Scenario Decomposition for 0-1 Stochastic Programs: Improvements and Asynchronous Implementation

Kevin Ryan[1], Deepak Rajan[2], and Shabbir Ahmed[1]

[1]School of Industrial & Systems Engineering, Georgia Institute of Technology
[2]Lawrence Livermore National Laboratory

November 13, 2015

Abstract

A recently proposed scenario decomposition algorithm for stochastic 0-1 programs finds an optimal solution by evaluating and removing individual solutions that are discovered by solving scenario subproblems. In this work, we develop an asynchronous, distributed implementation of the algorithm which has computational advantages over existing synchronous implementations of the algorithm. Improvements to both the synchronous and asynchronous algorithm are proposed. We test the results on well known stochastic 0-1 programs from the SIPLIB test library and are able to solve previously unsolved instances from the test set.

1 Introduction

In a recent paper [1], Ahmed proposes a decomposition algorithm for two-stage, 0-1 stochastic programs and describes serial and synchronous distributed implementations of the algorithm. The synchronous parallel approach is applied to a test set of instances from SIPLIB [2], and performs well when compared to CPLEX and other known implementations. In this paper, we seek to improve upon this distributed implementation in two ways.

First, we propose improvements to the serial and parallel algorithms. Next, we develop an asynchronous, distributed implementation which incorporates these improvements, and show that it outperforms the synchronous implementation. The asynchronous version of the algorithm performs well and is able to solve previously unsolved instances from [2]. We formally describe the class of problems which our algorithm is built to solve, two-stage 0-1 stochastic programs, below.

1.1 Two-Stage 0-1 Stochastic Programs

In this work, following the terminology of [1], we consider two-stage stochastic programs of the form

$$\min\{\mathbb{E}_P[f(x,\xi)] : x \in X \subseteq \{0,1\}^n\} \tag{1}$$

where x is the first stage solution decision variable and ξ is a random vector with support Ξ sampled from a known distribution P. The set X represents the set of feasible first stage solutions. For a given first stage solution x, the second stage problem is of the form.

$$f(\xi, x) = cx + \min\{\phi(y(\xi), \xi) : y(\xi) \in Y(\xi, x)\}$$

If $Y(\xi, x) = \emptyset$, then $f(\xi, x) = \infty$. Throughout this paper, we assume a finite support for ξ and thus a finite number of scenarios. We can then reformulate (1) as

$$\min \left\{ \sum_{k=1}^{K} f_k(x^k) : \; x^k = x^1, \; x^k \in X \subseteq \{0,1\}^n \; \forall k \in [K] \right\} \tag{2}$$

where K is the number of scenarios. We denote $f_k(x) = p_k f(\xi, x)$, where p_k is the probability that scenario k is realized. In (2), there exists a first stage decision vector x^k for each scenario $k \in [K]$. These first stage variables must be identical across scenarios due to the nonanticipativity constraints $x^k = x^1$. If the function $f_k(x^k)$ can be evaluated by solving a mixed integer program, then (2) can be solved as one large mixed integer program. We call this mixed integer program the *extensive formulation*.

1.2 Prior Work

In many algorithms designed to solve two-stage stochastic programs, a scenario decomposition approach is used. In these approaches, the nonanticipativity constraint on the first stage variables is dualized, producing the following lower bounding problem.

$$\min \left\{ \sum_{k=1}^{K} \left[f_k(x^k) + \sum_{j<k} \lambda_{kj}(x^k - x^j) \right] : \; x^k \in X \subseteq \{0,1\}^n \right\}$$

This gives a valid lower bound for (2) but may not produce feasible solutions. Our work is an exact scenario decomposition approach that is primarily based on the scenario decomposition algorithm proposed in [1] and incorporates optimality cuts proposed in [15] and used in [8].

There also exist other exact decomposition approaches, many of which are able to be adapted to parallel implementations. Dual Decomposition, proposed in [6], involves the original problem after dualizing the nonanticipativity constraint, 'averaging' the first stage solutions from the scenario subproblems and if not integer feasible, branching on the fractional variables. Branch and Fix, originally proposed in [3], solves the original problem by dualizing the nonanticipativity constraint with $\lambda = 0$ and then performing branching and node selection in a manner which enforces the nonanticipativity constraints.

Heuristic decomposition approaches include "Progressive Hedging" heuristics such as [7], [10], [16] as well as distributed-memory implementations of Dual Decomposition such as [8], [5]. In the "Progressive Hedging" style heuristics, the dual penalty on the nonanticipativity relaxation is updated to encourage agreement amongst the first stage solutions. Variable fixing is also done in certain "Progressive Hedging" algorithms in order to reduce the time until first stage variable agreement. In [8], additional cutting planes are introduced to a Dual Decomposition scheme which does not branch on integer variables. In [5], the authors present an asynchronous implementation of a Dual Decomposition scheme that illustrates some of the limitations of a synchronous implementation. These heuristics do not guarantee finite convergence, but some have been found to work well in practice.

