Projective Hedging Algorithms for Distributed Optimization under Uncertainty*

Jonathan Eckstein† Jean-Paul Watson‡ David L. Woodruff§

April 11, 2022

Abstract

We propose a decomposition algorithm for multistage stochastic programming that resembles the progressive hedging method of Rockafellar and Wets, but is provably capable of several forms of asynchronous operation. We derive the method from a class of projective operator splitting methods fairly recently proposed by Combettes and Eckstein, significantly expanding the known applications of those methods. Our derivation assures convergence for convex problems whose feasible set is compact, subject to some standard regularity conditions and a mild "fairness" condition on subproblem selection. The method's convergence guarantees are deterministic and do not require randomization, in contrast to some other proposed asynchronous variations of progressive hedging. We describe a distributed implementation of the method within the `mpi-sppy` system and present the results of computational experiments on up to a million scenarios and using as many as 2,400 processor cores in an HPC (high-performance computing) environment. These experiments evaluate the performance of the method when operating in a "block asynchronous" manner, meaning that it still alternates between non-overlapping decomposition and coordination phases, but only a subset of the subproblems are solved during each decomposition phase. Using a particular "greedy" heuristic to select which subproblems to solve, our experiments show that this tactic can make significantly more efficient use of computational resources than the original form of progressive hedging.

*Funded in part by National Science Foundation grant CCF-1617617

†Department of Management Science and Information Systems, Rutgers Business School Newark and New Brunswick, Rutgers University

‡Center for Applied Scientific Computing and Global Security Directorate, Lawrence Livermore National Laboratory

§Graduate School of Management, University of California, Davis

1 Introduction

The progressive hedging (PH) method for stochastic programming is a decomposition algorithm for multistage convex stochastic programming problems introduced by Rockafellar and Wets [31]. Consider a stochastic programming problem defined over a finite multistage scenario tree with n leaf nodes. We henceforth use the the term "scenario" to refer to an entire path through this tree from the root to one of the leaves. PH cycles through the following steps:

1. Solve each of the n scenarios as a separate subproblem, with a quadratic perturbation to the objective function.

2. For each non-leaf node in the scenario tree, average the portions of the scenario solutions that correspond to it.

3. Perform a Lagrange multiplier update.

The quadratic perturbation in each scenario subproblem includes both a linear Lagrange multiplier term and a quadratic penalty for deviating from the averaged solution found in step 2 of the previous iteration.

PH is a naturally parallel algorithm, since the n subproblems in step 1 may be solved concurrently, and the remaining calculations are quite simple and may readily be distributed and executed in parallel using standard communication patterns whose time requirements grow only logarithmically with the number of processors. However, the original form of PH is also strongly synchronous, in that every scenario subproblem must be fully solved before an iteration can be completed. This synchronousness has a number of drawbacks: first, even at iterations in which only a few subproblems diverge significantly from the globally averaged solution, all of them must be processed. Furthermore, if the run times of the subproblems are highly variable, most processors will accumulate significant idle time while waiting for the slowest subproblems to complete. This drawback is particularly acute when PH is applied to problems with integer variables, as in [37], since the subproblems are mixed-integer programs. Initially, such applications of PH were largely heuristic, but later it has been shown [5, 17] that with suitable adjustments the technique can find useful Lagrangian lower bounds on the global integer-optimal solution value.

This paper introduces a class of stochastic programming algorithms that resemble PH, but are much less synchronous in two ways: first, only a subset of the subproblems need to solved at each iteration, subject to some very loose restrictions. We call this feature of the algorithm being "block asynchronous" (in some earlier work, this mode of operation is called "block iterative" or "block activated"). The second departure from synchronousness is that subproblems may be "dispatched" for solution in one iteration, but their results incorporated into the coordination calculations some bounded number of iterations later. This feature is likely to be useful when the time to solve the subproblems is highly variable.

Our algorithms are an application of an operator splitting method described in [12], which is essentially a simplification of one of the methods developed in [8]. Although the original convergence proof for PH in [31] is self-contained, the method is an application of the alternating direction method of multipliers (ADMM) [15, 16] in a suitably formulated

space with a specially chosen inner product. As noted in [16], the ADMM is in turn an application of a ubiquitous general algorithmic template known as Douglas-Rachford splitting [27]. Broadly speaking, our new class of algorithms is obtained by substituting the relatively new asynchronous projective splitting algorithm of [12] for the more traditional Douglas-Rachford approach [13, 16, 27]. However, there are numerous differences in detail: for example, PH is an application of a two-operator splitting scheme in an n-way product space, whereas our new approach uses an n-operator splitting scheme. Since it is based on the asynchronous projective splitting methods proposed in [8, 12], we call our new method "Asynchronous Projective Hedging" (APH).

It should be noted that [12] contains two algorithms, an abstract monotone operator splitting method and a method resembing an asynchronous n-block implementation of the ADMM, the second being an application of the first. The stochastic programming method we present here is also an application of the first, more abstract method in [12], and thus of one of the methods in [8]. However, it is not an application of the second, ADMM-like method in [12]; attempting to apply the second method in [12] to stochastic programming would require representing the nonanticipativity of the solution through explicit linear constraints and appears to result in a factor of n increase in working storage as compared to the algorithm developed here.

While the fundamental convergence "engine" behind our algorithm is the abstract monotone operator splitting procedure from [8, 12] and is therefore not novel, we believe that new applications of this "engine" constitute a noteworthy contribution. This class of algorithms is relatively recent and only [23] and the preprint [6] describes its concrete applications, giving examples in data analysis and image recovery. Thus, we believe that adding a very different application of [8, 12] to the literature is of significant value. Further, considerable care must be exercised when deriving the method from [8, 12] in order to produce an algorithm whose general form and working storage requirements are comparable to those of PH.

After deriving the the APH algorithm, we present computational experiments with a distributed implementation in a high-performance computing (HPC) setting, solving very large-scale, relatively realistic stochastic programming instances on as may as 2,400 processor cores. We constructed the implementation within the `mpi-sppy` package [25], allowing for direct comparison with the PH implementation already in that software.

For this paper, we have restricted computational testing to cases for which our current analysis proves convergence, namely continuous problems with convex objective functions and constraints. Within these classes of problems, we did not observe any test cases in which subproblem solution times had significant variance, so our computational tests do not address the entire possible range of departures from synchronous behavior of which the proposed method is theoretically capable. However, our tests do thoroughly explore "block-asyncronous" operation, meaning that, although the processors alternate in a fully coordinated manner between subproblem solution and coordination phases, only a subset of the subproblems needs to be solved during each subproblem-solving phase. This property is sufficient to give the algorithm a very significant efficiency advantage over PH. Some of our computational tests involve problems for which there is no practical alternative to PH, so improving its efficiency is a noteworthy contribution.

2 Related Work

PH has been implemented with asynchronous subproblem solving in [10, 29, 32, 36], but these efforts were of a heuristic nature.

Rigorously derived, convergence-guaranteed asynchronous versions of PH were first published in an earlier preprint of this work (under a slightly different title) and by Bareilles *et al.* in [2]. The latter work presents three randomized versions of PH, each of which has a convergence proof based on a distinct underlying randomized block-coordinate fixed-point algorithm. The first of these versions of PH is block asynchonous, processing one randomly chosen scenario per iteration. The convergence theory of this version is based the randomized fixed-point algorithm developed in [22], where it was also used to derive an asychronous ADMM for certain optimization problems defined over networks. The second version of PH from [2] is also block asynchronous, processing a fixed-size but randomly chosen subset of scenarios at each iteration. The validity of this algorithm may be verified by using the more general randomized fixed-point iteration developed by Combettes and Pesquet in [9]. The final version of PH in [2] is potentially fully asynchronous, a property facilitated in the algorithm statement by allowing subproblem calculations to be based on information up to τ iterations old, where τ is some given integer. This version's theoretical underpinnings are the "ARock" randomized fixed-point procedure of Peng *et al.* [28].

In the ARock-based version of PH in [2], only one scenario can be processed per iteration, and the steps taken by the algorithm must be under-relaxed by a factor strictly below $n/(n+2\tau)$, where n is the number of scenarios. The one-scenario-per-iteration requirement of the algorithm could effectively increase τ: for example, suppose that the algorithm implementation simultaneously initiates processing of 5 scenario subproblems whose solutions all arrive simultaneously 3 iterations later. These scenario solutions must be incorporated into the master solution estimate one-by-one, with the first having a perceived delay of 3, the second having a perceived delay of 4, and so forth, with the fifth having a perceived delay of $3+5-1=7$, meaning that one must set $\tau \geq 7$ to ensure convergence. The convergence theory requires that larger values of τ require more under-relaxation of the algorithm stepsize, which can be expected to slow down convergence.

In all the algorithms from [2], the analysis relies on random selection of scenarios and yields *almost sure* convergence, meaning that over all possible infinite sequences of scenarios that might be selected over the course of the algorithm, those that exhibit convergence constitute a set with probability one. While this is a fairly strong guarantee and very common in algorithms associated with machine learning problems, it should be remembered that there are uncountably many such sequences, so the result does not actually assure convergence along any particular one.

The work presented here is fundamentally different, being based on the projective splitting methods in [8, 12]. These methods do not require randomization and deterministically guarantee convergence along any algorithmic path satisfying certain "fairness" conditions, so this same is true for the PH-like methods derived from them here. These resulting methods support various kinds of asynchronism, but with some important differences from algorithms derived from randomized fixed-point methods. In our approach, any number of subproblems may be processed per iteration, and while the maximum delay to process a scenario must be bounded, the bound does not appear explicitly in the algorithm's calculations, but only in

the convergence proof. As a result, the maximum delay need not be explicitly estimated and no delay-based underrelaxation is required. These advantages come at the cost of some additional per-scenario working variables that must undergo some relatively simple, linear-time updates at each iteration, including iterations in which the corresponding subproblem was not reoptimized. In a shared-memory environment, our method could therefore exhibit more memory contention than the algorithms in [2]. In pure master-worker implementations, our proposed methods could also somewhat increase the per-iteration workload on the master as compared to simpler algorithms. However, in a scalable distributed implementation without a centralized master processor, such as the one presented in Section 7 below, it is not clear whether the iterate-updating pattern of our algorithm has any implementation disadvantages as compared to the methods in [2].

The deterministic convergence proof for our method not only yields a technically stronger guarantee of validity, but more importantly allows subproblems to be selected for processing in a non-random manner. Specifically, Section 7 of this paper demonstrates a "greedy" non-random subproblem selection procedure that could not be applied in a randomized algorithmic setting.

Section 7 also presents computational results for much larger problems and on much more highly parallel hardware than the experiments in [2].

Since PH is an application of the ADMM, it is natural to ask whether any other prior proposals for asynchronous ADMM variants, besides those discussed immediately above, could yield asynchronous versions of PH. One possible candidate is the asynchronous ADMM proposed by Wei and Ozdaglar in [38]. It appears possible to derive a variant of PH from this method, but because of its strong graph orientation and related restrictive constraint structure assumptions, it does not appear that such a process would be very natural, nor would it offer any obvious advantages over the algorithms discussed above. Any such algorithm would, like those of [2], require randomized scenario selection and exhibit almost-sure convergence.

The other algorithm in the literature that can fairly be described as a rigorously founded asynchronous ADMM is Hong's method from [21]. While it is suitable for some machine-learning problems and has a local convergence analysis for nonconvex objectives, this algorithm requires assumptions that do not apply when developing decomposition methods for stochastic programming. It therefore appears that PH procedures based on randomized fixed-point iterations as in [2, 9, 22, 28] are currently the most plausible competitor algorithms to the one proposed here. Ideally, however, one would want to combine the delay tolerance of ARock-based algorithms [28] with the ability to process multiple blocks per iteration as in [9], but we are not presently aware of any randomized fixed-point algorithm with these properties.

3 Notation and Problem Setup

Consider a stochastic programming problem defined on a finite scenario tree $\mathcal{T}$ with stages indexed by $t \in 1..T$, and n leaf nodes indexed by $i \in 1..n$. Let $\mathcal{T}_t$ denote the set of tree nodes at each stage $t \in 1..T$, and for each $N \in \mathcal{T}_t$, we let $\mathcal{S}(N) \subseteq 1..n$ denote the set of leaf nodes that have N as an ancestor. The sets $\{\mathcal{S}(N) \mid N \in \mathcal{T}_t\}$ comprise a partition of $1..n$

for each $t \in 1..T$. For each $i \in 1..n$, we denote the probability of leaf node i by $\pi_i \in (0, 1]$, subject to $\sum_{i=1}^{n} \pi_i = 1$. For each node N in the tree, we also let $\pi(N) = \sum_{i \in \mathcal{S}(n)} \pi_i$ denote its probability. We assume that all zero-probability nodes have already been pruned from the scenario tree.

For simplicity (and without loss of generality), we assume that every stage-t node in the tree has the same number of decision variables $m_t \geq 1$, and we let $m = \sum_{t=1}^{T} m_t$ denote the full number of decision variables associated with each scenario. For each $i \in 1..n$, let $\mathcal{X}_i$ denote the space of vectors of possible decisions for scenario i, from all stages. We denote a generic element of $\mathcal{X}_i$ by x_i, and its subvector corresponding to each stage $t \in 1..T$ by $x_{it} \in \mathbb{R}^{m_t}$. We give $\mathcal{X}_i$ the inner product

$$\langle x_i, y_i \rangle_{\mathcal{X}_i} = \pi_i (x_i)^\top (y_i) = \pi_i \sum_{t=1}^{T} (x_{it})^\top (y_{it}), \tag{1}$$

We use the scaled inner product (1) to maximize the resemblance of our algorithm to the original PH method proposed in [31]. It is also possible to use an unweighted inner product for $\mathcal{X}_i$ in the analysis below, resulting in a very similar algorithm development, but with different averaging operations.

Let $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_n$, under the inner product induced by (1),

$$\langle x, y \rangle_{\mathcal{X}} = \sum_{i=1}^{n} \langle x_i, y_i \rangle_{\mathcal{X}_i} = \sum_{i=1}^{n} \pi_i (x_i)^\top (y_i) = \sum_{i=1}^{n} \pi_i \sum_{t=1}^{T} (x_{it})^\top (y_{it}). \tag{2}$$

One may envision the elements of $\mathcal{X}$ as expanding decision vectors defined on the scenario tree into a "grid" by making $|\mathcal{S}(N)|$ replicas of the decision variables at each node N in the tree. Thus, vectors in $\mathcal{X}$ have n sets of decision variables for each stage t, regardless of the number of tree nodes $|\mathcal{T}_t|$ at stage t.

Next, let $m' = m - m_T = \sum_{t=1}^{T-1} m_t$. For each $i \in 1..n$, let $\mathcal{Z}_i$ denote the space of all decision variables for scenario i, except those for stage T. We use a similar indexing scheme and inner product to $\mathcal{X}_i$, so that

$$\langle z_i, u_i \rangle_{\mathcal{Z}_i} = \pi_i (z_i)^\top (u_i) = \pi_i \sum_{t=1}^{T-1} (z_{it})^\top (u_{it}). \tag{3}$$

Let $\mathcal{Z} = \mathcal{Z}_1 \times \cdots \times \mathcal{Z}_n$ with the inner product induced by (3),

$$\langle z, u \rangle_{\mathcal{Z}} = \sum_{i=1}^{n} \langle z_i, u_i \rangle_{\mathcal{Z}_i} = \sum_{i=1}^{n} \pi_i (z_i)^\top (u_i) = \sum_{i=1}^{n} \pi_i \sum_{t=1}^{T-1} (z_{it})^\top (u_{it}). \tag{4}$$

Vectors in $\mathcal{Z}$ have the same "grid" interpretation as vectors in $\mathcal{X}$, except that the are missing the variables for the final stage. Essentially, $\mathcal{X}$ is $\mathbb{R}^{mn}$ and $\mathcal{Z}$ is $\mathbb{R}^{m'n}$, but with scenario-weighted inner products.

