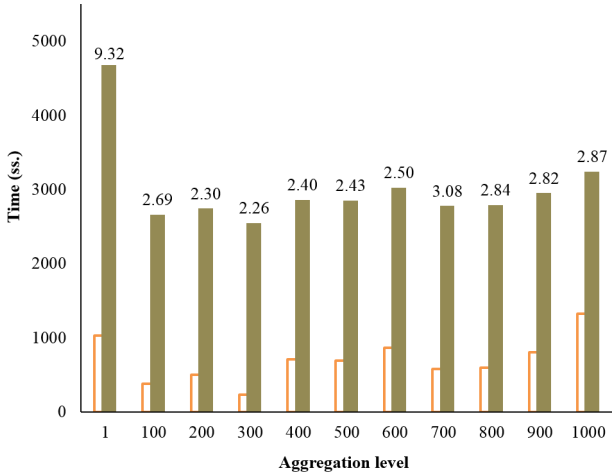

Figure 2: Impact of various cluster sizes on the synchronized parallel algorithm (the white bars show the average run time for the solved instances)

We observe from Figure 2 that neither of the two conventional strategies, i.e., single cut (column labeled "1") and multi-cut (column labeled "1000"), gives the best results, although the latter performs noticeably better than the former. The best aggregation level is associated with a cluster size of 300. All aggregation levels are able to solve 66.67% of the instances within the 2-hour time limit, except for the single cut method which could only solve 61.90% of the instances. In addition, we observe that the difference in running time among some aggregation levels is small. This is because the algorithm reaches the maximum run time limit for 33.33% of the instances. Thus, if we only consider the instances that are solved by all aggregation levels, we observe more significant differences in the running times. These results are presented by the white bars in Figure 2.

Comparing the results to those of the sequential algorithm, see Figure G.7 in Appendix G, we observe that a larger number of clusters gives the best results for the sequential algorithm. This is because the sequential algorithm performs less iterations in the same amount of time and thus, it is more sensitive to the loss of information in the aggregation step.

7.1.3. Cut Generation Strategy

We next study the cut generation strategies of Section 5.5. For computational purposes, we have set Γ to 10, 100 and ∞ and compare the results to the case for which we generate cuts merely at the root node, i.e., $\Gamma = 0$. For each Γ value, we have reported numerical results for two cut generation strategies:

- "Single" indicates generating cuts once per node ;
- "Multi" indicates generating cuts for each node as long as the lower bound improves by more than 0.5%.

Also, we have studied two cases for each strategy: (i) the fractional solutions are directly used and (ii) the feasibility of the infeasible ones is restored. The numerical results are summarized in Table 3.

Table 3: Numerical results for various cut generation strategies

Γ	Strategy	NoFeasibilityRestoration			FeasibilityRestoration		
		Time(ss)	Gap(%)	Sol.(%)	Time(ss)	Gap(%)	Sol.(%)
∞	Single	2737.66	20.19	57.14	2694.15	10.95	66.67
∞	Multi	3045.31	29.38	57.14	3164.12	24.80	61.90
100	Single	2538.98	16.27	52.38	2611.14	10.58	66.67
100	Multi	2999.89	25.00	57.14	3050.19	24.84	66.67
10	Single	**2398.36**	**2.16**	**71.43**	2405.82	2.41	71.43
10	Multi	2612.17	2.48	71.43	2583.14	2.40	71.43
0	-	2549.10	2.26	66.67	-	-	-

Note that, throughout this paper "Sol.(%)" refers to the percentage of the solved instances. From Table 3 we observe that generating cuts for many fractional nodes considerably increases the optimality gap. This happens because of two main reasons. First, generating cuts for many nodes is computationally expensive and second, it causes considerable handling costs. For the same reasons, the "Multi" strategy underperforms when compared to the one with the "Single" module. On the contrary, generating cuts merely for the first 10 nodes performs better or equal to generating cuts only at the root node, such that the average running time and optimality gap are reduced by 150.74 seconds and 0.1%, while the percentage of solved instances increases by 4.76%.

We next examine the strategies proposed to round the fractional solutions. The first strategy, denoted *Round*, rounds each fractional value to its closest integer value. The second strategy, denoted *Upward*, rounds each fractional value greater and equal than 0.1 to 1 and 0 otherwise. The third strategy employs a restricted MIP to find a close integer

Table 4: Numerical results for the rounding based cut generation strategies

Strategy	NoFeasibilityRestoration			FeasibilityRestoration		
	Time(ss)	Gap(%)	Sol.(%)	Time(ss)	Gap(%)	Sol.(%)
MIP	-	-	-	**2486.78**	**2.31**	**71.43**
Round	2629.98	2.20	61.90	2591.03	2.25	61.90
Upward	2579.79	2.35	66.67	2606.85	2.44	71.43
$\Gamma = 0$	2549.10	2.26	66.67	-	-	-

feasible solution to the current fractional solution. The numerical results are summarized in Table 4.

Note that, following the observations in Table 3, we have applied these strategies only at the first 10 fractional nodes. Comparing Tables 3 and 4, we observe that the rounding strategies do not perform better than directly using the fractional solutions. The main reason is that the cuts associated with the rounded solutions do not, in general, affect the local bounds. In addition, CPLEX's default heuristics and local branching are active in our implementations. This makes the marginal impact of the rounding strategies negligible.