1.3 Definitions

Throughout this paper, we denote the set $\{1, \ldots, K\}$ as $[K]$.

Definition 1. We define the *Second Stage Problem* $\forall k \in [K]$ as

$$f_k(x^k) = \min\{g(x^k, y^k) : y^k \in Y_k(x^k)\} \tag{3}$$

Observe that the function $f_k(x^k)$ in (2) can be evaluated, for a given x^k, by solving the Second Stage Problem for scenario k. No assumptions about the feasibility of the Second Stage Problem are required for the algorithm to converge finitely. Consider the relaxation given by removing the nonanticipativity constraints, $x^i = x^j$, which can be decomposed into the following independent problems for each scenario.

Definition 2. We define *Scenario Problems* $\forall k \in [K]$ as

$$z_k^*(X) = \min\left\{f_k(x^k) : \; x^k \in X \subseteq \{0,1\}^n\right\} \tag{4}$$

If (4) is infeasible, then $z_k^*(X) = \infty$. Note that if the set of first stage solutions consists of exactly one point, $X = \{x^s\}$, we may fix the first stage variables of the scenario problem to be equal to x^s.

Finally, we describe the following class of cuts which are added to the set X over the course of the algorithm presented in this paper. These cuts are essential to guarantee finite convergence.

Definition 3. Given a solution $x^* \in \{0,1\}^n$, let

- $O(x^*) = \{j \in [n] : x_j^* = 1\}$
- $Z(x^*) = \{j \in [n] : x_j^* = 0\}$

Then the set $\{0,1\}^n \setminus x^*$ can be represented using the *No-Good Cut*, written as

$$\sum_{j \in O(x^*)} (1 - x_j) + \sum_{j \in Z(x^*)} x_j \geq 1$$

This definition can be extended for several solutions using several No-Good cuts. We denote the addition of a no-good cut for a solution x^* to the scenario problems by writing $X \leftarrow X \setminus x^*$ and the addition of several no-good cuts removing a set of first stage solutions S by $X \leftarrow X \setminus S$.

1.4 Overview

The paper proceeds as follows. In Section 2, we define the synchronous version of the algorithm originally proposed in [1]. In Section 3, we present out proposed performance improvements. Section 4 contains a complete description of our asynchronous implementation. Finally, Section 5 presents the results of our computational experiments on four sets of instances from [2] and [11].

2 Synchronous Algorithm [1]

We present a description of the original synchronous parallel algorithm as proposed in [1]. The synchronous algorithm with exactly one worker is equivalent to the serial algorithm proposed in the same work. We present only the synchronous version for the sake of brevity.

2.1 Algorithm

The original version of the synchronous algorithm is given below. We begin with W worker processes. Let *Synch Point* denote synchronization points in the algorithm at which workers must wait for all other workers to complete their tasks before continuing.

During each master iteration of the algorithm, feasible first stage solutions $(S \subseteq X)$ are collected by solving individual Scenario Problems independently (lines 4 and 5). Evaluating each first stage solution, x^s, by solving for $z_k^*(x^s)\ \forall k \in [K]$ gives us an upper bound (line 9). The lower bound is given by the sum of the scenario problem optimal values (line 12). We remove these first-stage solutions by adding No-Good Cuts removing these solutions to all Scenario Problems. This reduces the set of candidate of first-stage solutions for future iterations (line 14), guaranteeing that the lower bound can only increase from one iteration to the next.

Algorithm 1 Synchronous Algorithm

1: Initialize: Set $UB = \infty, LB = -\infty, X = \{0,1\}^n, S = \emptyset$
2: **while** $UB > LB$ **do**
3: **for** k=1 to K **do**
4: Solve for optimal x^k, $z_k^*(X)$ on worker $(k \bmod W)$
5: $S \leftarrow S \cup x^k$
6: **end for**
7: *Synch Point:* End Scenario Solve
8: **for** k=1 to K **do**
9: Evaluate $z_k^*(x^s)$ on worker $(k \bmod W)$, for all $x^s \in S$
10: **end for**
11: *Synch Point:* End Solution Evaluation
12: $LB \leftarrow \sum_{k=1}^{K} z_k^*(X)$
13: $UB \leftarrow \min_{x^s \in S} \left(UB, \sum_{k=1}^{K} z_k(x^s) \right)$
14: $X \leftarrow X - S$
15: $S \leftarrow \emptyset$
16: *Synch Point:* End Bound Calculation
17: **end while**

The validity of the lower bound is clear during the first master iteration as $\sum_{k=1}^{K} z_k^*(X)$ is equal to the value of the nonanticipativity relaxation bound. In subsequent iterations, $\sum_{k=1}^{K} z_k^*(X)$ is equal to the value of the non-anticipativity relaxation of a similar problem restricted to first stage variables which have not yet been evaluated.