For each $i \in 1..n$, let M_i denote the linear map $\mathcal{X}_i \to \mathcal{Z}_i$ that simply drops that last-stage variables, $M_i(x_{i1}, \ldots, x_{iT}) = (x_{i1}, \ldots, x_{i,T-1})$. We then let $M : \mathcal{X} \to \mathcal{Z}$ be the linear map

given by

$$M : \big((x_{11}, \ldots, x_{1T}), \ldots, (x_{n1}, \ldots, x_{nT})\big) \mapsto (M_1 x_1, \ldots, M_n x_n) \\ = \big((x_{11}, \ldots, x_{1,T-1}), \ldots, (x_{n1}, \ldots, x_{n,T-1})\big), \quad (5)$$

which simply drops all the last-stage variables from its argument.

As in [31], we define a linear subspace $\mathcal{N}$ of $\mathcal{Z}$ by

$$\mathcal{N} = \left\{ z \in \mathcal{Z} \;\middle|\; \Big(\big(\forall\, t \in 1..(T-1)\big)\ (\forall\, N \in \mathcal{T}_t)\ \Big(\forall\, i, j \in \mathcal{S}(N)\Big) : z_{it} = z_{jt} \right\}. \quad (6)$$

Vectors in $z \in \mathcal{N}$ are *nonanticipative* in the sense that $z_{it} = z_{jt}$ whenever scenarios i and j are indistinguishable at stage t. They thus correspond to implementable plans for stages $1, \ldots, T-1$.

For each $i \in 1..n$, let $h_i : \mathbb{R}^m \to \mathbb{R} \cup \{+\infty\}$ be a closed proper convex function and consider the optimization problem

$$\begin{array}{ll} \min\limits_{x \in \mathcal{X}} & \sum\limits_{i=1}^{n} \pi_i h_i(x_i) \\ \text{ST} & Mx \in \mathcal{N}. \end{array} \quad (7)$$

This optimization model can subsume any convex stochastic programming problem defined on the scenario tree $\mathcal{T}$, in the following manner: within the context of the "clairvoyant" situation in which one knows that leaf node $i \in 1..n$ of the tree will occur, define $h_i(x_i) = +\infty$ whenever x_i is infeasible, and otherwise let $h_i(x_i)$ be the total cost of the plan described by x_i. In particular, $h_i(x_i)$ will be $+\infty$ if x_i violates any constraint within a stage or any coupling constraint between stages. For all scenarios $i \in 1..n$, such constraints are embedded within the objective function of (7), while the constraint in (7) requires the selection of an implementable, non-clairvoyant plan.

One can now derive the PH algorithm by defining convex functions $F : \mathcal{X} \to \mathbb{R} \cup \{+\infty\}$ and $G : \mathcal{Z} \to \mathbb{R} \cup \{+\infty\}$ as follows:

$$F(x) = \sum_{i=1}^{n} \pi_i h_i(x_i) \qquad\qquad G(z) = \begin{cases} 0, & z \in \mathcal{N} \\ +\infty, & z \notin \mathcal{N}. \end{cases} \quad (8)$$

Problem (7) may then be expressed as simply

$$\min_{x \in \mathcal{X}} \big\{ F(x) + G(Mx) \big\}, \quad (9)$$

which is a standard form for applying the ADMM. Doing so while keeping in mind the inner products defined in (2) and (4), yields the PH method.

The ADMM for the problem (9) interacts with F by solving partial augmented Lagrangian subproblems of the form $\min_x \big\{ F(x) + \langle w^k, x \rangle + \frac{\rho}{2} \big\| x - z^k \big\| \big\}$, where $w^k \in \mathcal{N}^\perp$ and $z^k \in \mathcal{N}$ are iterates of the method and $\rho > 0$ is a constant parameter. With F defined as in (8), this subproblem necessarily involves solving an optimization problem for all n

scenarios. The resulting method (progressive hedging) thus treats the scenarios completely synchronously, and the same would apply to any algorithm based directly on decomposition of the two-function formulation (9). So, to obtain a method in which scenarios are more loosely coordinated, we avoid the formulation (9) and will instead formulate the stochastic program through the n-function model problem

$$\min_{z \in \mathcal{H}_0} \left\{ \sum_{i=1}^{n} f_i(L_i z) \right\}, \tag{10}$$

where $\mathcal{H}_0, \ldots, \mathcal{H}_n$ are real Hilbert spaces with respective inner products $\langle \cdot\,, \cdot \rangle_{\mathcal{H}_0}, \ldots, \langle \cdot\,, \cdot \rangle_{\mathcal{H}_n}$, and, for all $i \in 1..n$, $f_i : \mathcal{H}_i \to \mathbb{R} \cup \{\infty\}$ is a closed proper convex function and $L_i : \mathcal{H}_0 \to \mathcal{H}_i$ is a bounded linear map. To apply it to stochastic programming problem (7), we assign the elements of this model as follows:

S1. $\mathcal{H}_0 = \mathcal{N}$, the nonanticipativity subspace, under the inner product $\langle \cdot\,, \cdot \rangle_{\mathcal{Z}}$ from (4).

S2. For all $i \in 1..n$, we let $\mathcal{H}_i = \mathcal{Z}_i$, thus using the inner product (3).

S3. For all $i \in 1..n$, we define L_i by $L_i : (z_1, \ldots, z_n) \mapsto z_i$, that is, L_i selects only the portion of its argument corresponding to scenario i.

S4. For all $i \in 1..n$, we define f_i by

$$f_i(z_i) = \pi_i \min\bigl\{ h_i\bigl((z_i, x_{iT})\bigr) \;\bigm|\; x_{iT} \in \mathbb{R}^{m_T} \bigr\}, \tag{11}$$

that is, given the values of all decision variables z_i for the scenario i except those for the last stage, f_i optimizes over the last-stage variables and returns the resulting objective value scaled by π_i.

This approach to applying (10) to stochastic programming, in which we implicitly minimize over the last-stage variables within the functions f_i, may seem slightly unnatural, but it is necessary to avoid generating undesired quadratic terms associated with the last-stage variables in the algorithm we will develop below.

We make the following blanket assumption for the remainder of this paper:

Assumption 1 *For all $i \in 1..n$, $\operatorname{dom} h_i = \{x_i \in \mathbb{R}^m \mid h_i(x_i) < \infty\}$ is compact.*

This assumption will allow us to model any finite-dimensional stochastic programming problem on the tree $\mathcal{T}$ whose feasible region is closed and bounded, a very weak restriction in most practical settings. It is also possible to relax this restriction, but we adopt it here in the interest of simplicity.

Proposition 2 *Under Assumption 1, the functions f_i defined by* (11) *are closed proper convex for all $i \in 1..n$.*

Proof. Closedness is established in [3, Lemma 1.30], while convexity is established in [3, Proposition 8.35]. □

Proposition 3 *Suppose that $\mathcal{H}_0, \ldots, \mathcal{H}_n$, $f_1, \ldots, f_n$, and $L_1, \ldots, L_n$ are constructed as in items S1-S4 above and Assumption 1 holds. Then a vector $z^* \in \mathbb{R}^{m'}$ is an optimal solution of the problem (10) if and only if it is of the form $z^* = Mx^*$, where x^* is an optimal solution of (7) and M is as defined in (5).*

Proof. Under Assumption 1, problem (7) must attain its minimum value, since its feasible region is the intersection of the compact set $\prod_{i=1}^n \operatorname{dom} h_i$ with the (necessarily closed) linear subspace $\{x \in \mathcal{X} \mid Mx \in \mathcal{N}\}$. Thus, it is the minimum value of a closed (lower semicontinuous) function over a compact set and must be attained; see for example [3, Theorem 1.29]. Under the stated hypotheses, problem (10) must also attain its minimum, because, for each $i \in 1..n$,

$$\operatorname{dom} f_i = \left\{ z_i \in \mathbb{R}^{m'} \mid \exists\, x_{iT} \in \mathbb{R}^{m_T} : (z_i, x_i T) \in \operatorname{dom} h_i \right\} = M_i \operatorname{dom} h_i. \tag{12}$$

Since $\operatorname{dom} h_i$ is compact, this set is also compact. By Proposition 2, the functions f_i are closed proper convex. The feasible region of problem (10), namely

$$\mathcal{F} = \{ z \in \mathcal{N} \mid (\forall\, i \in 1..n)\, L_i z \in \operatorname{dom} f_i \},$$

is clearly closed since the $\operatorname{dom} f_i$ are compact. It also cannot contain an unbounded sequence of points $\{z^k\}$, since at least one of the corresponding sequences $\{L_i z^k\}$, $i \in 1..n$, would then have to be unbounded and could not be contained in $\operatorname{dom} f_i$. Therefore $\mathcal{F}$ is compact since $\mathcal{N}$ is finite dimensional. Since the f_i are lower semicontinuous (closed), it is easily seen that $\sum_{i=1}^n f_i \circ L_i$ must also be lower semicontinous. Hence, problem (10) also involves minimization of a lower semicontinuous function over a compact set and attains its minimum. We then calculate

$$\begin{aligned}
\min_{z \in \mathcal{H}_0} \left\{ \sum_{i=1}^n f_i(L_i z) \right\} &= \min_{z \in \mathcal{N}} \left\{ \sum_{i=1}^n f_i(z_i) \right\} \\
&= \min_{z \in \mathcal{N}} \left\{ \sum_{i=1}^n \pi_i \min \left\{ h_i\big((z_i, x_{iT})\big) \mid x_{iT} \in \mathbb{R}^{m_T} \right\} \right\} \\
&= \min \left\{ \sum_{i=1}^n \pi_i h_i\big((z_i, x_{iT})\big) \;\middle|\; z \in \mathcal{N}, (\forall\, i \in 1..n)\, x_{iT} \in \mathbb{R}^{m_T} \right\} \\
&= \min \left\{ \sum_{i=1}^n \pi_i h_i(x_i) \;\middle|\; x \in \mathcal{X} : Mx \in \mathcal{N} \right\},
\end{aligned}$$

so the objective values of the two problems are equal.

Suppose x^* is an optimal solution of (7). If we let $z^* = Mx^*$, then for all $i \in 1..n$ we have $f(L_i z_i^*) \le \pi_i h(L_i z_i^*, x_{iT}^*) = \pi_i h_i(x_i^*)$. Summing, we have $\sum_{i=1}^n f_i(L_i z^*) \le \sum_{i=1}^n \pi_i h(x^* i)$, which in view of the just-established equivalence of optimal values means that z^* is optimal for (10). Conversely, let z^* be any optimal solution of (10), consider for each $i \in 1..n$ the problem $\min_{x_{iT} \in \mathbb{R}^{m_T}} \left\{ h_i\big((z_i^*, x_{iT})\big) \right\}$, which must be attained at some $x_{iT}^* \in \mathbb{R}^{m_T}$ since it is again the minimum of a lower semicontinuous function over a compact set. In view of the equality of optimal values of the two problems already established, $x^* = \big((z_1^*, x_{1T}^*) \ldots, (z_n^*, x_{nT}^*)\big)$ must be an optimal solution of (7), and we clearly have $Mx^* = z^*$. □

4 A General Asynchronous Splitting Method for Optimization

At this point, we have recast our stochastic programming problem into the relatively abstract, general optimization form (10). We next follow a two-step process to arrive at a solution procedure for stochastic programming problems in the form (7):

- Specialize the monotone inclusion algorithm given in [12] to solve any convex optimization problem in the form (10).
- Further specialize the resulting algorithm to particular case of (7) using the construction from Section 3 above.

This section handles the first of these steps, resulting in Algorithm 1 below, and the next section handles the second step. In specializing the monotone inclusion algorithm of [12] to the minimization problem (10), we need to consider three particular issues:

1. The standard sufficient optimality condition of (10) is the monotone inclusion given below as (14). So, we will solve (10) by applying the monotone inclusion algorithm of [12] to this sufficient optimality condition. Unfortunately, the sufficient optimality condition may fail to be necessary for optimality in some atypical cases, so we state a constraint-qualification-like assumption (Assumption 4 immediately below) guaranteeing that the condition is necessary. Without such a condition, it is possible that our algorithm might not be able find an optimal solution to (10) even though one exists.
2. The monotone inclusion algorithm of [12] handles each subproblem with an abstract "Prox" operation on a monotone operator. To obtain an implementable method for optimization, we specialize these operations to minimizations in Algorithm 1. This specialization is explained and justified in Proposition 6 below.
3. Having addressed the previous point, the general convergence theorem of [12] applies to Algorithm 1. We state the resulting conclusions in Proposition 7 below. Because there are some minor notation adjustments between the two methods and we are claiming a few additional convergence properties not explicitly established in [12], we provide a short proof for this proposition. This proof does not repeat any of the lengthy core analysis needed to establish the results in [12].

We begin by addressing the first issue:

Assumption 4 *Assume that at least one of the following holds:*

1. *The functions $h_1, \ldots, h_n$ are polyhedral, or*
2. *There exists $\bar{x} \in \mathcal{X}$ such that, for all $i \in 1..n$,*

$$(\bar{x}_{i1}, \ldots, \bar{x}_{i,T-1}) \in \operatorname{ri} M_i(\operatorname{dom} h_i), \tag{13}$$

where the operator ri *denotes taking the relative interior of a set as defined for example in [4, Section 1.4].*

Assumption 4(1) will hold in the case of linear stochastic programming problems, while the nonlinear case is handled by Assumption 4(2), which is essentially a Slater-class constraint qualification. For a typical nonlinear convex stochastic programming formulation translated into (10) through S1-S4, the assumption essentially states that there exists a point that is strictly feasible with respect to all inequality constraints at stages 1 through $T-1$.

By [3, Corollary 6.15], the set $\operatorname{ri} M_i(\operatorname{dom} h_i)$ on the right-hand side of (13) is identical to $M_i(\operatorname{ri}\operatorname{dom} h_i)$. It is possible to substitute less restrictive assumptions for Assumption 4, but we use the form above to keep the analysis relatively simple.

Proposition 5 *In the context of problem* (10)*, suppose that* $z \in \mathcal{H}_0$ *satisfies*

$$0 \in \sum_{i=1}^{n} L_i^* \partial f_i(L_i z), \tag{14}$$

where the summation sign denotes the Minkowski sum of sets, L_i^* *denotes the adjoint linear operator of* L_i*, and* ∂g *denotes the subgradient mapping of a convex function* g*. Then* z *is an optimal solution of* (10)*. Under Assumption 1 and the setup S1-S4, if one subsequently lets, for all* $i \in 1..n$*,*

$$x_{iT} \in \operatorname*{Arg\,min}_{\xi \in \mathbb{R}^{m_T}} \left\{ h_i(L_i z, \xi) \right\} \qquad \left(\forall\, t \in 1..(T-1)\right)\ x_{it} = z_{it}, \tag{15}$$

then the resulting vector $x \in \mathcal{X}$ *is optimal for* (7)*.*

Conversely, suppose that assumptions 1 and 4 hold, and x *is an optimal solution to* (7)*. Then* $z = Mx$ *must satisfy the optimality condition* (14)*.*

Parts of the proof of this result are somewhat technical, so we defer it to Appendix A. Our algorithm is based on finding a solution to the optimality condition (14), which is always a sufficent condition for optimality; if Assumption 4 holds, it is also necessary.

