Parallel Branch-and-Bound for Two-Stage Stochastic Integer Optimization

Akhil Langer‡, Ramprasad Venkataraman‡, Udatta Palekar*, Laxmikant V. Kale‡
‡Department of Computer Science, *College of Business
University of Illinois at Urbana-Champaign
{alanger, ramv, palekar, kale}@illinois.edu

***Abstract*---Many real-world planning problems require searching for an optimal solution in the face of uncertain input. One approach to is to express them as a two-stage stochastic optimization problem where the search for an optimum in one stage is informed by the evaluation of multiple possible scenarios in the other stage. If integer solutions are required, then branch-and-bound techniques are the accepted norm. However, there has been little prior work in parallelizing and scaling branch-and-bound algorithms for stochastic optimization problems.**

In this paper, we explore the parallelization of a two-stage stochastic integer program solved using branch-and-bound. We present a range of factors that influence the parallel design for such problems. Unlike typical, iterative scientific applications, we encounter several interesting characteristics that make it challenging to realize a scalable design. We present two design variations that navigate some of these challenges. Our designs seek to increase the exposed parallelism while delegating sequential linear program solves to existing libraries.

We evaluate the scalability of our designs using sample aircraft allocation problems for the US airfleet. It is important that these problems be solved quickly while evaluating large number of scenarios. Our attempts result in strong scaling to hundreds of cores for these datasets. We believe similar results are not common in literature, and that our experiences will feed usefully into further research on this topic.

I. Introduction

This paper presents our parallel algorithms for scalable stochastic integer optimization. Specifically, we are interested in problems with integer solutions, and hence, in Branch-and-Bound (BnB) approaches. Although BnB is a well-studied method, there has been little prior work in parallelizing or scaling two-stage, stochastic Integer Programs (IPs).

We structure this paper to expose the design influences on parallel solutions of stochastic IPs. Once past the introductory sections (II--IV), we present our approach to parallelizing stochastic IPs (V), and discuss the factors we considered while designing a parallel BnB for such optimization problems (VI). This section presents some of the challenges that set this problem apart from typical parallel computational science applications. We pick a programming model that enables the expression and management of the available parallelism in section VII. Finally, we present two primary design variations (VIII and IX), and analyze their performance in section X.

The context for our work is a US fleet management problem where aircraft are allocated to cargo movement missions under uncertain demands (III). However, the design discussions and parallelization techniques are not specific to it.

II. Two-stage Stochastic Integer Optimization

In real world situations, future outcomes are often dependent on factors that cannot be deterministically predicted (e.g: weather in agriculture, product demands in the manufacturing, stock prices for an investor). However, resources have to be allocated before these uncertain influences become known. When resource use has to be optimized under such conditions, the problem falls under the purview of stochastic optimization. Unlike deterministic programming, stochastic programming explicitly incorporates uncertain parameters by assuming a probabilistic distribution to make a more rational decision for optimal resource allocation. Application of stochastic programming spans a diverse set of fields ranging from production, financial modeling, transportation (road as well as air), supply chain and scheduling to environmental and pollution control, telecommunications and electricity.

In multi-stage stochastic programs, decisions are made in multiple stages. For example, in portfolio management, a fixed amount of cash available at time t_0 has to be invested across times $t_1, t_2, ..t_n$. Decisions taken at time t_i will depend on the decisions/outcomes from t_{i-1}. Unlike the case of portfolio management in which the unknown parameters are realized over a sequence of stages, in two-stage stochastic programs, all the unknown parameters are realized in a single stage. In the first stage, strategic decisions are made (the known resources are allocated to the different fields of activities) and in the second stage operational decisions are made for every scenario. A specific instantiation of the unknown parameters is called a scenario. Most applications can be formulated as two-stage programs. Multi-stage programs can be solved by conversion to two-stage programs or by applying the nested version of the methods used to solve two-stage programs. Therefore we focus our work on two-stage stochastic programs.

Two-stage stochastic optimization is commonly solved using Benders decomposition [1], where candidate solutions are generated in Stage 1 and are evaluated in Stage 2 for every scenario (Figure 1). Stage 1 (Eq.1) gets *feedback* from Stage 2 (Eq.2) in the form of *cuts* (Eq.3), which are used by the Stage 1 to improve the candidate solution. The process iterates until no improvement can be made.