7.2. Asynchronous Parallel Algorithm

We study the proposed strategies for the asynchronous parallel algorithm. We first study different synchronization levels as controlled by the γ parameter. Then, we analyze the proposed acceleration techniques.

7.2.1. The Synchronization Level

Figure 3 depicts the impact of the γ value on the running time. The values on this figure represent the average optimality gaps in percentage.

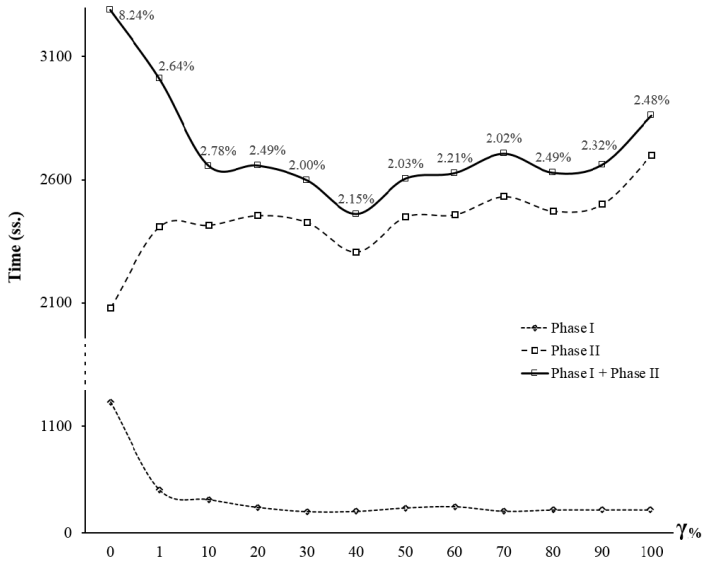

Figure 3: Effect of waiting portion of the master processor (gamma value) on the performance

We observe that the best performance lies neither at 100% nor at 1%. It is always numerically better to wait for some percentage in between. Moreover, comparing Figures 3 and G.7, we observe that full asynchronism (i.e., $\gamma = 0$) underperforms even when compared to the sequential algorithm. The best result is attained when the master processor waits until 40% of the SPs associated with the current solution are solved. This synchronization level solves 66.67% of the instances with an average optimality gap of 2.15% in 2465.26 seconds. This is comparable to the synchronous algorithm without the cut generation strategies, see Figure 2.

We observe that waiting for a larger portion of the SPs is more desirable when solving the LP relaxation of the MP, while the contrary is true in the second phase of the algorithm. In the former case, we gain nothing from quickly resolving the MP since no other useful task can be executed. In the latter case, however, we can perform useful work (i.e., evaluating open nodes of the search tree) while the cuts are being generated. We thus consider $\gamma = 100\%$ and 0% in the first and second phase of our hybrid algorithm.

7.2.2. Scenario Creation

Through the figures of Table 5 we investigate the impact of the artificial SPs on the convergence of our asynchronous algorithm. Note, we have created one artificial SP per slave processor.

Table 5: Impact of the artificial subproblems on the performance of the asynchronous algorithm

	NoArtificialSP			NumberOfArtificialSPs=$\|\mathcal{P}\|$		
	Time(ss.)	Gap(%)	Sol.(%)	Time(ss.)	Gap(%)	Sol.(%)
r04	689.50	0.75	100.00	759.39	0.73	100.00
r05	274.84	0.51	100.00	286.64	0.42	100.00
r06	2518.45	1.86	66.67	2546.88	1.48	66.67
r07	1442.27	0.84	66.67	1412.31	0.90	66.67
r08	2521.60	2.65	66.67	2466.25	2.02	66.67
r09	4898.42	3.98	33.33	4859.83	3.46	66.67
r10	4911.77	5.07	33.33	4880.33	4.53	33.33
Ave.	2465.26	2.25	66.67	2458.80	1.93	71.43

We observe in Table 5 that the creation of artificial scenarios increases the percentage of the solved instances by 4.76% while the average run time remains almost unchanged. The running time with artificial SPs increases for small instances and for larger instances the time improvement is not very much noticeable. In small problems, additional time is spent on generating cuts while they could have been solved quicker. For larger instances, in many cases, the algorithm reaches the time limit and thus the improvement on the average time does not appear significant. We observe, however, that the average optimality gap has been reduced. Thus, the artificial SPs are a valid strategy to accelerate the asynchronous algorithm.

7.2.3. Cut Aggregation

Figure 4 reports the impact of the proposed cut aggregation scheme for different cluster sizes within the asynchronous algorithm. The value on each bar shows the average optimality gap in percentages.

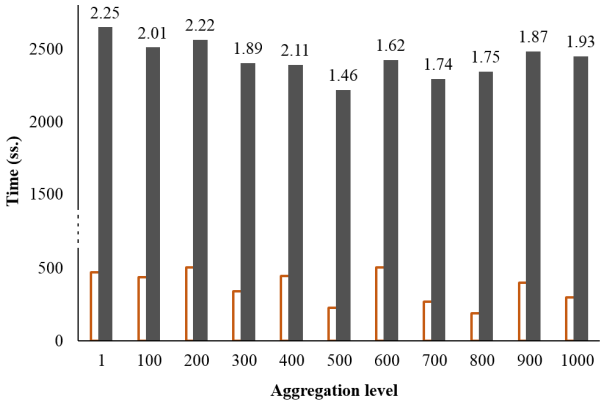

Figure 4: Impact of different cut aggregation levels on the asynchronous algorithm (the white bars show the average run time for the solved instances)

We observe that clustering improves the convergence of the algorithm. The best aggregation level is reached when number of the clusters is 500. Notice that this number is greater than the best value found for the synchronized algorithm, i.e., a cluster size of 300. With this aggregation level, our asynchronous algorithm solves 76.19% of the instances with an optimality gap of 1.46% in 2219.42 seconds and thus it outperforms the synchronized algorithm. Moreover, the single cut method does not perform much differently from other aggregation levels. This is a consequence of the introduction of a recourse variable for each SP when aggregating the cuts.