Algorithm 1 describes an algorithm to solve the monotone inclusion problem (14). Its iterates $\{z^k\}$ are estimates of the primal solution z in (14), while the iterates $\{w^k\} = \left\{(w_1^k, \ldots, w_n^k)\right\}$ are estimates of the dual solution. In this situation, a dual solution consists of vectors $w_1 \in \partial f_1(L_1 z), \ldots, w_n \in \partial f_n(L_n z)$ that verify (14) by having $\sum_{i=1}^n L_i^* w_i = 0$.

Lines 3-11 constitute the "decomposition" portion of the algorithm, which processes the individual functions f_i. Of these, lines 5-9 operate on some subset I_k of the possible indices $i \in 1..n$, solving subproblems to produce for each $i \in I_k$ some pair $(x_i^k, y_i^k) \in \mathcal{H}_i^2$. For $i \notin I_k$, line 11 instead "recycles" the earlier pair $(x_i^k, y_i^k) \in \mathcal{H}_i^2$. Lines 12-20 constitute the "coordination" phase of the algorithm. It uses the pairs $\left\{(x_i^k, y_i^k)\right\}_{i=1}^n$ to attempt to compute improved primal and dual solution estimates $(z^{k+1}, w^{k+1}) = (z^{k+1}, w_1^{k+1}, \ldots, w_n^{k+1})$. Its form, originating in earlier publications such as [1, 8, 12, 14], is based on projecting the current iterate (z^k, w^k) onto a halfspace containing all primal-dual solutions. This halfspace is constructed using the vectors $\left\{(x_i^k, y_i^k)\right\}_{i=1}^n$.

On lines 1 and 12, $\mathcal{W}$ denotes the linear subspace of $\mathcal{H}_1 \times \cdots \times \mathcal{H}_n$ given by

$$\mathcal{W} = \left\{ w = (w_1, \ldots, w_n) \in \mathcal{H}_1 \times \cdots \times \mathcal{H}_n \;\middle|\; \sum_{i=1}^{n} L_i^* w_i = 0 \right\}. \tag{16}$$

1 Start with any $z^0 \in \mathcal{H}_0$, $w^0 \in \mathcal{W}$ and $(x_i^{-1}, y_i^{-1} \in \mathcal{H}_i)_{i \in 1..n}$
2 **for** $k = 0, 1, \ldots$ **do**
3 Select some $I_k \subseteq 1..n$
4 **for** $i \in I_k$ **do**
5 Select some integer $d(i,k) \in 0..k$
6 Select some scalar $\zeta_{ik} > 0$
7 $r_i^k = L_i z^{d(i,k)} + \zeta_{ik} w_i^{d(i,k)}$
8 $x_i^k = \arg\min_{x_i \in \mathcal{H}_i} \left\{ f_i(x_i) + \frac{1}{2\zeta_{ik}} \left\| x_i - r_i^k \right\|_{\mathcal{H}_i}^2 \right\}$
9 $y_i^k = \frac{1}{\zeta_{ik}}(r_i^k - x_i^k)$
10 **end**
11 **for** $i \in 1..n \backslash I_k$ **do** $(x_i^k, y_i^k) = (x_i^{k-1}, y_i^{k-1})$
12 $u^k = \mathrm{proj}_{\mathcal{W}}(x_1^k, \ldots, x_n^k)$
13 $v^k = \sum_{i=1}^n L_i^* y_i^k$
14 $\tau_k = \left\| u^k \right\|^2 + \left\| v^k \right\|^2$
15 **if** $\tau_k > 0$ **then**
16 Choose some $\nu_k : 0 < \nu_k < 2$
17 $\theta_k = \dfrac{\nu_k}{\tau_k} \max\left\{0, \sum_{i=1}^n \langle L_i z^k - x_i^k, y_i^k - w_i^k \rangle_{\mathcal{H}_i}\right\}$
18 **else** $\theta_k = 0$
19 $z^{k+1} = z^k - \theta_k v^k$
20 $w^{k+1} = w^k - \theta_k u^k$
21 **end**

Algorithm 1: Abstract algorithm for problem (14)

On line 12, $\mathrm{proj}_{\mathcal{W}}$ denotes the orthogonal projection map from $\mathcal{Z} = \mathcal{H}_1 \times \cdots \times \mathcal{H}_n$ onto $\mathcal{W}$. In the general case, such a projection might be difficult to implement, in which case techniques exist for avoiding it [8, 23], but in the particular application to stochastic programming developed in this paper, it turns out to be straightforward.

On lines 3 and 5, Algorithm 1 describes the per-iteration parameters I_k and $d(i,k)$ as being explicitly "selected". In an fully asynchronous parallel implementation, these parameters would not be actively chosen but would instead be implicitly determined by the asynchronous execution path. For example, if the solution of a subproblem $\bar{i}$ actually initiated three iterations ago were to be completed and available in iteration k, then I_k would contain $\bar{i}$ and $d(\bar{i}, k)$ would be $k-3$. The computational tests in this paper will consider only block-asynchronous implementations which alternate between non-overlapping decomposition and coordination phases, the only departure from classical decomposition procedures being that the decomposition phase processes only a subset of the possible subproblems, with considerable freedom as to which subproblems to select. In such cases, I_k is indeed directly chosen (we will discuss how to make this choice later), and $d(i,k) = k$ for all $k \geq 0$ and $i \in I_k$.

We now establish a relationship between Algorithm 1 and the monotone inclusion algorithm proposed in [12].

Proposition 6 *For any closed proper convex functions $f_i : \mathcal{H}_i \to \mathbb{R} \cup \{+\infty\}$ and continuous linear maps $L_i : \mathcal{H}_0 \to \mathcal{H}_i$ (for $i \in 1..n$), Algorithm 1 develops the same sequences of iterates as [12, Algorithm 1] with (in the notation of [12]) $\hat{k} = 0$, the mappings T_i set to ∂f_i for all $i \in 1..n$, the parameters $\mu_{i,d(i,k)} = \zeta_{ik}$, for all i and k, and the error parameter e_i^k set to 0 for all $i \in 1..n$ and $k \geq 0$.*

Proof. Taking $\hat{k} = 0$, [12, Algorithm 1] reduces to a method identical to Algorithm 1, except that it has the single assignment

$$(x_i^k, y_i^k) = \mathrm{Prox}_{T_i}^{\mu_{i,d(i,k)}}\Big(L_i z^{d(i,k)} + \mu_{i,d(i,k)} w_i^{d(i,k)} + e_i^k\Big)$$

in place of lines 7-9. Setting $T_i = \partial f_i$ and $e_i^k = 0$, and setting $\mu_{i,d(i,k)} = \zeta_{ik}$ for all i and k, we may rewrite this equation as

$$(x_i^k, y_i^k) = \mathrm{Prox}_{\partial f_i}^{\zeta_{ik}}\Big(L_i z^{d(i,k)} + \zeta_{ik} w_i^{d(i,k)}\Big) = \mathrm{Prox}_{\partial f_i}^{\zeta_{ik}}(r_i^k), \tag{17}$$

where r_i^k is as computed in line 7. In [12], $\mathrm{Prox}_{\partial f}^{\mu}(r)$ is defined as the unique pair (x, y) such that $y \in \partial f(x)$ and $x + \mu y = r$, so (17) is equivalent to the conditions

$$y_i^k \in \partial f_i(x_i^k) \qquad\qquad x_i^k + \zeta_{ik} y_i^k = r_i^k.$$

The vectors x_i^k and y_i^k computed by lines 8 and 9 satisfy precisely these conditions, since lines 8 and 9 implement the standard operations for computing the proximal map of a closed convex function; see for instance [3, Example 23.3 and Definition 12.23]. □

Finally, we restate and slightly extend the convergence result in [12] in the case of Algorithm 1.

Proposition 7 *Suppose that* (14) *has a solution and that the parameter choices in Algorithm 1 conform to the following:*

A1. There exist $\underline{\nu}, \bar{\nu} \in]0, 2[$ such that $\underline{\nu} \leq \nu_k \leq \bar{\nu}$ for all k.

A2. For some scalars $0 < \underline{\zeta} \leq \bar{\zeta}$, we have $\underline{\zeta} \leq \zeta_{ik} \leq \bar{\zeta}$ for all $k \geq 0$ and $i \in i..n$.

A3. For some fixed integer $M \geq 1$, each index i is selected for membership in I_k at least once in every M iterations, that is,

$$(\forall\, j \geq 0) \quad : \quad \left(\bigcup_{k=j}^{j+M-1} I_k\right) = 1..n. \tag{18}$$

A4. For some fixed integer $D \geq 0$, we have $k - d(i,k) \leq D$ for all i, k with $i \in I_k$. That is, there is a constant bound on the extent to which the information $z^{d(i,k)}$ and $w_i^{d(i,k)}$ in line 7 may be "out of date".

Then the sequences developed by Algorithm 1 has the properties below. Here, "$\rightharpoonup$" denotes convergence in the weak topology, which is equivalent to ordinary convergence in finite-dimensional spaces.

1. $z^k \rightharpoonup z^\infty$, *where* z^∞ *is some solution of* (14)
2. $(\forall\, i \in 1..n)\ w_i^k \rightharpoonup w_i^\infty$, *where* $w_i^\infty \in T_i(L_i z^\infty)$, *and* $\sum_{i=1}^n L_i^* w_i^\infty = 0$
3. $(\forall\, i \in 1..n)\ x_i^k \rightharpoonup L_i z^\infty$
4. $(\forall\, i \in 1..n)\ y_i^k \rightharpoonup w_i^\infty$.
5. $v^k \rightharpoonup 0$
6. $u^k \rightharpoonup 0$.

Proof. Proposition 6 asserts that Algorithm 1 generates the same sequence of iterates as Algorithm 1 of [12] with $\hat{k} = 0$, $T_i = \partial f_i$ for all $i \in 1..n$, and $e_i^k = 0$ for all $i \in 1..n$ and $k \geq 0$. Assumption A2 is identical to Assumption 2.1(2) in [12], except that there has been some renumbering between the parameters μ_{ik} of [12] and the ζ_{ik} here. However since both assumptions simply stipulate fixed positive upper and lower bounds on the two parameter sets, the numbering differences are inconsequential and Assumption 2.1(2) of [12] is satisfied. Assumptions A1, A3, and A4 correspond exactly to asssumptions 2.1(1), 2.1(3), and 2.1(4) of [12]. Since $e_i^k = 0$ for all $i \in 1..n$ and $k \geq 0$, part 5 of Assumption 2.1 of [12] is also trivially satisfied. Thus, that entire assumption holds and the sequences developed by Algorithm 1 conform to the conclusions of Proposition 2.3 of [12]. Conclusions 1-4 then follow immediately, and conclusion 5 follows since Algorithm 1 sets $v^k = \sum_{i=1}^n L_i^* y_i^k$ and Proposition 2.3 of [12] asserts that $\sum_{i=1}^n L_i^* y_i^k \rightharpoonup 0$.

Finally, we consider conclusion 6. First, we note that $(L_1 z^\infty, \ldots, L_n z^\infty) \in \mathcal{W}^\perp$, since for any $(w_1, \ldots, w_n) \in \mathcal{W}$ we have

$$\begin{aligned}\big\langle (L_1 z^\infty, \ldots, L_n z^\infty), (w_1, \ldots, w_n) \big\rangle &= \big\langle (z^\infty, \ldots, z^\infty), (L_1^* w_1, \ldots, L_n^* w_n) \big\rangle \\ &= \sum_{i=1}^n \langle z^\infty, L_i^* w_i \rangle_{\mathcal{H}_0} = \Big\langle z^\infty, \sum_{i=1}^n L_i^* w_i \Big\rangle_{\mathcal{H}_0} = \langle z^\infty, 0 \rangle_{\mathcal{H}_0} = 0.\end{aligned}$$

Let $a = (a_1, \ldots, a_n)$ be any element of $\mathcal{H} = \mathcal{H}_1 \times \cdots \times \mathcal{H}_n$. Then

$$\big\langle \mathrm{proj}_{\mathcal{W}}(a), (x_1^k, \ldots, x_n^k) - (L_1 z^\infty, \ldots, L_n z^\infty) \big\rangle \to 0$$

because conclusion 3 asserts that $(x_1^k, \ldots, x_n^k) \rightharpoonup (L_1 z^\infty, \ldots, L_n z^\infty)$. Using that orthogonal projection maps onto linear subspaces are linear and self-adjoint, we then equivalently have

$$\big\langle a, \mathrm{proj}_{\mathcal{W}}(x_1^k, \ldots, x_n^k) - \mathrm{proj}_{\mathcal{W}}(L_1 z^\infty, \ldots, L_n z^\infty) \big\rangle \to 0. \tag{19}$$

Now, Algorithm 1 sets $u^k = \mathrm{proj}_{\mathcal{W}}(x_1^k, \ldots, x_n^k)$, and we have $\mathrm{proj}_{\mathcal{W}}(L_1 z^\infty, \ldots, L_n z^\infty) = 0$ since we have already established that $(L_1 z^\infty, \ldots, L_n z^\infty) \in \mathcal{W}^\perp$. Therefore, (19) is equivalent to $\langle a, u^k \rangle \to 0$. Since the vector a was arbitrary, it follows that $u^k \rightharpoonup 0$ and conclusion 6 is established. □

5 Deriving an Algorithm for Stochastic Programs

We now derive an algorithm for convex stochastic programming by applying Algorithm 1 to the setup described in items S1-S4 of Section 3. In order to apply this setup, we need to determine the form of the adjoint operators L_i^* and the subspace $\mathcal{W}$:

Proposition 8 *Under the choices of $\mathcal{H}_0, \ldots, \mathcal{H}_n$ and $L_1, \ldots, L_n$ specified in S1-S3,*

1. *For all $i \in 1..n$ the adjoint mapping $L_i^* : \mathcal{H}_i \to \mathcal{H}_0 = \mathcal{N}$ is given by*

$$L_i^* : w_i \mapsto \mathrm{proj}_{\mathcal{N}}(0, \ldots, 0, w_i, 0, \ldots, 0), \tag{20}$$

   where w_i appears in the i^{th} position in $(0, \ldots, 0, w_i, 0, \ldots, 0)$ and $\mathrm{proj}_{\mathcal{N}}$ denotes the orthogonal projection map from $\mathcal{Z} = \mathcal{H}_1 \times \cdots \times \mathcal{H}_n$ onto $\mathcal{N}$.

2. *For any vectors $w_1 \in \mathcal{H}_1, \ldots, w_n \in \mathcal{H}_n$, one has $\sum_{i=1}^n L_i^* w_i = \mathrm{proj}_{\mathcal{N}}(w_1, \ldots, w_n)$.*

3. *$\mathcal{W} = \mathcal{N}^\perp$, where $\mathcal{W}$ is the subspace defined in (16).*

Proof. Consider any particular $i \in 1..n$. The adjoint of $L_i : \mathcal{H}_0 \to \mathcal{H}_i$ is the unique linear operator $L_i^* : \mathcal{H}_i \to \mathcal{H}_0$ with the property

$$(\forall\, z \in \mathcal{H}_0)\ (\forall\, w_i \in \mathcal{H}_i) \quad \langle z, L_i^* w_i \rangle_{\mathcal{H}_0} = \langle L_i z, w_i \rangle_{\mathcal{H}_i}.$$

Using S1-S3, the above condition is equivalent to

$$\begin{aligned}(\forall\, z \in \mathcal{N})\ (\forall\, w_i \in \mathcal{Z}_i) \quad \langle z, L_i^* w_i \rangle_{\mathcal{Z}} &= \pi_i(z_i)^\top (w_i) \\ &= \langle z, (0, \ldots, 0, w_i, 0, \ldots, 0) \rangle_{\mathcal{Z}} \\ &= \langle z, \mathrm{proj}_{\mathcal{N}}(0, \ldots, 0, w_i, 0, \ldots, 0) \rangle_{\mathcal{Z}},\end{aligned}$$

where the last equality holds because $z \in \mathcal{N}$. Because $\mathrm{proj}_{\mathcal{N}}(0, \ldots, 0, w_i, 0, \ldots, 0) \in \mathcal{Z}$, we conclude that L_i^* has the form $w_i \mapsto \mathrm{proj}_{\mathcal{N}}(0, \ldots, 0, w_i, 0, \ldots, 0)$. Since $i \in 1..n$ was arbitrary, the first claim is established.