If any scenario problem k becomes infeasible, then $z_k^*(X) = \infty$ and the algorithm terminates. In this case, if $UB = \infty$, then the problem is infeasible. If $UB < \infty$, then we have evaluated all feasible first stage solutions and the original problem has been solved to optimality. Given this statement and the fact that S is finite, it is evident that the algorithm is finitely convergent.

3 Algorithmic Improvements

In this section we discuss improvements which reduce the running time of the serial and, in some cases, the synchronous version of the algorithm. These improvements come in two forms. First, we develop a method to reduce the time spent evaluating candidate solutions by calculating lower bounds on solution values and aborting the evaluation if these bounds are above the current upper bound. Second, we explore adding a known class of optimality cuts, originally proposed in [15], to the set of feasible first stage solutions.

3.1 Lower Bound on First Stage Evaluation

When evaluating x^s in line 9 of Algorithm 1, we must solve each of the second stage problems in order compute $\sum_{k=1}^{K} z_k^*(x^s)$. If computing $z_k^*(x^s)$ involves solving a mixed integer problem, we may quickly compute the linear programming relaxation of the problem with objective value $w_k^*(x^s)$. If we observe that $\sum_{k=1}^{K} w_k^*(x^s) \geq UB$ before evaluating $\sum_{k=1}^{K} z_k(x^s)$, we may terminate the solution evaluation early. A similar version of this idea was shown to be effective in [4].

The above approach can be effective if the linear programming relaxation of the second stage problem is tight, which may not always be the case. We develop the following bound which, in our experiments, almost always terminates first stage evaluations quickly provided a good upper bound is known.

Our goal is to develop a lower bound for $\sum_{k=1}^{K} z_k^*(x^s)$ which we may check against UB in order to terminate the evaluation early. In our algorithm, we evaluate each $z_k(x^s)$ sequentially over $[K]$. At the time of evaluation (line 9), we know that $\forall k \in [K]$, the scenario bound $z_k^*(X)$ satisfies

$$z_k^*(X) \leq z_k^*(x^s), \quad \forall k \in [K]$$

Suppose that we have so far evaluated $\sum_{k=1}^{l} z_k^*(x^s)$ for $1 < l < K$. Assuming that we have already calculated $w_k^*(x^s)$, $\forall k \in [K]$, we can check the following inequality before the evaluation of $z_{l+1}^*(x^s)$

$$\sum_{k=1}^{l} z_k^*(x^s) + \sum_{k=l+1}^{K} \max(w_k^*(x^s), z_k^*(X)) \geq UB$$

and if the above holds, we may terminate the evaluation of x^s early. The speedup from this improvement is substantial, especially for problem instances with a large number of scenarios. We demonstrate this in Section 5.2.

3.2 Optimality Cuts

We next consider adding the optimality cuts originally proposed in [15] to our scenario decomposition algorithm. These cuts are generated using a given feasible first stage solution x^s and may be added to the scenario problems when the no-good cut removing x^s has been added. The following is a brief description of the process. After evaluating a given solution x^s, we would like add a cut which removes all but those first stage solutions $\hat{x}$ which satisfy

$$UB \geq \sum_{k=1}^{K} z_k^*(\hat{x})$$

that is, better solutions which may improve the upper bound. We would like to construct a lower bound on $\sum_{k=1}^{K} z_k^*(\hat{x})$. Recall that, for a given fixing of the first stage solution x^s, the linear programming relaxation of the *Second Stage Problem* is

$$\begin{aligned} w_k(x^s) = c_k^T x^s + \quad \min_{y} \quad & q_k y^k \\ s.t \quad & W_k y^k \geq d_k - T_k x^s \\ & y^k \in \mathbb{R}_+^{m_1} \times \mathbb{R}_+^{m_2} \end{aligned}$$

and its dual problem is defined as

$$\begin{aligned} w_k(x^s) = c_k^T x^s + \max_{\alpha^k} \quad & \alpha^k (d_k - T_k x^s) \\ s.t \quad & W_k^T \alpha^k \leq q_k \\ & \alpha^k \in \mathbb{R}_+^{m_3} \end{aligned}$$

Let $\alpha^k(x^s)$ be an opimal α^k solution associated with the dual problem k for a given x^s. As the choice of x^s does not have any effect on the feasible region dual problem, $\alpha^k(x^s)$ must be feasible to the dual problem for any other choice of $\hat{x} \subseteq \{0,1\}^n$. Thus, for any $\hat{x}$, we have that

$$\sum_{k=1}^{K} z_k^*(\hat{x}) \geq \sum_{k=1}^{K} c_k \hat{x} + \alpha^k (x^s)^T (d_k - T_k \hat{x})$$

So, for any $\hat{x}$ which satisfies $UB \geq \sum_{k=1}^{K} z_k^*(\hat{x})$, it follows that

$$UB \geq \sum_{k=1}^{K} z_k^*(\hat{x}) \geq \sum_{k=1}^{K} c_k \hat{x} + \alpha^k (x^s)^T (d_k - T_k \hat{x})$$

and thus the cut

$$UB \geq \sum_{k=1}^{K} c_k \hat{x} + \alpha^k (x^s)^T (d_k - T_k \hat{x})$$

does not remove potential optimal first stage solutions. We demonstrate the effectiveness of the optimality cut within our scheme in Sections 5.4 and 5.5.