7.2.4. Cut Propagation

To assess the usefulness of the proposed cut propagation strategy, we compare our algorithm with and without the cut propagation of Section 5.3. Table 6 summarizes the results in terms of average run time and number of iterations. Note that the cut propagation is performed merely during the first phase of the algorithm. We thus report the time required and number of iterations to find the optimal LP solution.

Table 6: Impact of the cut propagation on the LP phase of the asynchronous algorithm

	WithoutPropagation		WithPropagation	
	Time(ss.)	#Iter.	Time(ss.)	#Iter.
r04	29.62	17.00	111.24	13.67
r05	50.88	21.33	213.86	14.00
r06	307.41	32.67	1377.90	18.67
r07	19.98	15.67	286.50	14.00
r08	59.61	21.00	415.22	15.33
r09	546.43	30.67	1085.35	16.00
r10	1158.31	41.67	1591.48	23.00
Ave.	310.32	25.71	725.94	16.38

We observe from Table 6 that cut propagation reduces the number of major iterations. However, it increases the overall run time requirement to optimize the LP relaxation of the problem. This is partially due to the additional time needed for handling the propagated

cuts. In addition, we can solve the LP relaxation of the considered instances within the considered time limit and, as expected, the cut propagation does not further tighten this bound. We thus believe that the proposed cut propagation strategy can be very useful for problems with a fairly small number of hard-to-solve SPs rather than many easy-to-solve SPs. Since this is not the case in our problem, we do not include this strategy in our method.

7.3. Speedup of the Parallel Algorithms

In Figure 5, we compare our parallel algorithms on 2, 3, 5, 10, 15 and 20 processors. In this experiment, we have considered only those instances that the sequential algorithm could solve within a 10-hour time limit in order to have a clear sense of the speedups. Note that the value on each bar indicates the speedup ratio calculated as the time of the best sequential algorithm divided by the time of the parallel algorithm.

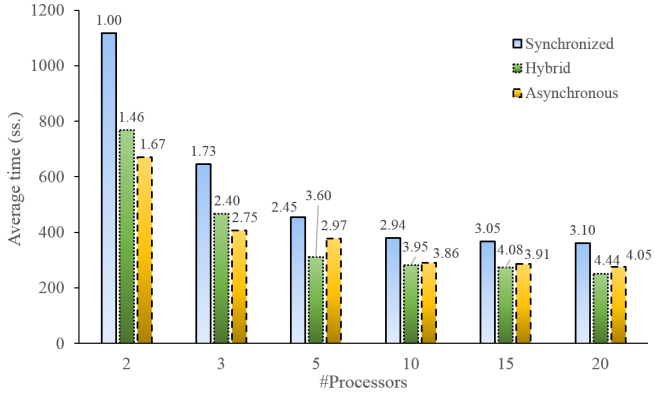

Figure 5: Speedup rates of our parallel Benders decomposition algorithms

The first interesting observation is related to the use of 2 processors, i.e., 1 slave and 1 master processor. In this case, the synchronized algorithm performs worse than the sequential algorithm because of the cut generation which slows down the algorithm. Our asynchronous algorithm, on the contrary, performs much better. This is clearly due to the better use of the processors in our asynchronous method since the two processors are less dependent. Also, we observe that the hybrid algorithm is better than the synchronized one, although they are roughly the same during the first phase of the algorithm. Thus, the speedup rate in our hybrid algorithm must be due to its asynchronous part. This indicates that our asynchronous method has noticeably reduced the computational bottleneck at the master level.

The speedups do not increase monotonically with the number of the processors. This is because increasing the number of slave processors does not alleviate the bottleneck at the master processor, although it significantly accelerates the cut generation cycle. We observe that the synchronous algorithm reaches super linear speedup during the LP phase of the algorithm, while it fails to reach even linear speedup overall. This is due to the second phase of the algorithm where the slave processors are not efficiently used and the heavy work is carried out by a single processor, i.e., master. The same situation also applies to the asynchronous and hybrid algorithms.

Our third observation concerns the hybrid method. It outperforms the synchronized method because of using non-blocking communications in the branching phase, which is the most time consuming part of the algorithm; see Figure 3. The advantage of the hybrid algorithm over the asynchronous method becomes more evident for larger instances in which solving the LP relaxation is noticeably time consuming. Moreover, we observe that its efficiency exceeds, or becomes closer to, the asynchronous method as the number of processors increases. This justifies the development of the hybrid algorithm.

7.4. Comparison with CPLEX

We conduct a comparison of our asynchronous and hybrid algorithms with CPLEX. In doing so, we ran our algorithms until reaching the same optimality gap which is obtained by CPLEX after 10 hours. Note that all algorithms are run on 15 processors. The average speedup rates are reported in Figure 6. These values are rounded to the closest integer point and they are obtained from dividing CPLEX's time to that of our methods.