Now take any $w_1 \in \mathcal{H}_1, \ldots, w_n \in \mathcal{H}_n$. Using the first claim and the linearity of the $\mathrm{proj}_{\mathcal{N}}$ operator, we have

$$\begin{aligned}\sum_{i=1}^n L_i^* w_i = \sum_{i=1}^n \mathrm{proj}_{\mathcal{N}}(0, \ldots, 0, w_i, 0, \ldots, 0) &= \mathrm{proj}_{\mathcal{N}}\left(\sum_{i=1}^n (0, \ldots, 0, w_i, 0, \ldots, 0) \right) \\ &= \mathrm{proj}_{\mathcal{N}}(w_1, \ldots, w_n),\end{aligned}$$

and the second claim is established. Combining the definition of $\mathcal{W}$ from (16) and the second claim, we then have

$$\mathcal{W} = \{(w_1, \ldots, w_n) \in \mathcal{Z} \mid \textstyle\sum_{i=1}^n L_i^* w_i = 0\} = \{w \in \mathcal{Z} \mid \mathrm{proj}_{\mathcal{N}}(w) = 0\} = \mathcal{N}^\perp,$$

and the third claim is established. □

It remains to consider how to perform the proximal computation on line 8 of Algorithm 1.

Proposition 9 *Under S1-S4, the computation of x_i^k on line 8 of Algorithm 1 for any $i \in 1..n$ may be implemented by finding*

$$(x_i^k, x_{iT}^k) \in \operatorname*{Arg\,min}_{\substack{x_i \in \mathbb{R}^{m'} \\ x_{iT} \in \mathbb{R}^{m_T}}} \left\{ h_i(x_i, x_{iT}) + \frac{1}{2\zeta_{ik}} \left\| x_i - r_i^k \right\|_2^2 \right\}, \tag{21}$$

where $\|\cdot\|_2$ denotes the usual Cartesian L_2-norm on $\mathbb{R}^{m_T}$. That is, one finds a minimizer of the function in (21) over all $(x_i, x_{iT}) \in \mathbb{R}^{m'} \times \mathbb{R}^{m_T} = \mathbb{R}^m$, and x_i^k consists of the first m' components of the that minimizer (those components not corresponding to the last stage).

Proof. Under S1-S4, line 8 of Algorithm 1 takes the form

$$\begin{aligned} x_i^k &= \arg\min_{x_i \in \mathcal{H}_i} \left\{ f_i(x_i) + \frac{1}{2\zeta_{ik}} \left\| x_i - r_i^k \right\|_{\mathcal{H}_i}^2 \right\} \\ &= \arg\min_{x_i \in \mathbb{R}^{m'}} \left\{ \pi_i \min_{x_{iT} \in \mathbb{R}^{m_T}} \left\{ h_i(x_i, x_{iT}) \right\} + \frac{1}{2\zeta_{ik}} \pi_i \left\| x_i - r_i^k \right\|_2^2 \right\} \\ &= \arg\min_{x_i \in \mathbb{R}^{m'}} \left\{ \min_{x_{iT} \in \mathbb{R}^{m_T}} \left\{ h_i(x_i, x_{iT}) \right\} + \frac{1}{2\zeta_{ik}} \left\| x_i - r_i^k \right\|_2^2 \right\}, \end{aligned}$$

Now, since

$$\begin{aligned} \min_{x_i \in \mathbb{R}^{m'}} \left\{ \min_{x_{iT} \in \mathbb{R}^{m_T}} \left\{ h_i(x_i, x_{iT}) \right\} + \frac{1}{2\zeta_{ik}} \left\| x_i - r_i^k \right\|_2^2 \right\} & \\ = \min_{\substack{x_i \in \mathbb{R}^{m'} \\ x_{iT} \in \mathbb{R}^{m_T}}} \left\{ h_i(x_i, x_{iT}) + \frac{1}{2\zeta_{ik}} \left\| x_i - r_i^k \right\|_2^2 \right\}, & \end{aligned}$$

the claim follows immediately. □

Note that Assumption 1 guarantees that (21) has a solution, since it involves minimizing a lower semicontinuous function with a compact effective domain; see for example the version of the classical Weierstrass theorem in [4, Proposition 2.1.1].

Considering that $r_i^k = L_i z^{d(i,k)} + \zeta_{ik} w_i^{d(i,k)}$ on line 7 of Algorithm 1, and using the form of L_i from S3, the minimand in (21) may be written

$$\begin{aligned} h_i(x_i, x_{iT}) + \frac{1}{2\zeta_{ik}} \left\| x_i - r_i^k \right\|_2^2 &= h_i(x_i, x_{iT}) + \frac{1}{2\zeta_{ik}} \left\| x_i - \left(z_i^{d(i,k)} + \zeta_{ik} w_i^{d(i,k)} \right) \right\|_2^2 \\ &= h_i(x_i, x_{iT}) - \left(w_i^{d(i,k)} \right)^\top (x_i) + \frac{1}{2\zeta_{ik}} \left\| x_i - z_i^{d(i,k)} \right\|_2^2. \end{aligned} \tag{22}$$

Further, the calculation of y_i^k in line 9 of Algorithm 1 may be written

$$y_i^k = \frac{1}{\zeta_{ik}} (r_i^k - x_i^k) = \frac{1}{\zeta_{ik}} (z_i^{d(i,k)} + \zeta_{ik} w_i^{d(i,k)} - x_i^k) = w_i^{d(i,k)} + \frac{1}{\zeta_{ik}} (z_i^{d(i,k)} - x_i^k). \tag{23}$$

1 Start with any $z^0 \in \mathcal{N}$, $w^0 \in \mathcal{N}^\perp$ and $(x_i^{-1}, y_i^{-1} \in \mathbb{R}^{m'})_{i \in 1..n}$
2 **for** $k = 0, 1, \ldots$ **do**
3 Select some $I_k \subseteq 1..n$
4 **for** $i \in I_k$ **do**
5 Select some integer $d(i,k) \in 0..k$
6 Select some scalar $\rho_{ik} > 0$
7 Find $(x_i^k, x_{iT}^k) \in \operatorname{Arg\,min}_{\substack{x_i \in \mathbb{R}^{m'} \\ x_{iT} \in \mathbb{R}^{m_T}}} \left\{ h_i(x_i, x_{iT}) + \left(w_i^{d(i,k)}\right)^\top (x_i) + \frac{\rho_{ik}}{2} \left\| x_i - z_i^{d(i,k)} \right\|_2^2 \right\}$
8 $y_i^k = w_i^{d(i,k)} + \rho_{ik}(x_i^k - z_i^{d(i,k)})$
9 **end**
10 **for** $i \notin I_k$ **do** $(x_i^k, y_i^k) = (x_i^{k-1}, y_i^{k-1})$
11 $u^k = \operatorname{proj}_{\mathcal{N}^\perp}(x_1^k, \ldots, x_n^k)$
12 $v^k = \operatorname{proj}_{\mathcal{N}}(y_1^k, \ldots, y_n^k)$
13 $\tau_k = \sum_{i=1}^n \pi_i \left(\left\| u_i^k \right\|_2^2 + \left\| v_i^k \right\|_2^2 \right)$
14 **if** $\tau_k > 0$ **then**
15 Choose some $\nu_k : 0 < \nu_k < 2$
16 $\theta_k = \dfrac{\nu_k}{\tau_k} \max\left\{0, \sum_{i=1}^n \pi_i (z_i^k - x_i^k)^\top (w_i^k - y_i^k)\right\}$
17 **else** $\theta_k = 0$
18 $z^{k+1} = z^k + \theta_k v^k$
19 $w^{k+1} = w^k + \theta_k u^k$
20 **end**

Algorithm 2: Asynchronous projective hedging (APH) — Algorithm 1 specialized to the setup S1-S4.

If we reverse the signs of the $\{w_i^k\}$ and $\{y_i^k\}$ sequences and replace ζ_{ik} with a reciprocal parameter $\rho_{ik} = 1/\zeta_{ik}$, we conclude that we may implement lines 7-9 of Algorithm 1 through the following calculations:

$$(x_i^k, x_{iT}^k) \in \operatorname*{Arg\,min}_{\substack{x_i \in \mathbb{R}^{m'} \\ x_{iT} \in \mathbb{R}^{m_T}}} \left\{ h_i(x_i, x_{iT}) + \left(w_i^{d(i,k)}\right)^\top (x_i) + \frac{\rho_{ik}}{2} \left\| x_i - z_i^{d(i,k)} \right\|_2^2 \right\} \tag{24}$$

$$y_i^k = w_i^{d(i,k)} + \rho_{ik}(x_i^k - z_i^{d(i,k)}). \tag{25}$$

Other than the potential delay encompassed in the iteration counter $d(i,k)$, the minimization problem (24) is identical to the subproblem solved in PH [31], while (25) resembles the PH Lagrange multiplier update.

Algorithm 2 displays the entire "Asynchronous Projective Hedging" (APH) method one obtains by specializing Algorithm 1, with the signs of $\{w_i^k\}$ and $\{y_i^k\}$ reversed and $\rho_{ik} = 1/\zeta_{ik}$ for all $i \in 1..n$ and $k \geq 1$, to the setup S1-S4. As with Algorithm 1, a fully asynchronous implementation would not explicitly "select" the parameters I_k and $d(i,k)$ on lines 3 and 5; instead, these parameters would be implicitly determined by the execution sequence.

In Algorithm 2, the foregoing discussion establishes that lines 7-8 are equivalent to lines

7-9 of Algorithm 1 under the setup S1-S4, after reversing the signs of $\{w_i^k\}$ and $\{z_i^k\}$ and substituting the parameter $\rho_{ik} = 1/\zeta_{ik}$ for ζ_{ik}. Proposition 8 guarantees that lines 11 and 12 are respectively equivalent to lines 12 and 13 of Algorithm 1, although the sign of v^k is reversed from Algorithm 1 due to the reversed signs of the y_i^k. Line 13 computes the squared length of the hyperplane normal $(-v^k, u^k)$ in the product norm induced by the norm $\langle \cdot\,, \cdot \rangle_{\mathcal{Z}}$ from (4) chosen for $\mathcal{H}_0 = \mathcal{N}$ and $\mathcal{W} = \mathcal{N}^\perp$. Line 16 is then equivalent to line 17 of Algorithm 1, considering the the choice of the inner products on $\mathcal{H}_1, \ldots, \mathcal{H}_n$ in S2 and the sign reversals for $\{w_i^k\}$ and $\{z_i^k\}$. Finally, line 18 is equivalent to the z update in line 19 of Algorithm 1 considering the sign reversal in v^k induced by the sign reversal of the $\{y_i^k\}$, while line 19 is equivalent to the w update in line 20 of Algorithm 1 when considering the sign reversal of the $\{w^k\}$ sequence.

To implement the algorithm, we need to perform the projections onto $\mathcal{N}^\perp$ and $\mathcal{N}$ specified on lines 11 and 12. Due to the probability-weighted form of the inner product $\langle \cdot\,, \cdot \rangle_{\mathcal{Z}}$, the projection $\mathrm{proj}_{\mathcal{N}}$ consists of the same probability-weighted tree averaging operation as in the originally proposed form of the PH algorithm [31]. Since $\mathrm{proj}_{\mathcal{N}^\perp} = \mathrm{Id} - \mathrm{proj}_{\mathcal{N}}$, where "Id" represents the identity map, the complementary projection onto $\mathcal{N}^\perp$ involves a nearly identical calculation to projection onto $\mathcal{N}$. Specifically, we may implement line 12 by

$$(\forall\, t \in 1..T-1)\ (\forall\, N \in \mathcal{T}_t): \qquad \text{compute } \bar{y}(N) = \frac{1}{\pi(N)} \sum_{i \in \mathcal{S}(N)} \pi_i y_{it}^k, \text{ then } (\forall\, i \in \mathcal{S}(N))\ v_{it}^k = \bar{y}(N), \quad (26)$$

whereas line 11 may be implemented by

$$(\forall\, t \in 1..T-1)\ (\forall\, N \in \mathcal{T}_t): \qquad \text{compute } \bar{x}(N) = \frac{1}{\pi(N)} \sum_{i \in \mathcal{S}(N)} \pi_i x_{it}^k, \text{ then } (\forall\, i \in \mathcal{S}(n))\ u_{it}^k = x_{it}^k - \bar{x}(N). \quad (27)$$

Proposition 10 *Suppose that a stochastic programming problem formulated as* (7) *has a solution and Assumption 4 holds. Suppose that we operate Algorithm 2 in accordance with the following conditions:*

1. *There exist* $\underline{\nu}, \bar{\nu} \in]0, 2[$ *such that* $\underline{\nu} \leq \nu_k \leq \overline{\nu}$ *for all* k.
2. *For some scalars* $0 < \underline{\rho} \leq \bar{\rho}$, *we have* $\underline{\rho} \leq \rho_{ik} \leq \bar{\rho}$ *for all* $k \geq 0$ *and* $i \in i..n$.
3. *For some fixed integer* $M \geq 1$, $\left(\bigcup_{k=j}^{j+M-1} I_k\right) = 1..n$ *for all* $j > 0$.
4. *For some fixed integer* $D \geq 0$, *we have* $k - d(i,k) \leq D$ *for all* i, k *with* $i \in I_k$.

Then

$$z^k \to x^\infty \qquad x^k \to x^\infty \qquad v^k \to 0 \qquad u^k \to 0, \quad (28)$$

where $x^\infty = M\bar{x}^*$ *for some optimal solution* $\bar{x}^* \in \mathbb{R}^m$ *of* (7), *that is,* x^∞ *is some optimal solution to* (7) *with the last-stage variables deleted.*

For each $i \in 1..n$ let $\mathcal{I}_i = \{k \geq 0 \mid i \in I_k\}$, which is infinitely large by assumption 3, and let $\{x_{iT}^k\}$ be any limit point of the sequence $\{x_{iT}^k\}_{i \in \mathcal{I}_i}$, where x_{iT}^k is the last-stage solution vector found in line 7 of Algorithm 2. Then the vector

$$\bar{x}^\infty = \left((x_i^\infty, x_{iT}^\infty)\right)_{i=1}^n = \left((x_1^\infty, x_{1T}^\infty), \ldots, (x_n^\infty, x_{nT}^\infty)\right)$$

is an optimal solution of (7).

Proof. First, Proposition 3 states that any solution $\bar{x}^*$ of (7) has a corresponding solution $z^* = Mx^*$ of (10). Since the Assumption 4 is in force, the second part of Proposition 5 implies that z^* must be a solution to the inclusion (14). Therefore a solution to (14) exists.