4 Asynchronous Parallel Implementation

The synchronous implementation proposed in [1] has the advantage of limited communication and coordination between worker processes and thus makes efficient use of additional resources for certain instances. However, if there is significant variability in scenario solve times or solution evaluation times, workers may sit idle at synchronization points. Additionally, as Algorithm 1 evaluates candidate first stage solutions in parallel, it will not be able to efficiently implement the performance improvements of Section 3.1. Finally, if K is not a multiple of W, for example $K = 11$ and $W = 10$, the workload may be unbalanced between workers. For these reasons, we consider the following asynchronous implementation which does not contain synchronization points.

Our goal is to reduce the amount of time during which worker processors may sit idle. In order to achieve this goal, we remove the synchronization points and instead rely on a master/worker implementation, in which the master process determines the current overall state of the algorithm and assigns tasks for the workers to process. We will first define a set of potential worker tasks and then describe the states for the master implementation.

4.1 Worker Tasks

1. $SOLVE\ SCENARIO(k, w, S)$
   - Input: Scenario k to evaluate on worker w, set of banned solutions S.
   - Output: $z_k^*(X \setminus S)$, corresponding optimal solution x^k
2. $EVALUATE\ SOLUTION(x^s, w)$
   - Input: Solution x^s to worker w
   - Output: $\sum_{k=1}^{K} z_k(x^s)$ or proof that $\sum_{k=1}^{K} z_k(x^s) \geq UB$

4.2 Master Algorithm

After an initial setup phase, the master process can be in one of two states. The two master states will be denoted *Master Solve Scenario* and *Master Evaluate Solution*. We next define the counters used by the master process

- ScenarioOut: Number of scenario solves sent out to workers in current iteration
- ScenarioIn: Number of scenario solves returned from workers in current iteration
- scenarioSolved[]: Length K boolean array, denoting if each scenario has been solved in current iteration
- PoolEval: number of solutions evaluated in current master iteration

We omit the complete description of the algorithm when optimality cuts and improvements are added. Instead, we describe the original asynchronous implementation with the following conditions. Whenever we denote an update to LB or UB, it is implied that the condition $UB > LB$ is checked to ensure that we have not reached optimality. The set S of cuts to add is managed in a way to avoid adding redundant cuts. This management is not reported in the algorithm below. When we **Reset** counters, the default value is zero for integers and FALSE for boolean. We also require two additional functions, defined immediately after Algorithm 2.

In Algorithm 2 we remove each of the Synchronization Points from Algorithm 1 by either creating new tasks for the worker processes or allowing workers to ignore the synchronization point and proceed to the next step in the algorithm. For example, the End Scenario Solve synchronization point waits for all the scenario solves to be completed in the Synchronous Algorithm (line 7 of Algorithm 1). In the Asynchronous Algorithm, when a worker returns with a Scenario Solve (line 10 of Algorithm 2), the master decides whether to ask the worker to solve a scenario again (lines 17-23) or switch over to evaluating solutions, based on whether all scenarios have been

Algorithm 2 Asynchronous Algorithm

Initialize: STATE: $\leftarrow$ *Master Solve Scenario*, $S = S_{ToEval} = \emptyset$
Reset scenarioSolved, ScenarioIn, ScenarioOut, PoolEval
for $w = 1$ to W **do**
  Send: $SOLVE\ SCENARIO(w, w, \emptyset)$
  ScenarioOut++
end for
while ($UB > LB \ || \ S_{ToEval} \neq \emptyset$) **do**
  Wait until worker w returns TaskType
  Read Information
    if (TaskType == $SOLVE\ SCENARIO(k, w, S)$) **then**
      $UpdateScenSolveCounter(k)$
      $S_{ToEval} \leftarrow S_{ToEval} \cup x^k$
    else if (TaskType == $EVALUATE\ SOLUTION(x^s, w)$) **then**
      Update UB
    end if
  Send Next Job
    if (STATE == *Master Solve Scenario*) **then**
      if (ScenarioOut $<$ K) **then**
        Send: $SOLVE\ SCENARIO$(ScenarioOut,w, S)
        ScenarioOut++
      else
        Send: $SOLVE\ SCENARIO(Rand(0, K), w, S \cup S_{ToEval})$
      end if
    else if (STATE == *Master Evaluate Solution*) **then**
      if (PoolEval == $|S_{ToEval}|$) **then**
        $UpdateEvalSolnCounter()$
      else
        Send: $EVALUATE\ SOLUTION(x^{PoolEval}, w)$
        PoolEval++
      end if
    end if
end while

function $UpdateScenSolveCounter(k)$
  if (scenarioSolved[k] == FALSE) **then**
    ScenarioIn++, scenarioSolved[k] = TRUE
    if (ScenarioIn == K) **then**
      STATE $\leftarrow$ Master Evaluate Solution
      Reset ScenarioIn, ScenarioOut, scenarioSolved
    end if
    Update LB
  end if
end function