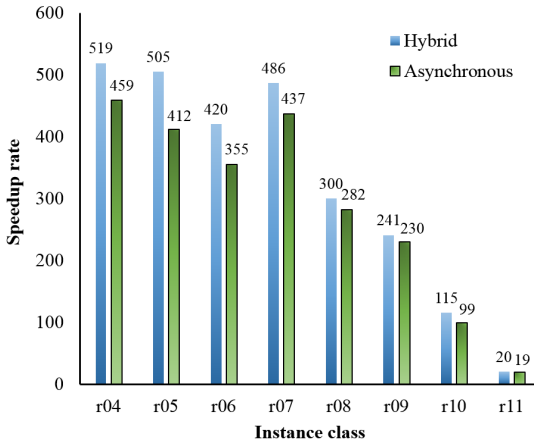

Figure 6: The speedup rate of our asynchronous and hybrid parallel algorithm compared to CPLEX

The asynchronous and hybrid parallel algorithms are, on average, 286.48 and 325.69 times faster than CPLEX in obtaining equal or better optimality gaps. We observe that the speedup for larger instances is smaller because solving their LP relaxation takes up to one hour and thus the speedup rates in scale of 10. Moreover, the hybrid parallelization reaches a better speedup rates since it solves the LP relaxation faster.

To complete the comparative study in this section, we compare the optimality gaps obtained by our parallel algorithms to that of CPLEX for a run time limit of 2 hours. Note that if CPLEX fails to find a feasible solution, we have set the optimality gap at 100%. The average optimality gap for each instance class is summarized in Table 7.

We observe from Table 7 that CPLEX fails to handle even small instances of the considered stochastic problem. Furthermore, the hybrid algorithm reaches better optimality gaps than the asynchronous method for larger instances.

Table 7: The average optimality gap obtained by each method after 2 hours

	CPLEX	**Asynchronous**	**Hybrid**
r04	14.49	0.33	0.34
r05	31.84	0.48	0.46
r06	56.11	2.00	2.75
r07	33.73	0.34	0.50
r08	43.90	2.18	2.48
r09	58.07	3.25	2.72
r10	56.15	3.58	2.28
r11	86.86	7.65	6.82
Ave.	**47.64**	**2.48**	**2.29**

8. Conclusions and Remarks

We studied parallelization strategies for the Benders decomposition method in which the subproblems are concurrently solved on different processors and the master problem on a single processor. We implemented the algorithm in a B&C framework and presented synchronous, asynchronous and hybrid parallelization frameworks along with various acceleration strategies. We designed the asynchronous algorithm such that it alleviates the interdependency of the master and subproblems.

Reporting numerical results on hard benchmark instances from stochastic network design problems with 1000 scenarios, we observed (super-)linear speedups when the master problem does not computationally dominate the algorithm. This is particularly true when solving the LP relaxation of the master problem. In similar cases, the synchronized method performed generally better than the asynchronous algorithm. The overall parallel algorithms did not reach a linear speedup. Moreover, we observed that our parallelizations did not scale with the number of the processors. The main reason for this is the fact that the most significant computational bottleneck of the parallel algorithm is solving the master problem sequentially on a single processor. All in all, the proposed asynchronous and hybrid algorithms displayed a better performance compared to the synchronous method. Compared to CPLEX, they were more than 286 times faster to obtain the same optimality gap. In addition, for the same time limit, they could also obtain optimality gaps that were more than 19 times lower than that of CPLEX.

This research opens the way for a number of interesting issues to be considered in future works. First and foremost, the master problem in our algorithm needs to be solved in parallel in order to reach a scalable algorithm. Second, it is worthwhile to study the proposed cut propagation strategy for the problems where generating a single optimality cut is significantly time consuming. Third, some of the well-known acceleration strategies for the Benders method need to be revisited in order to be properly applied in the parallel environment. An example would be the partial decomposition strategy of Crainic et al. (2016). Last but not least, heuristics are widely used to accelerate the BD method, but we are not aware of any integration of these methods in a parallel (cooperative) framework.

Acknowledgment

Partial funding for this project has been provided by the Natural Sciences and Engineer-

ing Council of Canada (NSERC), through its Discovery Grant program and by the Fonds de recherche du Québec through its Team Grant program. We also gratefully acknowledge the support of Fonds de recherche du Québec through their strategic infrastructure grants.

References

Ahmed, S., 2010. Two-stage stochastic integer programming: A brief introduction. In: Cochran, J. J., Cox, L. A., Keskinocak, P., Kharoufeh, J. P., Smith, J. C. (Eds.), Wiley Encyclopedia of Operations Research and Management Science. John Wiley & Sons, pp. 1–10.

Archibald, T. W., Buchanan, C. S., McKinnon, K. I. M., Thomas, L. C., 1999. Nested benders decomposition and dynamic programming for reservoir optimisation. Journal of the Operational Research Society 50 (5), 468–479.

Ariyawansa, K. A., Hudson, D. D., 1991. Performance of a benchmark parallel implementation of the Van Slyke and Wets algorithm for two-stage stochastic programs on the sequent/balance. Concurrency: Practice and Experience 3 (2), 109–128.

Arthur, D., Vassilvitskii, S., 2007. k-means++: The advantages of careful seeding. In: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, pp. 1027–1035.