$$\min \quad cx + \sum_{s=1}^{s} p_s \theta_s \quad s.t.\ Ax \leq b \tag{1}$$

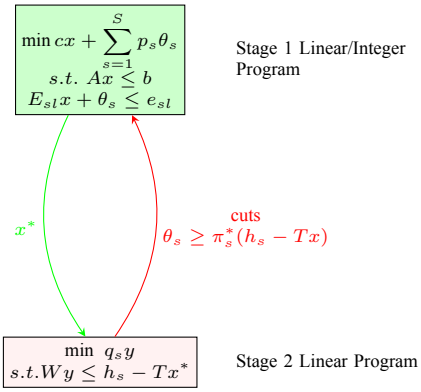


Fig. 1. Benders decomposition for 2-stage stochastic programs

$$\theta_s = \quad \min({q_s}^T y) \quad s.t.\ Wy \leq h_s - T_s x \tag{2}$$
$$\theta_s \geq \quad \pi_s^*(h_s - Tx^*) \tag{3}$$

where, x is the candidate solution, c is the cost coefficient vector, $\theta = \{\theta_s | s = 1..S\}$ are the Stage 2 costs for the S scenarios, p_s is the probability of occurrence of scenario s, π_s^* is the optimal dual solution vector for Stage 2 Linear Program (LP) of scenario s. This method is also called the multicut L-shaped method [2] in which one cut per scenario is added to the Stage 1 in every iteration/round.

We restrict our work to the problems in which Stage 2 has only linear variables. When only linear variables are present in Stage 1 also, we call it a stochastic LP. And when Stage 1 has integer variables, we call it a stochastic IP or a stochastic Mixed Integer Program (MIP). Louveaux and Schultz [3], Sahinidis [4], in the broader context of decision-making under uncertainty, give excellent overviews of stochastic IP problems.

III. Case Study: Military Aircraft Allocation

The US Air Mobility Command manages a large fleet of aircraft that are assigned to cargo and personnel movement missions. These missions operate under varying demands and experience sudden changes. The aim is to plan for an upcoming time period by accounting for the uncertainty in upcoming demands and allocating aircraft to missions such that the overall costs are minimized. The uncertainty in demands definitely puts this problem in the class of stochastic programs. Integer solutions are required because aircraft need to be dedicated completely to individual missions.

We use representative datasets that model this problem. They are classified based on the number of time periods (days) in the planning window and the number of possible scenarios that need to be evaluated to account for the uncertainty. The sizes of the LPs are given in Table I. For e.g., the 5t-120 dataset has approximately 135 integer variables in the Stage 1 IP, $1.6M$ variables in the Stage 2 LP, and about $1M$ Stage 2 constraints when evaluating 120 Stage 2 scenarios. Similarly, 3t-240 stands for the 3t model with 240 scenarios, and so on.

TABLE I
jetAlloc datasets: Stage 1 IP and Stage 2 LP sizes

Test	1st Stage		2nd-Stage Scenario		Nonzero Elements		
Problem	Vars.	Constrs.	Vars.	Constrs.	A	W_i	T_i
2t	54	36	6681	4039	114	20670	84
3t	81	54	8970	5572	171	27991	88
4t	108	72	11642	7216	228	36422	140
5t	135	90	13862	8669	285	43518	168
8t	216	144	20944	13378	456	66881	252
10t	270	180	25573	16572	570	82797	308

These models can be downloaded in SMPS[1] format from our website[2].

IV. Prior Work

Parallelizing IP optimizations using BnB is in itself a challenging problem. Large scale solvers for Mixed Integer Programs (MIPs) have been studied before [5], [6]. The difficulty in achieving high efficiencies has been documented. Kale et al [7] have studied the challenges of dynamic load balancing in parallel tree search implementations. Gurobi [8] has a state-of-the art mixed integer program solver that exploits multi-core architectures. However, Koch et al in [6] observe that Gurobi suffers from poor efficiency (typically about 0.1) as it scales from 1 to 32 threads, the reason being that the number of BnB vertices needed to solve an instance varies substantially with different number of threads.

Our work involves optimization of stochastic IPs, which have decomposable program structure and large size. It presents further challenges that make it even harder to parallelize than just IPs. Examples of the uses of stochastic integer programming can be found in literature. Bitran et al [9] model production planning of style goods as a stochastic mixed IP. Dempster et al [10] consider heuristic solutions for a stochastic hierarchical scheduling problems. A comprehensive listing of work on stochastic IPs can be found here [11].

A stochastic program can be solved using its extensive formulation, which is its deterministic equivalent in which variables and constraints from all the scenarios are combined together in a single large LP. This LP can then be fed to any of the several open or commercial LP/IP solvers. However, Escudero et al [12] note that MIP solvers such as CPLEX [13] do not provide solution for even toy instances of two stochastic IPs in a viable amount of time.