1: **function** $UpdateEvalSolnCounter()$
2: STATE $\leftarrow$ Master Solve Scenario
3: Send: $SOLVE\ SCENARIO(0, w, \emptyset)$
4: ScenarioOut = 1
5: $S = S \cup S_{ToEval}$, $S_{ToEval} \leftarrow \emptyset$
6: **Reset** PoolEval
7: **end function**

solved at least once (line 4 of function $UpdateScenSolveCounter$). The Asynchronous Algorithm also eliminates the other synchronization points of the Synchronous Algorithm. We demonstrate the effectiveness of the Asynchronous approach in Section 5.

5 Results

We perform computational experiments using four sets of instances from [2] and [11]. The instance sets used are Stochastic Server Location Problem (SSLP), Stochastic Server Location Problem Replications (SSLPRep), Stochastic Supply Chain Problems (SSCh) and Stochastic Multiple Knapsack Problems (SMKP). In these experiments, we compare versions of our asynchronous implementation with the original synchronous implementation and the extensive formulation solved using default CPLEX.

5.1 Experiment Settings

All computations were performed on the Sierra Cluster at Lawrence Livermore National Laboratory. The cluster consists of 1,944 nodes, with nodes connected using InfiniBand Interconnect. Each individual node consists of 2 Intel 6-core Xeon X5660 processors and 24GB of memory. The asynchronous implementation was written in C using MVAPICH 2 version 1.7 for parallelism. Optimization problems were solved using CPLEX 12.5. Unless otherwise noted, each asynchronous run consists of 1 master and $ncores - 1$ worker processes, with each worker running single threaded CPLEX.

Five approaches are tested. In columns labeled CPLEX, we solve the extensive formulation of the problem using twelve threaded CPLEX with default settings. In columns labeled Synch, the synchronous implementation of our algorithm as described in Algorithm 1 is tested. In Asynch, we use our asynchronous implementation as described in Algorithm 2 without any improvements. In Asynch++, we present our results from our asynchronous implementation including improvements from Section 3.1 with the optimality cut described in 3.2 excluded. In Asynch+Cut, we consider all improvements from Section 3, including the optimality cut. Instances with a $*$ indicate that the number of workers/cores is limited to the number of scenarios.

The entries in each table of this section represent the average solution time in seconds over five replications when using our approaches and one replication when using CPLEX. CPLEX has been tested on the extensive formulation of these problems in many other papers. Our CPLEX experiment served to show that the performance of CPLEX for these instances on our architecture is not significantly different than what was previously reported. Instances are solved to a relative optimality gap of 10^{-4}. Each run is given a time limit of 3 hours. If the time limit is reached, the

average relative optimality gap is reported as $(x\%)T$. If the memory limit is reached, the average relative optimaliy gap is reported as $(x\%)M$. Each individual run uses 1 node, 12 cores and 24GB of total available memory, with the exception of the SSCh Asynchronous results, which use 2 nodes, 24 cores and 48 GB of total available memory.

5.2 Stochastic Server Location Problem (SSLP)

The Stochastic Server Location Problem is described in detail in [12]. It is a common benchmark problem for 0-1 stochastic programs. In this problem set, the first stage 0-1 variables represent placement of servers at given locations. The uncertainty is in potential customer demand for the service. Second stage recourse variables represent a decision to serve a particular customer from a given server or to not serve the customer at all. Instances can be found as part of SIPLIB [2], a repository of benchmark stochastic programs. Instances are written in the form $sslp\_n_1\_n_2\_K$, where n_1 is the number of server locations, n_2 is the number of customer locations and K is the number of scenarios.

	Seconds/(Relgap)				
Problem	CPLEX	Synch	Asynch	Asynch++	Asynch+Cut
sslp_10_50_50	80	15	10	5	4
sslp_10_50_100	1,045	28	15	9	9
sslp_10_50_500	(0.13%)M	121	74	32	41
sslp_10_50_1000	(0.17%)T	142	122	63	82
sslp_10_50_2000	(0.30%)T	312	291	143	193
sslp_15_45_5*	2	4	2	1	1
sslp_15_45_10*	2	17	9	6	4
sslp_15_45_15	16	29	15	7	11

Table 1: SSLP

As seen in Table 1, CPLEX performs well for those instances with a small number of scenarios, but performs very poorly as the number of scenarios increases. This is likely due to the fact that the extensive formulation is very large when a large number of scenarios are considered. The synchronous implementation represents a significant improvement over CPLEX, and the updated asynchronous implementation ranges from two to four times faster than the synchronous implementation. Adding the optimality cuts to the improved asynchronous implementation does not reduce solution time in many of the SSLP instances, and in some instances leads to increased solution time.