Benders, J. F., 1962. Partitioning procedures for solving mixed-variables programming problems. Numerische mathematik 4 (1), 238–252.

Birge, J. R., Louveaux, F., 1997. Introduction to stochastic programming. Springer, New York.

Boland, N., Fischetti, M., Monaci, M., Savelsbergh, M., 2016. Proximity Benders: a decomposition heuristic for stochastic programs. Journal of Heuristics 22 (2), 181–198.

Botton, Q., Fortz, B., Gouveia, L., Poss, M., 2013. Benders decomposition for the hop-constrained survivable network design problem. INFORMS journal on computing 25 (1), 13–26.

Chermakani, D. P., 2015. Optimal aggregation of blocks into subproblems in linear programs with block-diagonal-structure. Available at https://arxiv.org/ftp/arxiv/papers/1507/1507.05753.pdf.

Chouman, M., Crainic, T. G., Gendron, B., 2017. Commodity representations and cut-set-based inequalities for multicommodity capacitated fixed-charge network design. Transportation Science 51 (2), 650–667.

Costa, A. M., 2005. A survey on Benders decomposition applied to fixed-charge network design problems. Computers & operations research 32 (6), 1429–1450.

Crainic, T. G., Fu, X., Gendreau, M., Rei, W., Wallace, S. W., 2011. Progressive hedging-based metaheuristics for stochastic network design. Networks 58 (2), 114–124.

Crainic, T. G., Hewitt, M., Rei, W., 2016. Partial Benders decomposition strategies for two-stage stochastic integer programs. Publication CIRRELT-2016-37, Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport, Université de Montréal, Montréal, QC, Canada.

Crainic, T. G., Le Cun, B., Roucairol, C., 2006. Parallel Branch-and-Bound algorithms. In: Talbi, E.-G. (Ed.), Parallel Combinatorial Optimization. John Wiley & Sons, Ch. 1, pp. 1–28.

Crainic, T. G., Toulouse, M., 1998. Parallel metaheuristics. In: Crainic, T. G., Laporte, G. (Eds.), Fleet Management and Logistics. Springer, Boston, MA, pp. 205–251.

Dantzig, G. B., Ho, J. K., Infanger, G., August 1991. Solving stochastic linear programs on a hypercube multicomputer. Tech. Rep. ADA240443, DTIC Document.

Dempster, M. A. H., Thompson, R. T., Jun 1998. Parallelization and aggregation ofnested benders decomposition. Annals of Operations Research 81 (0), 163–188.

Higle, J. L., Sen, S., 1991. Stochastic decomposition: An algorithm for two-stage linear programs with recourse. Mathematics of Operations Research 16 (3), 650–669.

Klibi, W., Martel, A., Guitouni, A., 2010. The design of robust value-creating supply chain networks: A critical review. European Journal of Operational Research 203 (2), 283 – 293.

Langer, A., Venkataraman, R., Palekar, U., Kale, L. V., Dec 2013. Parallel branch-and-bound for two-stage stochastic integer optimization. In: 20th Annual International Conference on High Performance Computing. pp. 266–275.

Latorre, J. M., Cerisola, S., Ramos, A., Palacios, R., 2009. Analysis of stochastic problem decomposition algorithms in computational grids. Annals of Operations Research 166 (1), 355–373.

Li, X., 2013. Parallel nonconvex generalized benders decomposition for natural gas production network planning under uncertainty. Computers & Chemical Engineering 55, 97 – 108.

Linderoth, J., Wright, S., 2003. Decomposition algorithms for stochastic programming on a computational grid. Computational Optimization and Applications 24 (2-3), 207–250.

Mateo, J., Plà, L. M., Solsona, F., Pagès, A., Dec 2018. A scalable parallel implementation of the cluster benders decomposition algorithm. Cluster Computing.

McDaniel, D., Devine, M., 1977. A modified Benders' partitioning algorithm for mixed integer programming. Management Science 24 (3), 312–319.

Moritsch, H. W., Pflug, G. C., Siomak, M., Mar 2001. Asynchronous nested optimization algorithms and their parallel implementation. Wuhan University Journal of Natural Sciences 6 (1), 560–567.

Naoum-Sawaya, J., Elhedhli, S., 2013. An interior-point Benders based branch-and-cut algorithm for mixed integer programs. Annals of Operations Research 210 (1), 33–55.

Nielsen, S. S., Zenios, S. A., 1997. Scalable parallel Benders decomposition for stochastic linear programming. Parallel Computing 23 (8), 1069–1088.

Ntaimo, L., 2010. Disjunctive decomposition for two-stage stochastic mixed-binary programs with random recourse. Operations Research 58 (1), 229–243.

Pacqueau, R., Francois, S., Le Nguyen, H., 2012. A fast and accurate algorithm for stochastic integer programming, appllied to stochastic shift scheduling. Publication G-2012-29, Groupe d'études et de recherche en analyse des décisions (GERAD), Université de Montréal, Montréal, QC, Canada.

Pfeiffer, L., Apparigliato, R., Auchapt, S., 2012. Two methods of pruning Benders' cuts and their application to the management of a gas portfolio. Optimization Online, Available at http://www.optimization-online.org/DB_FILE/2012/11/3683.pdf.

Rahmaniani, R., Crainic, T. G., Gendreau, M., Rei, W., 2017a. The benders decomposition algorithm: A literature review. European Journal of Operational Research 259 (3), 801 – 817.