We have not found systematic studies of large-scale stochastic integer optimization in literature. . PySP [14], [15] is a generic decomposition-based solver for large-scale multistage stochastic MIPs. It provides a Python based programming framework for developing stochastic optimization models. For the solution of the stochastic programs, it comes with parallel implementations of algorithms such as Rockafellar and Wets' progressive hedging. These tend to be heuristic algorithms that require substantial parameter tuning. To the extent of our knowledge, the computational and scaling behavior of this framework have not been explored and the solver suffers

[1] http://myweb.dal.ca/gassmann/smps2.htm
[2] http://ppl.cs.illinois.edu/jetAlloc/

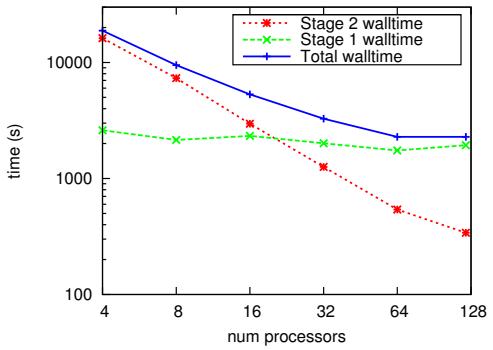


Fig. 2. Scaling limited by Amdahl's law in a master-worker design for stochastic linear optimization (earlier work). Results for 10t-1000 model obtained on Abe (dual quad-core 2.33GHz Intel Clovertown nodes with GigE)

from poor parallel efficiency because of MIP solve times. Recent work of Lubin et al [16] is based on parallelizing the dual decomposition of Stage 1 integer program by using interior-point solvers. Their study is limited to 32 cores and the approach suffers from load imbalance.

V. Parallelization Approach

Two-stage stochastic optimization problems have a natural expression in a two-stage software structure. The first stage proposes candidate solutions and the second stage evaluates multiple scenarios that helps refine the solution from the first stage. In earlier work [17] on stochastic LP, we focused on an iterative, two-stage master-worker design for the solution of stochastic linear programs. This tapped the readily available parallelism in Stage 2 by evaluating multiple possible scenarios simultaneously (3a). Although such a design captured much of the low-hanging, easily exploitable parallelism, it was quickly limited by the serial bottleneck of performing Stage 1 computations (Figure 2).

In contrast to earlier work, this paper focuses on the solution of stochastic integer programs (IP), which requires that Stage 1 solve an IP for every iteration. Since solving an IP is much more computationally expensive than an LP, this will magnify the serial bottleneck of the master-worker design such that it becomes completely untenable. Thus, it is imperative to reduce and hide this sequential bottleneck by exposing more parallelism.

Our approach to parallelizing stochastic IPs is by using BnB to obtain integer solutions to Stage 1 variables. We start by relaxing the integrality constraints in Stage 1 and solve the stochastic LP. BnB proceeds by branching on fractional parts of a solution obtained from the stochastic LP and restricting each branch to disjoint portions of the search space until gradually all variables in the solution become integral. This yields a tree where each vertex has one additional constraint imposed on the feasible space of solutions than its parent. We find a solution to this additionally constrained two-stage stochastic LP at this vertex, and then continue to branch. Therefore, each vertex in our BnB tree is a stochastic LP. The stochastic LP at each vertex permits evaluating each of the multiple scenarios in parallel. Additionally, the BnB search for integer solutions permits exploring the disjoint portions of the search space (i.e. the tree vertices) in parallel. Thus there are two sources of parallelism - simultaneous evaluation of Stage 2 scenarios and the simultaneous exploration of BnB tree vertices. This nested parallelism has to be exploited for any reasonable scalability.

A relevant observation that influences processor utilization is the mutual exclusivity of the two stages of the stochastic programs. For a given vertex, Stage 1 cannot proceed while it is waiting for feedback from Stage 2, and Stage 2 is necessarily dependent on Stage 1 for each new candidate solution. Ensuring high utilization of compute resources will therefore require interleaving the iterative two-stage evaluation of multiple BnB vertices. This is also what makes this application distinct from the traditional applications of BnB. In traditional applications of BnB such as integer programming, traveling salesman problem (TSP), game tree search algorithms, etc. each tree vertex is an atomic unit of work i.e. when a vertex is processed it is either pruned or tagged as an incumbent solution or branches to generate children. No further processing of that vertex is required. On the other hand, in our application, each tree vertex is a stochastic LP optimization and therefore can require multiple rounds of Stage 1 and Stage 2 computations for optimization. While a vertex is being processed in Stage 2, its Stage 1 state has to be saved, so that it can be retrieved for the next Stage 1 computation (which will happen when the corresponding current Stage 2 finishes).