We have already established that Algorithm 2 is a special case of Algorithm 1 under the setup S1-S4, with the signs of $\{w_i^k\}$ and $\{y_i^k\}$ reversed and $\rho_{ik} = 1/\zeta_{ik}$. Setting $\underline{\zeta} = 1/\overline{\rho}$ and $\overline{\zeta} = 1/\underline{\rho}$, condition 2 here is equivalent to condition A2 of Proposition 7 if $\rho_{ik} = 1/\zeta_{ik}$ for all $i \in 1..n$ and $k \geq 0$. The remaining conditions here are identical to those of Proposition 7, so all of the assumptions of Proposition 7 hold. That proposition then immediately asserts that $z^k \rightharpoonup z^\infty$, $x^k \rightharpoonup z^\infty$, $v^k \rightharpoonup 0$, and $u^k \rightharpoonup 0$, where z^∞ is some solution of (14). By Proposition 5, such a point is necessarily a solution to (10), and so Proposition 3 guarantees it is of the form $M\bar{x}^*$ for some optimal solution $\bar{x}^*$ to (7). Since all the spaces are finite dimensional, weak and ordinary convergence are equivalent, and so we have established (28).

To establish the last claim, let $\bar{x}^* = \left((z_1^\infty, x_{1t}^*), \ldots, (z_n^\infty, x_{nT}^*)\right)$ be an optimal solution of (7) such that $z^\infty = M\bar{x}^*$. Consider any $k \geq 0$ and $i \in I_k$. By the calculation in line 7, we must have for all $k \geq 0$ and $i \in I_k$ that

$$h_i(x_i^k, x_{iT}^k) + \left(w_i^{d(i,k)}\right)^\top (x_i^k) + \frac{\rho_{ik}}{2} \left\| x_i - z_i^{d(i,k)} \right\|_2^2 \leq h_i(z_i^\infty, x_{iT}^*) + \left(w_i^{d(i,k)}\right)^\top (z_i^\infty) + \frac{\rho_{ik}}{2} \left\| z_i^\infty - z_i^{d(i,k)} \right\|_2^2$$

Under assumption 4, $d(i,k) \to \infty$ as $k \to \infty$ for all $i \in 1..n$, and from Proposition 7 we know that $\{w^k\}$ converges to some $w^\infty \in \mathcal{W}$. Let $\mathcal{K}_i \subseteq \mathcal{I}_i$ denote an infinite set of indices such that $\lim_{k \to \infty, k \in \mathcal{K}_i} \{x_{iT}^k\} = x_{iT}^\infty$. Taking limits over $k \in \mathcal{K}_i$ in the above inequality and using that $\{x^k\}$ and $\{z^k\}$ both converge to the same limit z^∞, we conclude that

$$\limsup_{\substack{k \to \infty \\ k \in \mathcal{K}_i}} \{h(x_i^k, x_{iT}^k)\} + (w_i^\infty)^\top (z_i^\infty) \leq h_i(z_i^\infty, x_{iT}^*) + (w_i^\infty)^\top (z_i^\infty)$$

Canceling the identical terms $(w_i^\infty)^\top (z_i^\infty)$ and using that f_i is a lower semicontinuous function, we then have that

$$h_i(z_i^\infty, x_{iT}^\infty) \leq \liminf_{\substack{k \to \infty \\ k \in \mathcal{K}_i}} \{h(x_i^k, x_{iT}^k)\} \leq \limsup_{\substack{k \to \infty \\ k \in \mathcal{K}_i}} \{h(x_i^k, x_{iT}^k)\} \leq h_i(z_i^\infty, x_{iT}^*).$$

Multiplying by π_i and summing over i, we obtain that

$$\sum_{i=1}^n \pi_i h_i(z_i^\infty, x_{iT}^\infty) \leq \sum_{i=1}^n \pi_i h_i(z_i^\infty, x_{iT}^*) = \sum_{i=1}^n \pi_i h_i(\bar{x}_i^*)$$

and thus that $\bar{x}^\infty$ is an optimal solution to (7). □

It is also possible to show that the common limit w^∞ of $\{w^k\}$ and $\{y^k\}$ is an appropriately defined dual solution of (7), but we omit this result as it is not required for our subsequent analysis and experiments.

The following result shows that if the algorithm encounters the condition $\tau_k = 0$ encountered in line 14, then it has reached exact convergence.

Proposition 11 *In Algorithm 2, if $u^k = 0$ and $v^k = 0$ (or equivalently $\tau_k = 0$) for any k sufficiently large that $\bigcup_{\ell=0}^{k} I_\ell = 1..n$, then $x^k = (x_1^k, \ldots, x_n^k)$ is an optimal solution of the problem formulation* (10).

Proof. If $\bigcup_{\ell=0}^{k} I_\ell = 1..n$, every subproblem has been processed at least once, and we have $y_i^k \in \partial f_i(L_i x^k) = \partial f_i(x_i^k)$ for all $i \in 1..n$. Since $u^k = \text{proj}_{\mathcal{N}^\perp}(x^k)$ from line 11 of the algorithm, the hypothesis $u^k = 0$ implies that $x^k \in \mathcal{N}$. From line 12 and Proposition 8(2-3), we have $v^k = \text{proj}_{\mathcal{N}}(y_1^k, \ldots, y_n^k) = \sum_{i=1}^n L_i^* y_i^k$, so from the hypothesis $v^k = 0$ we obtain

$$0 = \sum_{i=1}^{n} L_i^* y_i^k \in \sum_{i=1}^{n} L_i^* \partial f_i(L_i x^k),$$

where the inclusion follows because we have already established that $y_i^k \in \partial f_i(L_i x^k)$ for all $i \in 1..n$. Therefore the sufficient optimality condition (14) holds and x is an optimal solution of (10). □

In practice, we have never observed the condition $u^k = 0$ and $v^k = 0$ to hold exactly; however, since both $u^k \to 0$ and $v^k \to 0$ as established in Proposition 10, we can use small values of $\|u^k\|$ and $\|v^k\|$ as numerical termination conditions for the algorithm.

As a last remark, we note that it would also be possible derive an asynchronous decomposition method for stochastic programming from the second, ADMM-like algorithm in [12], instead of directly from its abstract monotone inclusion algorithm. In that case, however, one would have to represent the nonanticipativity constraints as $\Theta(mn)$ linear constraints (recall that m is the number of decision variables per scenario). Applying the second algorithm in [12], each of the vectors playing the role of w_i^k and y_i^k (along with some other variables) in the resulting method would all have dimension on the order of the number of constraints, that is $\Theta(mn)$. Since there are n possible values of i, the working storage requirements of the resulting algorithm would be $\Theta(mn^2)$, as opposed to $\Theta(mn)$ for Algorithm 2. We briefly considered whether the resulting methods might be equivalent to some more memory-efficient algorithm, but did not find such a simplification. Therefore, there does not seem to be any simpler path to deriving a progressive-hedging-like algorithm from the operator splitting methods in [8, 12].

6 Including a Scaling Parameter

Consider two optimization problems $\mathcal{P}$ and $\tilde{\mathcal{P}}$, identical except that the objective function of $\tilde{\mathcal{P}}$ is obtained by multiplying $\mathcal{P}$'s objective function by some positive constant C. The PH algorithm has a property we may call *parametric scale invariance*, as follows: if we apply PH to $\mathcal{P}$ and $\tilde{\mathcal{P}}$ with respective penalty parameters ρ and $C\rho$ and initialize the second

algorithm at $(\tilde{z}^0, \tilde{w}^0) = (z^0, Cw^0)$, it will generate respective sequences of iterates $\{(z^k, w^k)\}$ and $\{(\tilde{z}^k, \tilde{w}^k)\}$ that are equivalent in the sense that $(\tilde{z}^k, \tilde{w}^k) = (z^k, Cw^k)$ will hold for all $k \geq 0$. Essentially, the two instances of PH will follow the same path, except that the dual iterates for $\tilde{\mathcal{P}}$ will be scaled by C. Thus, the penalty parameter ρ can be used to compensate for the relative scaling of the primal and dual variables.

Unfortunately, the APH algorithm as we have stated it so far lacks this desirable property. If we initialize $(\tilde{z}^0, \tilde{w}^0) = (z^0, Cw^0)$, multiply all the proximal parameters for the $\tilde{\mathcal{P}}$ instance by C, and keep all the other parameters the same, running APH on $\mathcal{P}$ and $\tilde{\mathcal{P}}$ will not generate equivalent paths. The reason is that the projection operations in APH minimize the distance to a separating hyperplane in the primal-dual space of iterates (z^k, w^k). If the dual w^k iterates become larger (as must eventually happen if $C > 1$), then they will become a larger fraction of the squared distance to each separating hyperplane, so APH will emphasize dual convergence over primal convergence for sufficiently large C. Conversely, for C sufficiently small, the primal iterates z^k will start to dominate the distance to the separating hyperplane and APH will emphasize primal convergence over dual convergence.

In projective splitting algorithms, one can compensate for such effects by using a scaled norm for the projections, as discussed in [23]. In the case of the stochastic programming setup developed in Section 3, this means selecting a scalar parameter $\gamma > 0$ and using the following inner product on the space $\mathcal{H}_0 \times \mathcal{H}_1 \times \cdots \times \mathcal{H}_n = \mathcal{N} \times \mathcal{Z}$:

$$\big\langle (z, w), (z', w') \big\rangle_\gamma = \sum_{i=1}^{n} \pi_i \cdot \Big(\gamma \langle z_i, z_i' \rangle + \langle w_i, w_i' \rangle \Big). \tag{29}$$

Essentially, all the primal components of the inner product are multiplied by γ. Using the norm induced by this inner product to perform the hyperplane projections results in the following two changes to Algorithm 2:

Line 13 becomes:	$\tau_k = \sum_{i=1}^n \pi_i \Big(\big\|u_i^k\big\|_2^2 + \gamma^{-1} \big\|v_i^k\big\|_2^2 \Big)$
Line 18 becomes:	$z^{k+1} = z^k + \gamma^{-1} \theta_k v^k.$

Otherwise, Algorithm 2 remains unchanged.

These modifications make it possible to recover a form of parameterized scale invariance: for two hypothetical problems $\mathcal{P}$ and $\tilde{\mathcal{P}}$ as described above, if we initialize $(\tilde{z}^0, \tilde{w}^0) = (z^0, Cw^0)$, multiply all the proximal parameters for the $\tilde{\mathcal{P}}$ instance by C, multiply its value of γ by C, and leave all the other algorithm parameters unchanged, we will recover the path equivalence that $(\tilde{z}^k, \tilde{w}^k) = (z^k, Cw^k)$ for all $k \geq 0$. More crucially, the parameter γ allows the user to adjust the APH algorithm's relative emphasis on primal and dual convergence: increasing γ shifts emphasis towards primal convergence, and decreasing γ shifts emphasis toward dual convergence.

For the problem instances used in the experiments of the following section, we found APH to exhibit good practical performance with $\gamma = 1$. For other problem instances, however, using $\gamma \neq 1$ may be critical to the performance of APH.

7 Computational Experiments

We implemented the APH method as stated in Algorithm 2 within the `mpi-sppy` stochastic programming environment [25], in which models are expressed using the Python-based `Pyomo` optimization modeling system [7, 20]. PH as implemented in `mpi-sppy` can be applied to general multi-stage stochastic programming problems, including those with nonconvex and mixed-integer attributes. Our APH implementation uses the same infrastructure as the PH implementation, and may therefore be applied to nonconvex and integer problems. However, the tests described in this paper only consider convex problems, in keeping with the convergence theory developed herein. Also mirroring the PH implementation in `mpi-sppy`, our APH implementation supports a mechanism known as *bundling* — see for example [11, 37] — in which each subproblem is a "bundle" composed of multiple stochastic programming scenarios. Each individual bundle is an extensive form of a smaller stochastic programming problem, and the bundles are tied together with nonanticipativity constraints. The bundles themselves form a coarser-grain tree which may be called a "bundle tree". When running either PH or APH with bundling, each "scenario" in the algorithm statement is replaced by an extensive form bundle of multiple scenarios and the scenario tree is replaced by the bundle tree, but the algorithm is otherwise unchanged.

`Mpi-sspy` uses the standard MPI message-passing library [18, 35] to orchestrate parallelism. In MPI, the compute platform is divided into *ranks*, each of which consists of a single memory space shared by one or more processor cores. MPI provides messaging mechanisms to communicate between different ranks. Each rank may have multiple compute threads communicating with one another through the rank's shared memory space, using whatever multithreading capabilities are available on the host computing platform.

Modern HPC computers typically consist of a potentially large collection of *compute nodes* connected by a high-speed, low-latency network. Each node typically has a RAM memory bank and one or two CPU chips, each containing multiple CPU cores (usually between 12 and 64). Each of these cores may access all memory on the node, while off-node communication entails sending messages through the network. If each node has c_{n} processor cores, then each MPI rank may have a number of allocated cores c_{r} falling between 1 and c_{n}, and $\lfloor c_{\mathrm{n}}/c_{\mathrm{r}} \rfloor$ MPI ranks are hosted by each compute node.

In order to be scalable, the PH implementation in `mpi-sppy` does not employ a "central" or "master" rank. Every rank is allocated a fixed set of subproblems, each of which may consist of single scenario or a single bundle of multiple scenarios. At each PH iteration, each rank solves a quadratically perturbed version of every subproblem assigned to it, and then executes the PH coordination procedure, which involves averaging decision variables at each node of the scenario tree, followed by some simple vector calculations. The vectors involved in these operations are distributed over multiple ranks, with only the averaging operations requiring communication between processors. This averaging is implemented with `MPI_Allreduce` collective operations with an optimal complexity of $V \log r$, where r is the number of MPI ranks involved and V is the length of the vector being averaged, which is equal to the dimension of the individual subspaces $\mathcal{Z}_i$ introduced in Section 3. For large problem instances, this scalability makes it possible to efficiently use a large number of ranks r and a proportionately large number of processor cores. For some further information on the PH implementation in `mpi-sppy`, see [25].

The APH implementation is similar to the PH implementation, with some key differences. Each rank stores the vectors x_i^k, x_{iT}^k, y_i^k, u_i^k, v_i^k, z_i^k and w_i^k for all its assigned subproblem indices i; this setup is similar to that for PH, except that each rank stores several additional vectors. `MPI_Allreduce` operations implement the averaging operations (26)-(27) required for the projections computed in lines 11 and 12 of Algorithm 2. These calculations are essentially the same as PH averaging operations, but involve twice as much data since both u^k and v^k need to be calculated. `MPI_Allreduce` operations also implement the summations on lines 13 and 16 of Algorithm 2, but only two scalar values need to be reduced. All remaining operations in the coordination procedure comprising lines 11-19 of Algorithm 2 are implemented through simple vector calculations localized to each subproblem and thus local to each rank, or very simple, constant-time scalar calculations replicated across all ranks. In summary, although slightly more complicated than for PH, our APH coordination procedure similarly requires only `MPI_Allreduce` calls in conjunction with local scalar and vector operations.

The key difference between the PH and APH implementations is in the processing of subproblems. While we have begun developing mechanisms by which APH can operate in a fully asynchronous manner, the experiments here only investigate block-asynchronous operation. In block-asynchronous operation, $d(i,k) = k$ for all i and k, meaning that each iteration selects some set of subproblems $|I_k|$, solves them, and then proceeds to the coordination step. If one were to take $I_k = 1..n$ at each iteration k, this mode of operation would be fully synchronous, just like the implementation of PH in [25]. Our particular APH implementation, however, only requires $I_k = 1..n$ in the first two iterations ($k = 0, 1$).[1] Subsequently, it may be configured to solve a smaller set of subproblems at each iteration, determined by each rank selecting `dispatch` of its allocated subproblems, where `dispatch` is a configurable parameter in the range $1..(n/r)$. Thus, $|I_k| = r \cdot$ `dispatch` for $k \geq 1$. For the problems classes we tested, we found that there was very little variance in the time required to solve subproblems, hence little reason to investigate alternatives to block asynchronism: if each rank starts solving the same number `dispatch` of subproblems at the same time, then all ranks will finish solving them at about the same time, and enter the coordination phase of the algorithm approximately simultaneously. With more variable subproblem solution times, e.g., in mixed-integer programming models, the situation could be different.