Rahmaniani, R., Crainic, T. G., Gendreau, M., Rei, W., 2017b. A benders decomposition method for two-stage stochastic network design problems. Publication CIRRELT-2017-22, Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport, Université de Montréal, Montréal, QC, Canada.

Rahmaniani, R., Crainic, T. G., Gendreau, M., Rei, W., 2018. The benders dual decomposition method (CIRRELT-2017-22).

Ralphs, T., Ladányi, L., Saltzman, M., Sep 2003. Parallel branch, cut, and price for large-scale discrete optimization. Mathematical Programming 98 (1), 253–280.

Rei, W., Cordeau, J.-F., Gendreau, M., Soriano, P., 2009. Accelerating benders decomposition by local branching. INFORMS Journal on Computing 21 (2), 333–345.

Rockafellar, R. T., Wets, R. J.-B., 1991. Scenarios and policy aggregation in optimization under uncertainty. Mathematics of Operations Research 16 (1), 119–147.

Ruszczyński, A., 1997. Decomposition methods in stochastic programming. Mathematical Programming 79 (1), 333–353.

Sen, S., 1993. Subgradient decomposition and differentiability of the recourse function of a two stage stochastic linear program. Operations Research Letters 13 (3), 143 – 148.

Tarvin, D. A., Wood, R. K., Newman, A. M., 2016. Benders decomposition: Solving binary master problems by enumeration. Operations Research Letters 44 (1), 80 – 85.

Trukhanov, S., Ntaimo, L., Schaefer, A., 2010. Adaptive multicut aggregation for two-stage stochastic linear programs with recourse. European Journal of Operational Research 206 (2), 395–406.

Uryasev, S., Pardalos, P. M., 2013. Stochastic optimization: algorithms and applications. Vol. 54. Springer Science & Business Media.

Van Slyke, R. M., Wets, R., 1969. L-shaped linear programs with applications to optimal control and stochastic programming. SIAM Journal on Applied Mathematics 17 (4), 638–663.

Vladimirou, H., 1998. Computational assessment of distributed decomposition methods for stochastic linear

programs. European Journal of Operational Research 108 (3), 653–670.
Wolf, C., Koberstein, A., 2013. Dynamic sequencing and cut consolidation for the parallel hybrid-cut nested l-shaped method. European Journal of Operational Research 230 (1), 143 – 156.
Yang, H., Gupta, J. N., Yu, L., Zheng, L., 2016. An improved L-shaped method for solving process flexibility design problems. Mathematical Problems in Engineering 2016.
Zou, J., Ahmed, S., Sun, X. A., 2017. Multistage stochastic unit commitment using stochastic dual dynamic integer programming. Optimization Online, Available at http://www.optimization-online.org/DB_HTML/2017/05/6003.html.

Appendix A. Proof of Proposition 1

If $|\mathcal{S}_g| = 1$, the results follows immediately from the disaggregated version of the BD method Van Slyke and Wets (1969). To prove the validity of the cuts for $1 < |\mathcal{S}_g| \leq |\mathcal{S}|$, we need to show that the derived SP gives a valid lower approximation of the aggregated recourse variables for any $y \in Y$.

$$\theta_s \geq \min_{x \in \mathbb{R}^m_+} \{c^\top x_s : \ Wx_s \geq h_s - T_s y\} \qquad s \in \mathcal{S}_g$$

$$\beta_s \geq 0 \ \rightarrow \ \beta_s \theta_s \geq \min_{x \in \mathbb{R}^m_+} \{\beta_s c^\top x_s : \ Wx_s \geq h_s - T_s y\} \qquad s \in \mathcal{S}_g$$

$$\sum_{s \in \mathcal{S}_g} \beta_s \theta_s \geq \min_{x \in \mathbb{R}^{m|\mathcal{S}|}_+} \{\sum_{s \in \mathcal{S}_g} \beta_s c^\top x_s : Wx_s \geq h_s - T_s y, \ s \in \mathcal{S}_g\} \geq$$

$$\min_{x \in \mathbb{R}^{m|\mathcal{S}|}_+} \{c^\top \sum_{s \in \mathcal{S}_g} \beta_s x_s : \sum_{s \in \mathcal{S}_g} W \beta_s x_s \geq \sum_{s \in \mathcal{S}_g} \beta_s (h_s - T_s y)\}$$

The last inequality holds because we have aggregated the constraints using convex combination weights $1 \geq \beta_s \geq 0$ such that $\sum_{s \in \mathcal{S}_g} \beta_s = 1$. We next consider a variable transformation $\sum_{s \in \mathcal{S}_g} \beta_s = 1, \ x_g = \sum_{s \in \mathcal{S}_g} \beta_s x_s, \ h_g = \sum_{s \in \mathcal{S}_g} \beta_s h_s, \ T_g = \sum_{s \in \mathcal{S}_g} \beta_s T_s$. Thus,

$$\sum_{s \in \mathcal{S}_g} \beta_s \theta_s \geq \min_{x \in \mathbb{R}^m_+} \{c^\top x_g : \ Wx_g \geq h_g - T_g y\} = \max_{\alpha \in \mathbb{R}^l_+} \{(h_g - T_g y)^\top \alpha : \ W^\top \alpha \leq c\} \quad \forall y \in Y.$$

Also, we observe that the dual polyhedron is identical to a regular dual SP which indicates the validity of the feasibility cut. □