5.3 Stochastic Server Location Problem Replications (SSLPRep)

We also present the results of the Asynch++ and CPLEX implementations over a set of replications of the *SSLP* test set of instances called (SSLPRep). The instances can be found at [11].

	Seconds/(Relgap)					
	CPLEX					
Problem	a	b	c	d	e	Average
sslp_10_50_50	132	93	704	55	377	272
sslp_10_50_100	3,587	116	495	197	(0.01%)M	-
sslp_10_50_500	(4.93%)M	(0.18%)M	(0.16%)M	(0.02%)M	(11.2%)M	(3.30%)
sslp_10_50_1000	(4.79%)M	(0.24%)T	(9.57%)T	(0.04%)M	(13.74%)M	(5.82%)
sslp_10_50_2000	(1.43%)T	(0.29%)T	(11.56%)T	(0.14%)M	(6.98%)T	(4.09%)
sslp_15_45_5	2	1	5	2	3	3
sslp_15_45_10	13	5	7	9	1	7
sslp_15_45_15	114	44	52	346	9	113

Table 2: SSLP Replications - CPLEX

	Asynch++					
Problem	a	b	c	d	e	Average
sslp_10_50_50	5	3	4	2	6	4
sslp_10_50_100	8	5	6	4	8	6
sslp_10_50_500	32	23	23	25	35	27
sslp_10_50_1000	63	54	50	51	73	58
sslp_10_50_2000	151	115	123	124	169	136
sslp_15_45_5	2	3	122	15	8	30
sslp_15_45_10	16	22	203	100	-	85
sslp_15_45_15	95	8	112	33	7	51

Table 3: SSLP Replications - Asynch++

The results from our experiements the previous section are mostly confirmed. The problem structure has more of an impact on solution time for instances with a small number of scenarios as shown by the variabliity in solution time between the 15_45_$*$ instances as opposed to the solution times for the 10_50_$*$ instances.

5.4 Stochastic Supply CHain (SSCh) Planning

The Stochastic Supply CHain planning (SSCh) instances were first proposed in [3] and can be downloaded from [11]. The SSCh instances are two stage stochastic programs with 0-1 first stage variables and mixed binary second-stage variables. These instances model supply chain design with uncertainty in product demand and product/resource pricing. First stage decision variables represent plant location, plant size and allocation of product to plant decisions. The uncertainty is realized in product demand and price as well as resource cost. Second stage decisions involve processing decisions at individual plants, such as amount of product to process or store and product shipping decisions.

This test set of problems has been studied before in [3], [13] and [14] amongst others. The set consists of 9 instances (we could not access c9 online), with each instance comprised of 23 scenarios and 67-78 first stage binary variables. In the second stage, each scenario contains 36 binary

and approximately 3,000 continuous variables. The instance set is computationally challenging, evidenced by the fact that five of the nine instances were previously unsolved.

We present our computational results in Table 4. The column BKS represents the best known solution for each of the instances, all of which were known prior to this work. The column 'Previous' contains the best known relative optimality gaps from the literature. Note that we test CPLEX on the extensive formulation using only 12 cores. Our goal was not to directly compare our running time with that of CPLEX. Instead, we wanted to illustrate that the problems from this instance set remain challenging for CPLEX running on a single node.

		Seconds/(Absgap)			
Problem	BKS	Asynch++	Asynch+Cut	Previous	CPLEX
c1	184,489	(15,480)	(8,766)	(7,322)	(19,971)
c2	0	(16,422)	**110**	**OPT**	**40**
c3	230,368	(13,837)	(5,901)	(9,832)	(18,427)
c4	201,454	(12,202)	(3,523)	(7876)	(16,629)
c5	0	(7,820)	**1,150**	**OPT**	**2,080**
c6	231,368	(10,514)	(4,828)	(10,273)	(8,825)
c7	144,181	(9,649)	**682**	**OPT**	**169**
c8	100,523	(5,071)	**2,545**	(3,106)	(13,842)
c10	139,738	(1,732)	**126**	**OPT**	(1,259)

Table 4: SSCH

We are able to prove optimality in 5 out of 9 instances, solving one previously unsolved instance (c8) and improving the best known bounds for three additional problems (c2,c4,c6). We did encounter numerical stability issues in later iterations of the algorithm for certain instances in the SSCH instance set. Due to these issues, the results presented are run with conservative settings to ensure numerical stability. The extensive formulation of c1 also exhibted numerical stability issues when solved using CPLEX. We are able to find the best known first stage solution in the first iteration of the algorithm in every instance. This means that, for this instance set, the best known first stage solution is an optimal first stage solution for one of the original scenario problems.