When $n/r > 1$ (i.e., there is more than one subproblem per rank), so that it is possible to have `dispatch` $< n/r$, our implementation can perform significantly less computation per iteration than PH, which carries benefit when not offset by other factors. To select the subproblems for dispatch, our implementation chooses subproblems by giving priority to subproblems i with the most negative values of $\phi_i \doteq \pi_i \sum_{t=1}^{T-1} (z_{it} - x_{it})^\top (w_{it} - y_{it})$; a rationale for this heuristic is given in [23, Section 4.3]. If there are fewer than `dispatch` indices i for which $\phi_i < 0$, then the remaining subproblems are chosen randomly. Furthermore, for some adjustable parameter $\bar{M}$, we give the highest priority to any subproblem not processed within the last $\bar{M}$, iterations, regardless of the value of ϕ_i. Specifically, we prioritize subproblems in lexically increasing order of $\big(\min\{0, \bar{M} - b_i\}, \min\{0, \phi_i\}\big)$, where b_i is the number

[1] We set $I_0 = 1..n$ to assure that the projections in the algorithm are valid from the outset; otherwise the method may not start converging toward the solution until after M iterations. We set $I_1 = 1...n$ due to technical considerations within the implementation.

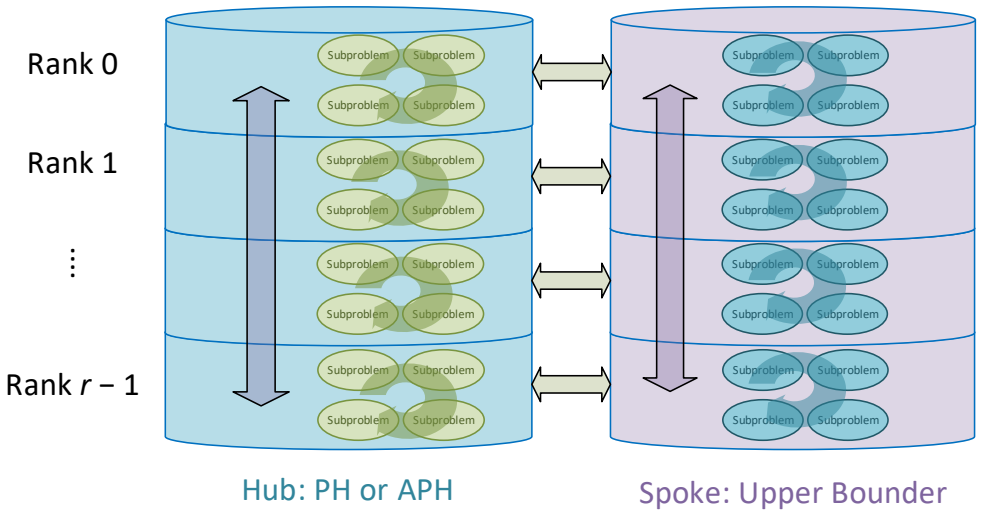

Figure 1: Hub-and-spoke architecture of the `mpi-sppy`-based computational experiments.

of iteration since subproblem i was last processed, this rule guarantees that the dispatch rule meets (18) with $M = \bar{M} + \lceil (n/r)/\texttt{dispatch} \rceil$. We set $\bar{M} = 99$ in our experiments, so it is present largely for theoretical purposes.

In addition to running the PH or APH algorithm, our experiments repeatedly compute feasible solutions that provide an upper bound on the true optimal solution value. Such feasible solutions are not automatically provided in the course of executing either PH or APH, i.e., implementable solutions are not a by-product of either algorithm prior to convergence. In APH, the reason is that while the x_i^k vectors are feasible for the respective subproblems and the z^k vectors satisfy nonanticipativity, only in the limit when the x_i^k and z_i^k become equal do these conditions become jointly satisfied. By using the "hub-and-spoke" architecture described in [25], the implementation calculates upper bounds in a manner that imposes a minimal performance penalty on the core PH and APH algorithms. In short, the r ranks executing PH or APH form a single "hub cylinder" that executes the core solution algorithm (options other than PH and APH are also available, but are not applicable to as broad a range of problems). Any number of additional "spoke" cylinders, each consisting of r MPI ranks, may be configured to compute accompanying upper bounds "seeded" by the core algorithm. In the experiments here, there is only one spoke cylinder, which executes one of the upper bounding heuristics from [25]; Figure 1 depicts this configuration. The spoke cylinders map subproblems to their MPI ranks in the same manner as the hub cylinder, and each communication between cylinders takes place through a bank of parallel messages, one message per pair of corresponding ranks. These messages are depicted by the horizontal arrows in Figure 1. Communication within each cylinder uses MPI collective operations, typically `MPI_Allreduce`; this communication is depicted by the vertical arrows in Figure 1. Since neither PH nor APH automatically provide lower bounds on the optimal solution value, lower bounders are also available and operate similarly.

The upper bounding heuristic we selected from [25] operates as follows: periodically, the hub cylinder sends the approximate solution vector $x^k = (x_1^k, \ldots, x_n^k)$ to the spoke cylinder to "seed" the heuristic. In a two-stage problem, the spoke selects a subproblem $i \in 1..n$

by cycling through a randomly generated list and uses i's first-stage variable values for all subproblems, reoptimizing the second-stage variables for all remaining subproblems. If the result is feasible for all subproblems and improves upon the current upper bound, it becomes the new upper bound. When there are more than two stages, the heuristic applies the same procedure recursively, passing from the scenario tree root to the leaves. The procedure repeats indefinitely with cyclically varying subproblem choices, incorporating new x^k vectors whenever they arrive from the hub.

All computational experiments were carried out on the `quartz` high-performance computing (HPC) cluster at Lawrence Livermore National Laboratory. This cluster comprises 3004 compute nodes, each with 128GB of RAM and two 18-core 2.1GHz Intel Xeon E5-2695 v4 processors, resulting in $2 \cdot 18 = 36$ CPU cores per compute node; the nodes communicate through an Omni-Path interconnect.

Our APH implementation solves each subproblem with Gurobi 9.5.0 [19], via `Pyomo`'s "persistent" Gurobi interface. The persistent interface maintains the subproblem optimization models in RAM, with only the objective function changing between iterations in PH or APH. We used the `Cython` extensions in `Pyomo` to accelerate model construction and manipulation. In all APH experiments, we set $\gamma = \nu = 1$.

7.1 Results: `ssn`

Our first set of experiments consider a standard stochastic programming test problem family known as `ssn` [33, 34], which has 89 first-stage (nonanticipative) variables and second-stage data defined on a discrete probability space that is too large to avoid scenario sampling. Furthermore, hundreds of thousands of scenarios are required to fully stabilize the optimal objective function value. The `ssn` problem is well scaled, with an optimal objective function value on the order of 10. Thus, we used proximal parameter values of $\rho_{ik} \equiv 1$ (or just $\rho = 1$ for PH) in all the associated experiments. We note that improved performance can be obtained by adjusting proximal parameter values in PH and APH, and the γ and ν APH parameters. Our goal in these experiments was not to demonstrate the fastest method of solving the `ssn` problem; in this regard, there is little hope of competing with the specialized algorithm and implementation by Sen and Liu [34] on the problem sizes that they attempted. Their implementation uses a compiled language, whereas our PH and APH implementations use Python and accommodate a much broader range of problems. Comparison is further complicated because the algorithm of Sen and Liu does not simply solve the problem for one set of sampled scenarios, but sequentially solves problems of increasing size until a stopping rule based on confidence intervals is satisfied. The principal aim of the experiments here is to compare APH with PH, but we did test `ssn` instances that are over two orders of magnitude larger than those considered in [34]. Parallel computing has previously been applied to `ssn` in [26], using 100 CPUs to process 5,000 scenarios through the sample average approximation (SAA) method [24]; the instances we consider here range from 4 to 200 times larger and our largest runs used 2,400 processor cores.

For `ssn`, we observed that the barrier method is the preferred subproblem solver strategy. Empirically, we found that the Gurobi barrier method did not accelerate significantly once more than 6 processor cores are assigned to individual bundle subproblems, so we allocated $c_\mathrm{r} = 6$ cores per MPI rank. Although a compute node could in principle host $\lfloor c_\mathrm{n}/c_\mathrm{r} \rfloor =$

$\lfloor 36/6 \rfloor = 6$ ranks, we observed poor performance in this configuration, which we suspect is due to memory contention and cache behavior: the Gurobi barrier solver likely has a very different memory usage than the discretized partial differential equation problems that were the main design target for `quartz`. We observed much better performance when allocating 4 MPI ranks of 6 cores each to every compute node, using only $4 \cdot 6 = 24$ of the 36 possible cores. Henceforth, we simply treat each compute node as having 24 available cores.

The lower bounding spokes currently available in `mpi-sppy` are not able to compute tight lower bounds for `ssn`, so the runs consist of a hub cylinder and single spoke cylinder to compute upper bounds.

7.1.1 Unlimited Compute Resources: One Rank per Subproblem

First, we considered the performance of APH and PH without an effective limit on the available MPI ranks, allocating one 6-core MPI rank per subproblem. Our smallest `ssn` instances comprise 20,000 scenarios, which we formed into bundles of 1000 scenario apiece. With six 6 CPU cores and Gurobi threads per rank as described above, the hub-and-spoke cylinders each consisted of 20 MPI ranks, for a total of $2 \cdot 10 = 40$ ranks using a total of $6 \cdot 40 = 240$ cores and spread across $10 = 240/24$ compute nodes. The individual subproblems are time consuming to solve, typically requiring between 20 and 25 seconds apiece.

Figure 2 displays computational results for three 20,000-scenario `ssn` instances. As in subsequent experiments, the differences between the instances arise from varying random sampling of scenarios. The horizontal axis of each graph shows wall-clock time, while the vertical axis shows the objective value of the best fully feasible (implementable) solution found by the upper-bounding spoke. Both PH and APH require about 200 seconds of wall clock time to report an upper bound, which includes job startup on the HPC cluster, `Pyomo` model instantiation for the scenarios in each bundle, transfer of the `Pyomo model` to the Gurobi in-memory interface, and a few iterations before the upper bound spoke finds a feasible incumbent. We observe in all three instances that APH outperforms PH, albeit by a small absolute margin. Figure 3 displays the same information for three larger, 100,000-scenario `ssn` instances, with each cylinder again having one 6-core MPI rank per subproblem. The hub-and-spoke cylinders now each consist of 100,000/1,000 = 100 MPI ranks, yielding a total of $2 \cdot 100 = 200$ MPI ranks and $6 \cdot 200 = 1{,}200$ cores spread across $1{,}200/24 = 50$ compute nodes. Overall, the relative behavior of PH and APH remains qualitatively similar to the 20,000-scenario `ssn` instances.

7.1.2 Limited Compute Resources: Five Subproblems per Rank

Next, we present experiments in which the ratio of CPU cores and MPI ranks to subproblems was lower; simulating a situation with limited compute resources per subproblem. In this situation, multiple subproblems are assigned to a single MPI rank, with each rank being able to solve at most one subproblem at a time. In particular, we consider taking the same `ssn` problem instances as in the previous subsection, with the same bundles, but allocating 5 subproblems to each rank. Thus, each run uses 20% as many CPU cores and compute nodes as in Section 7.1.1, but more memory per compute node. In this new setting, the 20,000-scenario, 20-bundle runs used only $20/5 = 4$ ranks per cylinder, hence $2 \cdot 4 = 8$

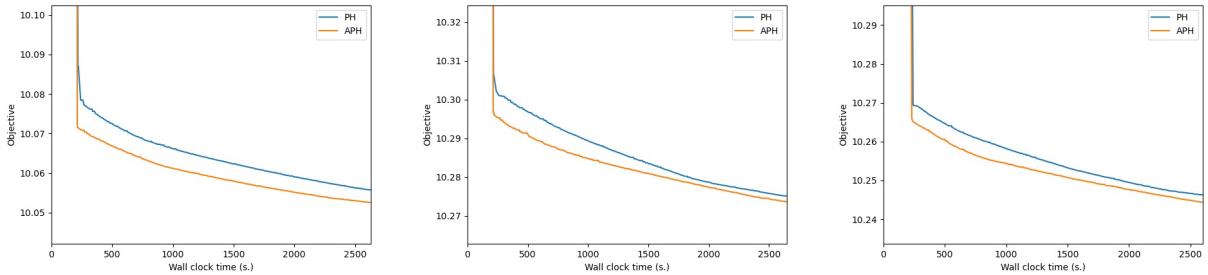

Figure 2: Results for 3 randomized 20K-scenario instances of `ssn`, with one subproblem per MPI rank. In all figures, the reported wall clock time is on the LLNL `quartz` HPC cluster.

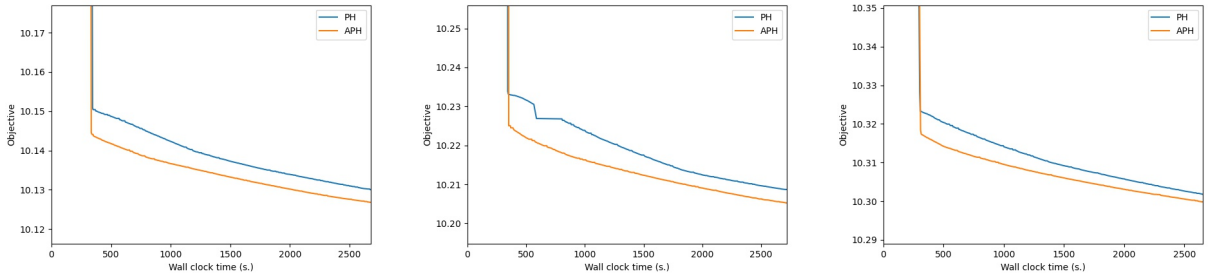

Figure 3: Results for 3 randomized 100K-scenario instances of `ssn`, with one subproblem per MPI rank.

total ranks and $8 \cdot 6 = 48$ cores, thus fitting on $48/24 = 2$ compute nodes. The 100,000-scenario, 100-bundle runs employed $100/5 = 20$ ranks per cylinder, hence $2 \cdot 20 = 40$ total ranks and $40 \cdot 6 = 240$ cores, thus fitting on $240/24 = 10$ compute nodes. Results for the 20,000-scenario and 100,000 scenario instances are shown in Figures 4 and 5, respectively. In APH, we set `dispatch` $= 1$ (i.e., each rank processes only one subproblem per iteration). Initialization time scales essentially linearly, with each rank having 5 times as much work, so the start-up takes 5 times as long, roughly 1,000 seconds. After start-up, APH performs significantly better than PH relative to results reported in Section 7.1.1, by a wider absolute margin. Essentially, the PH algorithm slows down by a factor of 5 when it has a factor of 5 fewer CPUcores, but APH does not slow down as much: in the presented case, solving just one subproblem per iteration means that the iterations are about 5 times faster than when using PH. This improvement in iteration speed appears not to be proportionately offset by deterioration in the speed of convergence, making APH more efficient overall.