Appendix B. Proof of Proposition 2

It is trivial to observe that $\lceil \bar{y} \rceil \in Y$ due to the assumption. Let assume that the recourse cost associated with the rounded solution $\lceil \bar{y} \rceil$ is known and given by ν'_s for each $s \in \mathcal{S}$. Thus, for this integer solution, a valid upper bound can be calculated from $f^\top \lceil \bar{y} \rceil + \sum_{s \in \mathcal{S}} \rho_s \nu'_s \geq z^*$. On the other hand, based on the assumption, we have $\nu^*_s = Q(\bar{y}, s) \geq Q(\lceil \bar{y} \rceil, s) = \nu'_s$. Thus, $f^\top \lceil \bar{y} \rceil + \sum_{s \in \mathcal{S}} \rho_s \nu^*_s \geq f^\top \lceil \bar{y} \rceil + \sum_{s \in \mathcal{S}} \rho_s \nu'_s \geq z^*$. □

Appendix C. Proof of Theorem 1

For an arbitrary first-stage solution $y \in Y$, let $\sigma^g = (h_g - T_g y)^\top \alpha_g$ be the right hand side of the optimality cut generated from artificial scenario $g \in \mathcal{G}$ and let $\theta_s^* = (h_s - T_s y)^\top \alpha_s^{i^*}$, where $i^* \in \arg\max_{i \in E_s^t} (h_s - T_s y)^\top \alpha_s^i$ for which E_s^t is the set of optimality cuts associated with scenario s at iteration t. If the cut from the artificial SP is violated by the y solution, it means that $\sigma^g > \sum_{s \in \mathcal{S}_g} \beta_s \theta_s^*$. For a given y solution, the MP can be separated for each cluster $g \in \mathcal{G}$ and can be defined as:

$$\begin{aligned} \min \sum_{s \in \mathcal{S}_g} \rho_s \theta_s \\ \sum_{s \in \mathcal{S}_g} \beta_s \theta_s \geq \sigma^g \\ \theta_s \geq \theta_s^* \qquad s \in \mathcal{S}_g. \end{aligned}$$

It can be shown that the above formulation has a knapsack structure and since $\sigma^g > \sum_{s \in \mathcal{S}_g} \beta_s \theta_s^*$, the recourse variable $\theta_{\tilde{s}}$ takes the extra violation $\sigma^g - \sum_{s \in \mathcal{S}_g} \beta_s \theta_s^* > 0$, where $\tilde{s} \in \arg\min_{s \in \mathcal{S}_g} \frac{\rho_s}{\beta_s}$. This gives $\theta_{\tilde{s}} = \theta_{\tilde{s}}^* + \frac{1}{\beta_{\tilde{s}}}(\sigma^g - \sum_{s \in \mathcal{S}_g} \beta_s \theta_s^*)$ which results in a lower bound improvement of $\Delta := \frac{\rho_{\tilde{s}}}{\beta_{\tilde{s}}}(\sigma^g - \sum_{s \in \mathcal{S}_g} \beta_s \theta_s^*)$ at the given y solution. The maximum value of Δ is achieved when $\beta_s = \rho_s$. Considering $\sum_{s \in \mathcal{S}_g} \beta_s = 1$ and $\mathcal{S}_\} \leq |\mathcal{S}|$, the maximum value of Δ is achieve at $\beta_s = \frac{\rho_s}{\sum_{s \in \mathcal{S}_g} \rho_s}$. □

Appendix D. Restricted MILP Problem

This problem is derived by considering the extensive formulation for a small subset of scenarios and fixing variables to 1 if their current value is greater than $1 - \lambda$ and to 0 if it is less than $\hat{\lambda}$.

Appendix E. Stochastic Network Design Problem

The MCFNDSD is defined on a directed graph consisting a set of nodes $\mathcal{N}$ and a set of potential arcs $\mathcal{A}$. In this problem, a set of commodities $\mathcal{K}$ exist where each commodity $k \in \mathcal{K}$ has an uncertain amount of demand that needs to be routed from its unique origin $O(k) \in \mathcal{N}$ to its unique destination $D(k) \in \mathcal{N}$. We assume that the demand is characterized with a set of discrete scenarios $\mathcal{S}$. Therefore, each commodity $k \in \mathcal{K}$ has a stochastic demand which is denoted by $d_s^k \geq 0$ for each possible scenario $s \in \mathcal{S}$. We assume that the realization probability for each scenario $s \in \mathcal{S}$ is known and denoted by ρ_s such that $\sum_{s \in \mathcal{S}} \rho_s = 1$. The goal is to select a proper subset of the arcs to meet all the flow requirements at minimum cost. To use arc $a \in \mathcal{A}$ we need to pay a fixed cost of f_a units and to route a unit of commodity $k \in \mathcal{K}$ on this arc c_a^k units will be charged. In addition, there is a capacity limit u_a on each arc $a \in \mathcal{A}$. Thus, the objective function is to minimize sum of the fixed costs and the expected flow costs.