5.5 Stochastic Multiple Knapsack Problem

The Stochastic Multiple Knapsack Problem can be found as part of [2] and are originally proposed in [4]. The set consists of thirty instances, each with 240 first and 120 second stage variables, found in 20 equiprobable scenarios. There are 50 first stage only constraints and 5 second stage constraints per scenario.

	Seconds/(Relgap)		
Problem	CPLEX	Asynch++	Asynch+Cut
smkp1-7	(0.14%)	3,524	2,291
smkp8-11,13	(0.19%)	(0.05%)T	6,213
smkp12,14-29	(0.19%)	(0.11%)T	(0.08%)T

Table 5: SMKP

We divide the instances into three sets, based on the ability of Asynch++ and Asynch+Cut to solve the instances to optimality. We solve seven instances to optimality when using Asynch++ and in twelve instances when the optimality cut is added. Additionally, in every instance which Asynch+Cut does not solve the problem to optimality, the optimality cut improves the final relative gap obtained when compared with Asynch++.

These instances have many more constraints and variables in the first stage than in the second stage. As our algorithm uses a scenario decomposition approach, these instances may be especially challenging for our algorithm when compared to other stagewise decomposition approaches. Additionally, we rely on the solution of individual scenario problems via optimization software such as CPLEX, as opposed to stagewise decomposition approaches such as the Integer L-Shaped Method. An average of two master iterations are completed using Asynch++ on the SMKP instance set, suggesting that for SMKP, the individual scenario problems are especially challenging for CPLEX to solve.

5.6 Computational Results - Scaling

As our implementation is designed to take advantage of distributed memory systems, we measure the improvements in solution time as more computational resources become available. We use the SSLP set of test instances to perform two scaling experiments.

In Table 6 we test the scaling of our algorithm compared to CPLEX on a single node. We vary the number of cores available from two to twelve cores, where twelve is the total number of cores on one node. Our Asynch++ algorithm uses 1 master and $ncores - 1$ workers, while we run CPLEX on the extensive formulation using *ncores* CPLEX threads. Given more threads, CPLEX searches the Branch-and-Bound (B&B) tree with multiple threads. We see that for the SSLP set of test instances, our Asynch++ scales much more efficiently than the default CPLEX implementation. Furthermore, it scales monotonically, as opposed to CPLEX, where the number of the threads is a poor predictor of performance. A detailed explanation of the challenges involved and reasons for loss of efficiency as the number of parallel threads in a B&B tree search in increased can be found in [9]. Some of the apparent superlinear speedup of the Asynch++ algorithm is explained by the fact that we test the scaling with respect to the total number of cores used, not the total number of worker processes. Additionally, with more than one worker, some workers can move past former synchronization points to improve bounds while other workers complete solution evalautions.

In Table 7, we test two scaling possibilities for our implementation of Asynch++. We begin both experiments using Asynch++ using 1 master and 4 workers. In the New Worker experiment, as we add cores, we add new workers. In the Multiple Threads experiment, as we add cores, we allow each worker to spawn multi threaded CPLEX. For example, using Multiple Threads with 8 worker cores, we allow each worker to spawn CPLEX with two threads. The results are presented in Table 7 below. We see that for the $sslp\_n_1\_n_2\_K$ instance class that the New Worker scheme uses additional resources much more efficiently than the Multiple Threads approach. This supports the results from the previous experiments: for stochastic programs with many scenarios and relatively small scenario problems, it is likely to be more efficient to add new worker processes as opposed to giving CPLEX more resources.

		Cores					
Problem	Type	2	4	6	8	10	12
sslp_10_50_50	CPLEX	152	498	2835	116	150	80
sslp_10_50_50	Asynch++	47	13	7	7	5	4
sslp_10_50_100	CPLEX	512	4,011	1,579	1,334	313	1,045
sslp_10_50_100	Asynch++	81	26	14	11	9	7
sslp_10_50_500	CPLEX	(0.21)M	(0.22)M	(0.22)T	(4.75)M	(0.51)T	(0.13)M
sslp_10_50_500	Asynch++	322	99	60	45	36	32
sslp_10_50_1000	CPLEX	(0.18)T	(0.18)T	(0.23)M	(0.22)T	(0.17)M	(0.17)T
sslp_10_50_1000	Asynch++	621	188	115	87	72	61
sslp_10_50_2000	CPLEX	(0.34)T	(0.31)T	(0.29)T	(0.30)T	(0.32)T	(0.30)T
sslp_10_50_2000	Asynch++	1297	404	254	194	160	144