7.1.3 Million-Scenario SSN with Limited Compute Resources

As a last excercise with `ssn`, we report in Figure 6 the results for a single instance of `ssn` with 1 million scenarios. We again formed bundles of 1,000 scenarios, resulting in $1{,}000{,}000/1{,}000 = 1{,}000$ bundles, and assigned 5 bundles per MPI rank. Hence, each cylinder has $1{,}000/5 = 200$ MPI ranks. Since there are two cylinders, the configuration used $2 \cdot 200 = 400$ MPI ranks comprising 2,400 CPU cores and $2{,}400/24 = 100$ compute notes. Due to resource limitations, we considered only one problem instance rather than three repli-

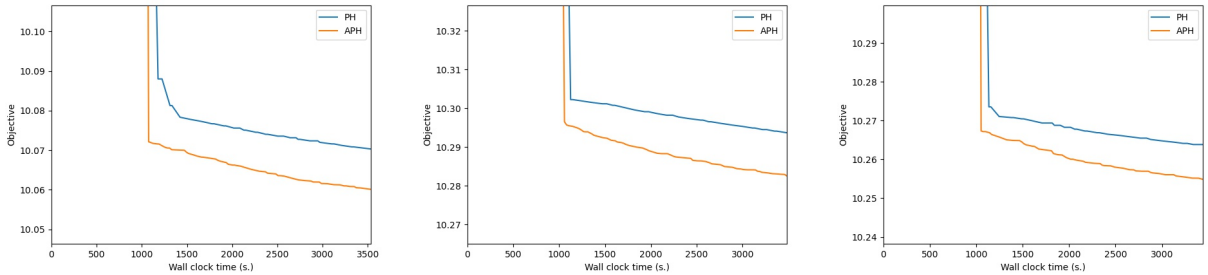

Figure 4: Results for the same 3 randomized 20K-scenario instances of `ssn`, with 5 subproblems per rank and thus using 20% as many CPU cores as the corresponding "unlimited" experiment in Figure 4.

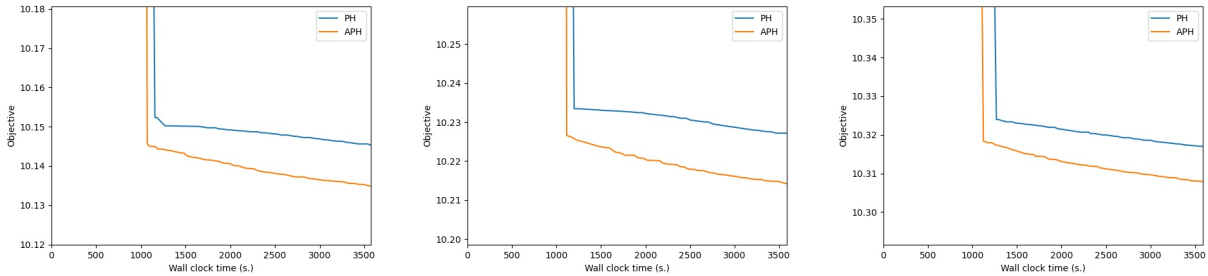

Figure 5: Results for the same 3 randomized 100K-scenario instances of `ssn`, with 5 subproblems per rank and thus using 20% as many CPU cores as the corresponding "unlimited" experiment in Figure 4.

cates. The results appear very similar to those reported for the 100,000-scenario instances in Figure 5, indicating that APH can make significantly better use of limited computational resources than PH for extremely large stochastic programs.

7.2 Results: `aircond`

For a second demonstration, we used the `aircond` (air conditioner) problem class, which involves a single-product planning model with overtime production, inventory carry-forward, and quadratic penalties for backorders. This model can be generated with any number of stages and any stochastic process for uncertain demand. For this demonstration, we generated a 5-stage problem instance with 1,000,000 scenarios and demands evolving according to an AR1 (one-stage autoregressive) random walk. For more details regarding this model, see Appendix B.

For `aircond`, the standard Lagrangian lower bounder spoke available in `mpi-sppy` computes high quality bounds, so we ran with two spokes, one being the same upper bounder used in the `ssn` experiments and the other being a Lagrangian lower bounder. We terminated the runs when the bounds produced by the two spokes converged to essentially the same value.

We used a bundle size of 1,000 scenarios, as with `ssn`. However, `aircond` bundles were most efficiently solved by the Gurobi's QP simplex method, as opposed to its barrier method.

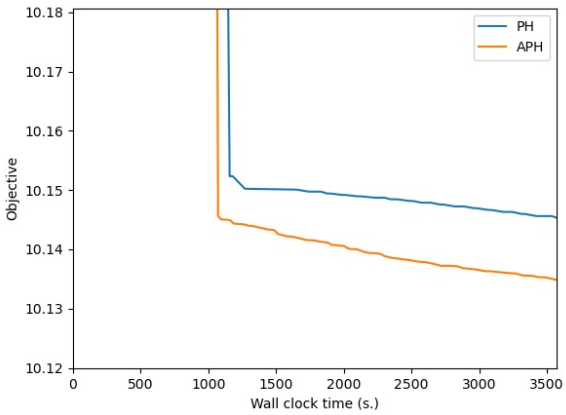

Figure 6: Results for a single instance of a 1M-scenario `ssn` instance, assigning 5 bundles of 1,000 scenarios to each MPI rank.

The simplex solver is less able to make use of parallelism, so we allocated 2 cores per rank, instead the 6 used for `ssn`. We packed 9 ranks into each node of the `quartz` system, using 18 of each node's 36 cores. Using more cores resulted in memory contention slowdowns within the nodes. We experimented with the following configurations:

1 bundle per rank: Each rank was allocated one bundle, with both PH and APH processing this single bundle at each iteration. In this configuration, each cylinder used 1,000 ranks and 2,000 CPU cores. Including the hub and both spokes, the runs used 3,000 ranks and 6,000 CPU cores, spread over 334 nodes.

5 bundles per rank: Each rank was allocated 5 bundles. PH solved all 5 bundles in each iteration, but APH solved just one bundle, selected by the "greedy" heuristic. Each cylinder used 200 ranks and 400 cores. The full hub-and-spoke configuration used 600 ranks and 1,200 cores, spread across 67 nodes.

10 bundles per rank: In this case, 10 bundles were assigned to each rank. Again, PH must process every bundle at each iteration, but we configured APH to process just one, selected in the same greedy manner. Each cylinder used 100 ranks and 200 cores, and the full configuration employed 300 ranks and 600 cores, allocated across 34 nodes.

The results are shown in Figure 7, with the left-hand chart showing 1 bundle per rank, the middle chart showing 5 bundles per rank, and the right-hand chart showing 10 bundles per rank. The graphs show the objective value for best incumbent (upper bound) solution at each point, and the runs were terminated when this valued converged with best value produced by the lower bounder. APH converges significant faster than PH in all cases, but its relative margin of superiority increases as the available computing power shrinks and the number of bundles per rank increases. The results thus resemble those for `ssn`: with more bundles per rank, APH does less work per iteration, while PH must perform the same work per iteration with fewer resources. While the lower-effort iterations affect the convergence rate of APH, the effect is less than proportional to the decrease in effort per iteration, so APH's relative margin of superiority over APH increases.

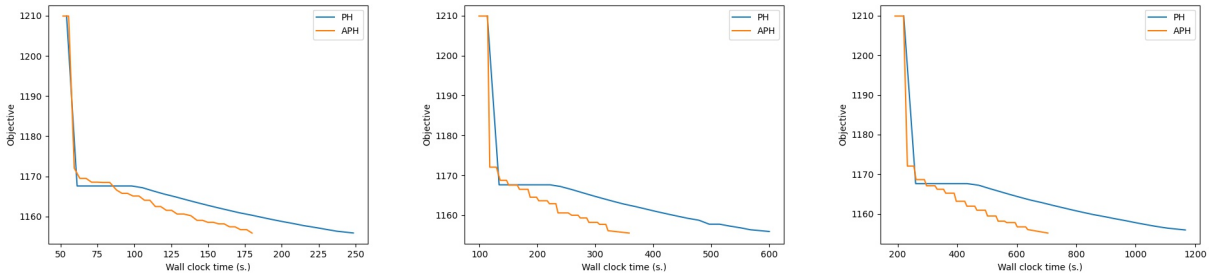

Figure 7: Results for a 5-stage, 1,000,000-scenario `aircond` instance. The left panel shows 1 bundle per rank, the middle panel shows 5 bundles per rank, and right panel shows 10 bundles per rank. In all figures, the reported wall clock time is on the LLNL `quartz` HPC cluster.

8 Possible Further Work

APH presents opportunities for further research both in theory and in applications. A dual derivation of the method may be worth pursuing and might yield convergence under different assumptions (for example, without the compactness condition in Assumption 1). Although we have so far only studied the theory of APH for the convex case, the software we have developed can be applied to a very broad range of multi-stage optimization problems with uncertainty represented by scenarios. Progressive hedging has helped advance the applicability of stochastic programming methods and we hope that the projective hedging approach will contribute to further advances. In addition to experimentally exploring the practical behavior of APH for mixed-integer and other nonconvex problems, its theoretical behavior in such settings also deserves study, and it would be interesting to consider whether it can be adapted to provide lower bounds on the optimal solution value in such cases.

Acknowledgments

This research was funded in part by National Science Foundation grant CCF-1617617.

References

[1] Abdullah Alotaibi, Patrick L. Combettes, and Naseer Shahzad. Solving coupled composite monotone inclusions by successive Fejér approximations of their Kuhn-Tucker set. *SIAM J. Optim.*, 24(4):2076–2095, 2014.

[2] Gilles Bareilles, Yassine Laguel, Dmitry Grishchenko, Franck Iutzeler, and Jérôme Malick. Randomized progressive hedging methods for multi-stage stochastic programming. *Ann. Oper. Res.*, 295(2):535–560, 2020.

[3] Heinz H. Bauschke and Patrick L. Combettes. *Convex analysis and monotone operator theory in Hilbert spaces*. Springer, 2013. Second edition.

[4] Dimitri P. Bertsekas, Angelia Nedić, and Asuman E. Ozdaglar. *Convex Analysis and Optimization*. Athena Scientific, Belmont, MA, USA, 2003.

[5] Natashia Boland, Jeffrey Christiansen, Brian Dandurand, Andrew Eberhard, Jeff Linderoth, James Luedtke, and Fabricio Oliveira. Combining progressive hedging with a Frank-Wolfe method to compute Lagrangian dual bounds in stochastic mixed-integer programming. *SIAM J. Optim.*, 28(2):1312–1336, 2018.

[6] Minh N. Bùi, Patrick L. Combettes, and Zev C. Woodstock. Block-activated algorithms for multicomponent fully nonsmooth minimization. Preprint 2103.00520, ArXiv, 2021.

[7] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo—optimization modeling in Python*, volume 67 of *Springer Optimization and Its Applications*. Springer, third edition, 2021.

[8] Patrick L. Combettes and Jonathan Eckstein. Asynchronous block-iterative primal-dual decomposition methods for monotone inclusions. *Math. Program.*, 126(1–2):645–672, 2018.

[9] Patrick L. Combettes and Jean-Christophe Pesquet. Stochastic quasi-Fejér block-coordinate fixed point iterations with random sweeping. *SIAM J. Optim.*, 25(2):1221–1248, 2015.

[10] Teodor Gabriel Crainic, Xiaorui Fu, Michel Gendreau, Walter Rei, and Stein W. Wallace. Progressive hedging-based metaheuristics for stochastic network design. *Networks*, 58(2):114–124, 2011.

[11] Teodor Gabriel Crainic, Mike Hewitt, and Walter Rei. Scenario grouping in a progressive hedging-based meta-heuristics for stochastic network design. *Comput. Oper. Res.*, 43:90–99, 2014.

[12] Jonathan Eckstein. A simplified form of block-iterative operator splitting and an asynchronous algorithm resembling the multi-block alternating direction method of multipliers. *J. Optim. Theory Appl.*, 173(1):155–182, 2017.

[13] Jonathan Eckstein and Dimitri P. Bertsekas. On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators. *Math. Program.*, 55(3):293–318, 1992.

[14] Jonathan Eckstein and B. F. Svaiter. General projective splitting methods for sums of maximal monotone operators. *SIAM J. Control Optim.*, 48(2):787–811, 2009.

[15] Michel Fortin and Roland Glowinski. On decomposition-coordination methods using an augmented Lagrangian. In M. Fortin and R. Glowinski, editors, *Augmented Lagrangian methods: Applications to the numerical solution of boundary-value problems*, volume 15 of *Studies in Mathematics and its Applications*, pages 97–146. North-Holland, 1983.

[16] Daniel Gabay. Applications of the method of multipliers to variational inequalities. In M. Fortin and R. Glowinski, editors, *Augmented Lagrangian methods: Applications to the numerical solution of boundary-value problems*, volume 15 of *Studies in Mathematics and its Applications*, pages 299–340. North-Holland, 1983.

[17] Dinakar Gade, Gabriel Hackebeil, Sarah M. Ryan, Jean-Paul Watson, Roger J.-B. Wets, and David L. Woodruff. Obtaining lower bounds from the progressive hedging algorithm for stochastic mixed-integer programs. *Math. Program.*, 157(1):47–67, 2016.

[18] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI – The Complete Reference, Volume 2:, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[19] Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2022.

[20] William E. Hart, Jean-Paul Watson, and David L. Woodruff. Pyomo: Modeling and solving mathematical programs in Python. *Math. Program. Comput.*, 3(3), 2011.

[21] Mingyi Hong. A distributed, asynchronous, and incremental algorithm for nonconvex optimization: An ADMM approach. *IEEE Transactions on Control of Network Systems*, 5(3):935–945, 2018.

[22] Franck Iutzeler, Pascal Bianchi, Philippe Ciblat, and Walid Hachem. Asynchronous distributed optimization using a randomized alternating direction method of multipliers. In *52nd IEEE Conference on Decision and Control*, pages 3671–3676, 2013.

[23] Patrick R. Johnstone and Jonathan Eckstein. Projective splitting with forward steps. *Math. Programming*, 2020. Available online.

[24] Anton J. Kleywegt, Alexander Shapiro, and Tito Homem-de Mello. The sample average approximation method for stochastic discrete optimization. *SIAM J. Optim.*, 12(2):479–502, 2002.

[25] Bernard Knueven, David Mildebrath, Christopher Muir, John Siirola, Jean-Paul Watson, and David Woodruff. A parallel hub-and-spoke system for large-scale scenario-based optimization under uncertainty. Preprint 2020-11-8088, Optimization Online, 2020.

[26] Jeff Linderoth, Alexander Shapiro, and Stephen Wright. The empirical behavior of sampling methods for stochastic programming. *Ann. Oper. Res.*, 142:215–241, 2006.

[27] Pierre-Louis Lions and Bertrand Mercier. Splitting algorithms for the sum of two nonlinear operators. *SIAM J. Numer. Anal.*, 16(6):964–979, 1979.

[28] Zhimin Peng, Yangyang Xu, Ming Yan, and Wotao Yin. ARock: an algorithmic framework for asynchronous parallel coordinate updates. *SIAM J. Sci. Comput.*, 38(5):A2851–A2879, 2016.

[29] Guido Perboli, Luca Gobbato, and Francesca Maggioni. A progressive hedging method for the multi-path traveling salesman problem with stochastic travel times. *IMA J. Manage. Math.*, 28:65–86, 2015.

[30] R. Tyrrell Rockafellar. *Convex analysis*. Princeton University Press, 1970.

[31] R. Tyrrell. Rockafellar and Roger J.-B. Wets. Scenarios and policy aggregation in optimization under uncertainty. *Math. Oper. Res.*, 16(1):119–147, 1991.

[32] Kevin Ryan, Deepak Rajan, and Shabbir Ahmed. Scenario decomposition for 0-1 stochastic programs: Improvements and asynchronous implementation. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 722–729, 2016.