To model this stochastic problem we define the binary first-stage variables y_a indicating if arc $a \in \mathcal{A}$ is used 1 or not 0. To model the second-stage, we define continuous variables $x_a^{k,s} \geq 0$ to reflect the amount of flow on arc $a \in \mathcal{A}$ for commodity $k \in \mathcal{K}$ under realization $s \in \mathcal{S}$. The extensive formulation of MCFNDSD is thus:

$$MCFNDSD = \min_{y \in \{0,1\}^{|\mathcal{A}|}, x \in \Re_+^{|\mathcal{A}||\mathcal{K}||\mathcal{S}|}} \sum_{a \in \mathcal{A}} f_a y_a + \sum_{s \in \mathcal{S}} \rho_s \sum_{k \in \mathcal{K}} \sum_{a \in A} c_a^k x_a^{k,s} \tag{E.1}$$

$$\text{s.t:} \sum_{a \in \mathcal{A}(i)^+} x_a^{k,s} - \sum_{a \in \mathcal{A}(i)^-} x_a^{k,s} = \begin{cases} d_s^k & \text{if } i = O(k) \\ -d_s^k & \text{if } i = D(k) \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in \mathcal{N}, k \in \mathcal{K}, s \in \mathcal{S} \tag{E.2}$$

$$\sum_{k \in \mathcal{K}} x_a^{k,s} \leq u_a y_a \qquad \forall a \in \mathcal{A}, s \in \mathcal{S}, \tag{E.3}$$

where $\mathcal{A}(i)^+$ and $\mathcal{A}(i)^-$ indicate the set of outward and inward arcs incident to node i. The objective function minimizes the total fixed costs plus the expected routing costs. For each scenario, constraint set (E.2) imposes the flow conservation requirements for each commodity and node. Constraints (E.3) enforce the capacity limit on each arc in every scenario. To improve performance, we introduce the complete recourse property for the above formulation, we add a dummy arc between each O-D pair with large routing cost as an outsourcing strategy.

Appendix F. Test Instances

Table F.8 presents the detail of the used instances problems from **R** family which we used in this article.

Table F.8: Attributes of the instance classes

Name	$\|N\|$	$\|A\|$	$\|K\|$	$\|\Omega\|$	Cost/Capacity Ratio	Correlation	#Instances
r04	10	60	10	1000	1, 3, 5, 7, 9	0, 0.2, 0.4, 0.6, 0.8	25
r05	10	60	25	1000	1, 3, 5, 7, 9	0, 0.2, 0.4, 0.6, 0.8	25
r06	10	60	50	1000	1, 3, 5, 7, 9	0, 0.2, 0.4, 0.6, 0.8	25
r07	10	82	10	1000	1, 3, 5, 7, 9	0, 0.2, 0.4, 0.6, 0.8	25
r08	10	83	25	1000	1, 3, 5, 7, 9	0, 0.2, 0.4, 0.6, 0.8	25
r09	10	83	50	1000	1, 3, 5, 7, 9	0, 0.2, 0.4, 0.6, 0.8	25
r10	20	120	40	1000	1, 3, 5, 7, 9	0, 0.2, 0.4, 0.6, 0.8	25
r11	20	120	100	1000	1, 3, 5, 7, 9	0, 0.2, 0.4, 0.6, 0.8	25

Appendix G. Numerical results of the sequential algorithm

To make fair comparisons, we have incorporated the techniques that we developed in sections 4.4.2 , 5.1 and 5.4 into our sequential method (presented in Algorithm 1). We have also incorporated the classical acceleration techniques which we have used in our parallel algorithms, see section 6.

In this appendix, we present some numerical results to complement our numerical assessments in section 7. In Figure G.7, we thus study impact of the cut aggregation over the sequential algorithm. For each aggregation level, we have ran the algorithm for 2 hours. The value on each column indicates the average optimality gap in percentages.

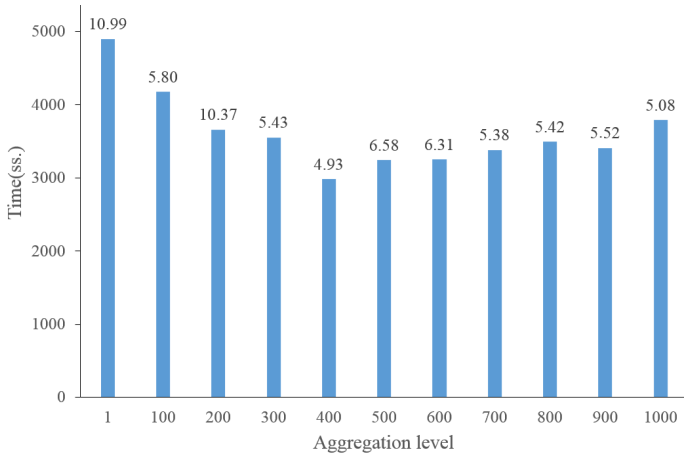

Figure G.7: Comparison of various cut aggregation levels for the sequential B&BC method

We next present the numerical performance of the sequential algorithm for a run time limit of 10 hours. The results are presented in Table G.9.

Table G.9: Numerical results of the sequential B&BC algorithm

	#Instance	Time to solve LP	Total Time	Gap(%)	Sol.(%)
r04	3	31.43	1098.21	0.36	100.00
r05	3	159.20	798.77	0.47	100.00
r06	3	775.99	14671.58	1.23	66.67
r07	3	34.81	12087.66	0.76	66.67
r08	3	224.44	12259.00	2.34	66.67
r09	3	1188.19	24270.19	2.79	33.33
r10	3	3582.01	24521.04	6.80	33.33
Ave.	**3**	**856.58**	**12815.21**	**2.11**	**66.67**