Table 6: Scaling: CPLEX vs Asynch++

		Worker Cores				
Problem	Type	4	8	12	24	48
sslp_10_50_50	Multi	10	9	9	9	12
sslp_10_50_50	New Wrk	10	5	4	4	4
sslp_10_50_100	Multi	18	17	16	16	20
sslp_10_50_100	New Wrk	18	10	7	4	3
sslp_10_50_500	Multi	76	69	71	70	85
sslp_10_50_500	New Wrk	76	39	29	19	12
sslp_10_50_1000	Work	139	134	136	133	173
sslp_10_50_1000	New Wrk	139	75	57	36	26
sslp_10_50_2000	Work	307	300	298	297	391
sslp_10_50_2000	New Wrk	307	166	126	81	55

Table 7: Scaling: Worker vs CPLEX Thread

6 Conclusion

In this work, we have developed performance improvements and an asynchronous implementation of the scenario decomposition algorithm proposed in [1]. These improvements were demonstrated via results on the SSLP, SSLPRep, SSCh and SMKP instances from [2] and [11]. We demonstrated that the given scenario decomposition algorithm outperforms CPLEX on the instances in question and its relative performance improves as the ratio of number of first stage variables to number of scenarios decreases. We also show that the scenario decomposition algorithm can possibly be improved by reducing the time taken to solve deterministic scenario problems. Finally, through the addition of the optimality cut, we demonstrate that our algorithm can be improved upon by incorporating other known cut generation schemes. We believe that the incorporation of other schemes into the scenario decomposition algorithm may lead to further improvement.

7 Acknowledgements

Some of this work is performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. This research also has been supported in part by the Office of Naval Research Grant N00014-15-1-2078.

References

[1] AHMED, S. A scenario decomposition algorithm for 0-1 stochastic programs. *Operations Research Letters 41*, 6 (2013), 565 – 569.

[2] AHMED, S., GARCIA, R., KONG, N., NTAIMO, L., PARIJA, G., QIU, F., AND SEN, S. A stochastic integer programming test library. `http://www2.isye.gatech.edu/~sahmed/siplib/`, 2015.

[3] ALONSO-AYUSO, A., ESCUDERO, L., GARIN, A., ORTUNO, M., AND PEREZ, G. An approach for strategic supply chain planning under uncertainty based on stochastic 0-1 programming. *Journal of Global Optimization 26*, 1 (2003), 97–124.

[4] ANGULO, G., AHMED, S., AND DEY, S. Improving the integer L-shaped method. *Optimization Online* (2014).

[5] ARAVENA, I., AND PAPAVASILIOU, A. A distributed asynchronous algorithm for the two-stage stochastic unit commitment problem. In *Power Energy Society General Meeting, 2015 IEEE* (July 2015), pp. 1–5.

[6] CAROE, C., AND SCHULTZ, R. Dual decomposition in stochastic integer programming. *Operations Research Letters 24* (1997), 37–45.

[7] CRAINIC, T. G., FU, X., GENDREAU, M., REI, W., AND WALLACE, S. W. Progressive hedging-based metaheuristics for stochastic network design. *Networks 58*, 2 (2011), 114–124.

[8] KIM, K., AND ZAVALA, V. M. Algorithmic innovations and software for the dual decomposition method applied to stochastic mixed-integer programs. *Optimization Online* (2015).

[9] KOCH, T., RALPHS, T., AND SHINANO, Y. Could we use a million cores to solve an integer program? *Mathematical Methods of Operations Research 76* (2012), 67–93.

[10] LKKETANGEN, A., AND WOODRUFF, D. Progressive hedging and tabu search applied to mixed integer (0,1) multistage stochastic programming. *Journal of Heuristics 2*, 2 (1996), 111–128.

[11] NTAIMO, L. Stochastic mixed-integer programming test problems. `http://ise.tamu.edu/people/faculty/ntaimo/personal_web/test_instances.htm`, 2015.

[12] NTAIMO, L., AND SEN, S. The million-variable march for stochastic combinatorial optimization. *Journal of Global Optimization 32*, 3 (2005), 385–400.

[13] NTAIMO, L., AND SEN, S. A comparative study of decomposition algorithms for stochastic combinatorial optimization. *Computational Optimization and Applications 40*, 3 (2008), 299–319.

[14] NTAIMO, L., AND TANNER, M. Computations with disjunctive cuts for two-stage stochastic mixed 0-1 integer programs. *Journal of Global Optimization 41*, 3 (2008), 365–384.

[15] SANTOSO, T., AHMED, S., GOETSCHALCKX, M., AND SHAPIRO, A. A stochastic programming approach for supply chain network design under uncertainty. *European Journal of Operational Research 167*, 1 (2005), 96 – 115.

[16] WATSON, J.-P., AND WOODRUFF, D. Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Computational Management Science 8*, 4 (2011), 355–370.