[33] Suvrajeet Sen, Robert D. Doverspike, and Steve Cosares. Network planning with random demand. *Telecommun. Syst.*, 3(1):11–30, 1994.

[34] Suvrajeet Sen and Yifan Liu. Mitigating uncertainty via compromise decisions in two-stage stochastic linear programming: Variance reduction. *Oper. Res.*, 64(6):1422–1437, 2016.

[35] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, 2nd. (revised) edition, 1998.

[36] Michael Somervell. Progressive hedging in parallel. In *Proceedings of the 33rd Annual Conference of the Operational Research Society of New Zealand*, pages 84–93, 1998.

[37] Jean-Paul Watson and David L. Woodruff. Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Comp. Manage. Sci.*, 8(4):355–370, 2011.

[38] E. Wei and A. Ozdaglar. On the O(1/k) convergence of asynchronous distributed alternating direction method of multipliers. In Ahmed Tewfik, editor, *2013 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 551–554, 2013. Proofs and additional relevant material in ArXiv preprint 1307.8524.

[39] David L. Woodruff, Brian C. Knight, Xiaotie Chen, and Sylvain Cazaux. `Aircond`: an example for optimization under uncertainty. Technical report, GitHub, 2022. `https://github.com/DLWoodruff/aircond/blob/main/aircond.pdf`.

A Proof of Proposition 5

Proof. First, consider any z solving (14). Applying [3, Proposition 16.5(ii)] n times, we have that

$$\sum_{i=1}^{n} L_i^* \partial f_i(L_i z) \subseteq \partial H(z), \qquad \text{where} \quad H : u \mapsto \sum_{i=1}^{n} f_i(L_i u).$$

Therefore, $0 \in \partial H(z)$ and hence z solves problem (10). Further, for each $i \in 1..n$, some x_{iT} as defined in (15) must exist by Assumption 1, and the optimality for (7) of x as constructed in (15) is immediate from the setup S1-S4. This proves the first half of the proposition.

We now commence the proof of the second half of the proposition, which supposes that x is some optimal solution to (7) and that assumptions 1 and 4 hold. By Proposition 3, $z = Mx$ must be an optimal solution of (10), which is equivalent to $\partial H(z) \ni 0$. We must show that this condition is equivalent to (14).

To this end, we first claim that $\partial(f_i \circ L_i) = L_i^* \circ \partial f_i \circ L_i$ for all $i \in 1..n$. To see this, first fix any $i \in 1..n$. From [3, Corollary 16.42(i)], we know that the claim will hold if $0 \in \text{sri}(\text{dom}\, f_i - \text{rge}\, L_i)$, where "sri" denotes the strong relative interior of a set, which is identical to the relative interior in finite dimension. From S3, $\text{rge}\, L_i = \mathcal{H}_i$, so we have $\text{sri}(\text{dom}\, f_i - \text{rge}\, L_i) = \text{ri}\, \mathcal{H}_i = \mathcal{H}_i \ni 0$, and the claim is established.

To complete the proof, it is sufficient to show that $\partial\big(\sum_{i=1}^n f_i \circ L_i\big) = \sum_{i=1}^n \partial(f_i \circ L_i)$, since we would then have

$$\partial H = \partial\big(\sum_{i=1}^n f_i \circ L_i\big) = \sum_{i=1}^n \partial(f_i \circ L_i) = \sum_{i=1}^n L_i^* \circ \partial f_i \circ f_i,$$

where the last equality follows from the previous claim. Instantiating this equation at z yields (14).

We establish that $\partial\big(\sum_{i=1}^n f_i \circ L_i\big) = \sum_{i=1}^n \partial(f_i \circ L_i)$ by induction on the number of terms in the sum. That is, we will show that for any $k \in 1..n$, one has $\partial\big(\sum_{i=1}^k f_i \circ L_i\big) = \sum_{i=1}^k \partial(f_i \circ L_i)$. This clearly holds for $k = 1$. Now suppose it holds for an arbitrary $k < n$; we will establish that it must then hold for $k + 1$.

First, consider the case that Assumption 4(1) holds. We start this case by observing, for each $i \in 1..n$, that h_i being a polyhedral function implies that f_i is polyhedral. Specifically, $(z_i, \alpha) \in \text{epi}\, \mathcal{H}_i \times \mathbb{R}$ is in $\text{epi}\, f_i$ if and only if there exists $x_{iT} \in \mathbb{R}^{m_T}$ such that $\big((z_i, x_{iT}), \alpha/\pi_i\big) \in \text{epi}\, h_i$. Thus, $\text{epi}\, f_i$ is the image of $\text{epi}\, h_i$ under the linear mapping $P : \big((z_i, x_{iT}), \beta\big) \mapsto (z_i, \pi_i \beta)$. Since h_i is defined as $\text{epi}\, h_i$ being a polyhedral set and the spaces involved are all finite-dimensional, the argument of [30, Theorem 19.3] implies that $\text{epi}\, f_i = P(\text{epi}\, h_i)$ is polyhedral, and f_i is polyhedral. The argument of [30, Corollary 19.3.1] then implies that $f_i \circ L_i$ is also polyhedral, and repeated use of the argument of [30, Theorem 19.4] establishes that $\sum_{i=1}^k f_i \circ L_i$ is polyhedral, being a finite sum of polyhedral functions. Now, we are in the situation that $\sum_{i=1}^k f_i \circ L_i$ is polyhedral, $f_{k+1} \circ L_{k+1}$ is polyhedral, and their domains intersect, both containing the point $z = Mx$. Since the setting is finite dimensional, we have

$$\begin{aligned}
\partial\big(\sum_{i=1}^{k+1} f_i \circ L_i\big) &= \partial\left(\big(\sum_{i=1}^{k} f_i \circ L_i\big) + f_{k+1} \circ L_{k+1}\right) && (30)\\
&= \partial\big(\sum_{i=1}^{k} f_i \circ L_i\big) + \partial(f_{k+1} \circ L_{k+1}) && \text{[3, Theorem 16.37(iii)]}\\
&= \sum_{i=1}^{k} \partial(f_i \circ L_i) + \partial(f_{k+1} \circ L_{k+1}) && \text{[By the induction hypothesis]}\\
&= \sum_{i=1}^{k+1} \partial(f_i \circ L_i), && (31)
\end{aligned}$$

and the induction is complete.

We next turn to the situation in which Assumption 4(2) holds. We begin this case by observing that for any $i \in 1..n$,

$$\operatorname{dom}(f_i \circ L_i) = \{u \in \mathcal{H}_0 \mid L_i u \in \operatorname{dom} f_i\} = L_i^{-1} \operatorname{dom} f_i,$$

where $L_i^{-1}S$ denotes the preimage of the set S under the linear map L_i. Since we are in finite dimension, applying the argument of [30, Theorem 6.7] leads to the conclusion that

$$(\forall\, i \in 1..n) \qquad \operatorname{ri}\operatorname{dom}(f_i \circ L_i) = \operatorname{ri} L_i^{-1} \operatorname{dom} f_i = L_i^{-1} \operatorname{ri}\operatorname{dom} f_i. \tag{32}$$

Next, we observe that the constraint qualification (13) is equivalent to

$$(\forall\, i \in 1..n) \qquad L_i M\bar{x} \in \operatorname{ri} M_i(\operatorname{dom} h_i) = \operatorname{ri}\operatorname{dom} f_i, \tag{33}$$

where the equality is in view of (12). Therefore, we know that $\bar{z} = M\bar{x}$ has the property the $L_i\bar{z} \in \operatorname{ri}\operatorname{dom} f_i$ for all $i \in 1..n$. Hence, for any $i \in 1..n$, we have $\bar{z} \in L_i^{-1} \operatorname{ri}\operatorname{dom} f_i = \operatorname{ri}\operatorname{dom}(f_i \circ L_i)$.

For the Assumption 4(2) case, we adjoin to the induction hypothesis that

$$\operatorname{ri}\operatorname{dom}\left(\sum_{i=1}^{k} f_i \circ L_i\right) = \bigcap_{i=1}^{k} \big(L_i^{-1} \operatorname{ri}\operatorname{dom} f_i\big),$$

which holds for $k = 1$ by letting $i = 1$ in (32).

We may rewrite (33) as $\bar{z} \in \bigcap_{i=1}^{n} \big(L_i^{-1} \operatorname{ri}\operatorname{dom} f_i\big)$, hence we have both

$$\begin{aligned} \bar{z} &\in \bigcap_{i=1}^{k} \big(L_i^{-1} \operatorname{ri}\operatorname{dom} f_i\big) = \operatorname{ri}\operatorname{dom}\left(\sum_{i=1}^{k} f_i \circ L_i\right) \\ \bar{z} &\in L_{k+1}^{-1} \operatorname{ri}\operatorname{dom} f_{k+1} = \operatorname{ri}\operatorname{dom} f_{k+1} \circ L_{k+1}. \end{aligned}$$

We may then invoke the same inductive logic as in (30)-(31), but using Corollary 16.38(iv) instead of Theorem 16.37(iii) of [3], and conclude that $\partial\big(\sum_{i=1}^{k+1} f_i \circ L_i\big) = \sum_{i=1}^{k+1} \partial(f_i \circ L_i)$. To complete the induction in this case, we also need to establish that

$$\operatorname{ri}\operatorname{dom}\left(\sum_{i=1}^{k+1} f_i \circ L_i\right) = \bigcap_{i=1}^{k+1} \big(L_i^{-1} \operatorname{ri}\operatorname{dom} f_i\big).$$

This relation follows by combining the additional induction hypothesis with [3, Fact 6.14(v)], because $\bar{z}$ lies in both $\operatorname{ri}\operatorname{dom}\big(\sum_{i=1}^{k} f_i \circ L_i\big)$ and $\operatorname{ri}\operatorname{dom} f_{k+1} \circ L_{k+1}$. □

B The `Aircond` Model

In addition to the `ssn` family of test problems, we performed computational tests on a single-product multi-period production planning model with overtime production, inventory carry-forward, and quadratic back-order costs. Demand for the product need not be stage-wise independent. In the interest of completeness, we describe here the version of the model used for the experiments in this paper. More information is provided in the technical report [39] available on GitHub.

B.1 Parameters

The following quantities are taken as parameters in our construction of the problem. For our purposes, only the demand $\mathbf{D}_t$ is stochastic, although the formulation may be extended to have stochastic inventory capacities and costs. Each parameter takes a real nonnegative value. All are known at the beginning of period 1 except for the demand.

$\mathbf{D}_t$	Random variable for demand in period $t = 1, \ldots, T$
D_t	Realized demand in period $t = 1, \ldots, T$
I_t	Single-period unit inventory cost in period $t = 1, \ldots, T$
N_t	Linear cost coefficient for negative inventory (failure to meet cumulative demand) in period $t = 1, \ldots, T$ (positive)
Q_t	Quadratic cost coefficient for single-period negative inventory (failure to meet cumulative demand) in period $t = 1, \ldots, T$
K_t	Regular-time capacity in period $t = 1, \ldots, T$
C_t	Regular-time unit production cost in period $t = 1, \ldots, T$
O_t	Overtime unit production cost in period $t = 1, \ldots, T$
y_0	Beginning inventory.

Demand is known at the beginning of period t, $\mathbf{D}_1$ is degenerate.

B.2 Variables

Given T stages (periods) and a single product, we define the following variables over each stage $t = 1, \ldots, T$:

$x_t \in \mathbb{R}_+$	Number to make in regular time (decided at the start of period t)
$w_t \in \mathbb{R}_+$	Number to make with overtime (decided at the start of period t)
$y_t \in \mathbb{R}$	Number to carry forward (determined by x_t, w_t, y_{t-1}, and the demand during period t)
$y_t^+ \in \mathbb{R}_+$	The positive part of y_t
$y_t^- \in \mathbb{R}_+$	The negative part of y_t

B.3 Formulation

The objective is to minimize the expected cost of production subject to inventory and demand constraints. Defining for each $t = 1, \ldots, T$

$$\vec{D}^t = (D_1, \ldots, D_t) \qquad\qquad \vec{\mathbf{D}}^t = (\mathbf{D}_1, \ldots, \mathbf{D}_t)$$

the problem is defined recursively by:

$$\begin{array}{ll} \min\limits_{x_1} & \left(C_1 x_1 + O_1 w_1 + I_1 y_1^+ + N_1 y_1^- + (Q_1)\left(y_1^-\right)^2 + \mathbb{E}_{\mathbf{D}}[\phi_2(x_1; D_1, \mathbf{D})]\right) \\ \text{ST} & x_1 \le K_1, \\ & y_0 + x_1 + w_1 - y_1 = D_1 \\ & y_1^+ - y_1^- = y_1 \\ & x_1, y_1^+, y_1^-, w_1 \ge 0 \end{array}$$

where, for $t = 2, \ldots, T-1$,

$$\begin{aligned}
\phi_t(x_{t-1}; \vec{D}^t, \mathbf{D}) = \min_{x_t} \quad & \Big(C_t x_t + O_t w_t + I_t y_t^+ + N_t y_t^- + (Q)_t \left(y_t^-\right)^2 \\
& \qquad + \mathbb{E}_{\mathbf{D}} \Big[\phi_{t+1}(x_t; \vec{x}^{t-1}, \vec{D}^t, \mathbf{D}^{t+1} \mid \vec{\mathbf{D}}^t = \vec{D}^t) \Big] \Big) \\
\text{ST} \quad & x_t \leq K_t, \\
& y_{t-1} + x_t + w_t - y_t = D_t \\
& y_t^+ - y_t^- = y_t \\
& x_t, y_t^+, y_t^-, w_t \geq 0
\end{aligned}$$

and

$$\begin{aligned}
\phi_T(x_{T-1}; \vec{D}^T, \cdot) = \min_{x_T} \quad & \Big(C_T x_T + O_T w_T + I_T y_T^+ + N_T y_T^- + (Q)_T \left(y_T^-\right)^2 \Big) \\
\text{ST} \quad & x_T \leq K_T, \\
& y_{T-1} + x_T + w_T - y_T = D_T \\
& y_T^+ - y_T^- = y_T \\
& x_T, y_T^+, y_T^-, w_T \geq 0.
\end{aligned}$$

B.4 Sample Data

Most of the parameter values in our experiments are fixed across stages, except that the unit inventory cost changes significantly in the final stage, effectively becoming a credit for inventory at the end of the time horizon. We used the assignments

$D_1 = 200$

$I_t = 0.5$, $t = 1, \ldots, T-1$, $I_T = -0.8$

$N_t = 0.5$, $t = 1, \ldots, T$

$Q_t = 0.3$, $t = 1, \ldots, T$

$K_t = 200$, $t = 1, \ldots, T$

$C_t = 1$, $t = 1, \ldots, T$

$O_t = 3$, $t = 1, \ldots, T$

$y_0 = 50$.

We assumed the demand process $\mathbf{D}$ to be a random walk whose increments follow a normal distribution with mean 0 and standard deviation 40, with barriers at 0 and 400. With a branching factor of b_t at period $t = 1, \ldots, T-1$, take b_t i.i.d. samples d_t from $\mathcal{N}(0, 40)$, define R_t to be a discrete random variable with with probability $1/b_t$ of taking each of those values, and define D_{t+1} as

$$D_{t+1} = \begin{cases} 0 & \text{if } D_t + R_t \leq 0 \\ 400 & \text{if } D_t + R_t \geq 400 \\ D_t + R_t & \text{otherwise.} \end{cases}$$

We used $T = 5$ stages and branching factors $b_1 = 1000$, $b_2 = 25$, $b_3 = 10$, and $b_4 = 4$. This setup results in a scenario tree with $1000 \cdot 25 \cdot 10 \cdot 4 = 1{,}000{,}000$ nodes.