VI. Design Considerations

A. Coarse-Grained Decomposition

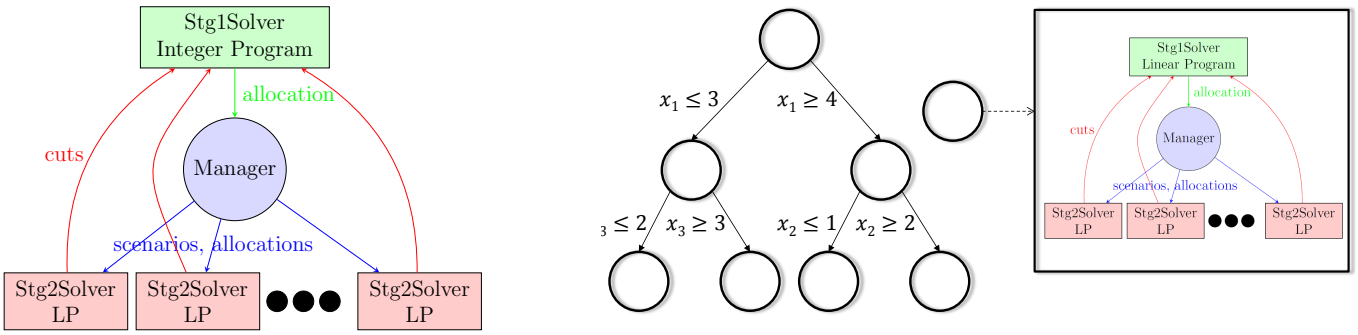
In our designs, we choose to delegate sequential LP solutions in Stage 1 and Stage 2 to an existing optimization library. This allows us to leverage the expertise encapsulated in these highly tuned libraries and focus on the parallelization and accompanying artifacts. Hence, the fundamental unit of sequential computation in our designs is a single linear program solve. This results in very coarse grain sizes.

B. Unpredictable Grain Sizes

There is sizeable variation in the time taken for an LP solve in both Stage 1 and Stage 2. Additionally, there is no persistence in the time taken for LP solves. A single Stage 1 LP for a given vertex may take widely varying times as a result of the addition of a few cuts from Stage 2. Likewise, we do not observe any persistence in Stage 2 LP solve times either across different scenarios for a given Stage 1 candidate solution, or for the same scenario across different candidate solutions. An illustrative execution profile is presented in Figure 4.

C. Varying Amounts of Available Parallelism

The BnB tree exposes a varying amount of parallelism as the search for an optimum progresses. The search starts with a single vertex (the tree root) being explored. More parallelism is gradually uncovered in a ramp-up phase, as each vertex branches and creates new vertices. However, once candidate integer solutions are found, the search tree can be pruned to avoid unnecessary work. For large enough search trees, there is



(a) Naive parallelisation with Benders decomposition

(b) Nested parallelism with Branch-and-Bound and Benders decomposition

Fig. 3. Exploiting nested prallelism in stochastic integer programs

Fig. 4. Sample execution profile of evaluating multiple Stage 2 scenarios for candidate Stage 1 solutions. Each processor (horizontal line) is assigned a specific Stage 2 scenario, and evaluates multiple candidate solutions from Stage 1 one after the other. Colored bars represent an LP solve, while white stretches are idle times on that processor. LP solve times vary significantly and show no persistence, both across scenarios and across candidate solutions.

usually a middle phase when there are a large, but fluctuating number of vertices on the exploration front depending on branching and pruning rates. Once the optimum is found, the remaining work involves proving its optimality by exploring the tree until all other vertices are pruned. Towards the end, pruning starts to dominate and the front of exploration shrinks rapidly. Any parallel design has to necessarily cope with, and harness these varying levels of available concurrency.

D. Load Balance

The utter lack of persistence in the sizes of the sequential grains of computation and the constantly varying amount of available parallelism imply that a static *a priori* partition of work across different compute objects (or processors) will not ensure high utilization of the compute resources. It also precludes the use of any persistence-based dynamic load balancing solutions. Hence, our designs adopt pull-based or stealing-based load balancing techniques to ensure utilization. To avoid idle time, a parallel design must maintain pools of available work that can be doled out upon pull requests.

E. Solver Libraries Maintain Internal State

Unlike other numerical libraries, LP solvers maintain internal state across calls. They maintain the optimal basis of the previous problem that was solved. Most use cases for such solvers involve iterating over a problem with repeated calls to the library. Typically, each call supplies only mildly modified inputs as compared to the previous invocation. In such cases, the search for an optimum can be greatly sped up by starting from the previous solution. Hence, it is highly desirable to retain this internal state across calls as it greatly shortens the time to solution. This is known as a ``warm" start or ``advanced" start.

The two-stage optimization problems that interest us follow this pattern too. There are many iterations (rounds) to converge to a solution. In Stage 1, each iteration only adds/deletes a few constraints on the feasible search space. In Stage 2, the coefficient matrix of the LP remains the same, and only the right-hand sides of the constraints are modified across calls. A more detailed discussion on the impact of advanced starts can be found in [17].

Hence, it is beneficial to (*a*) allow all the solver library instances in the parallel execution to maintain state across calls and, (*b*) to maintain an affinity between the solvers and the problems that they work on across iterations. It is desirable to pick a parallel programming paradigm that will permit encapsulating and managing multiple solver instances per processor.

F. Concurrency Limited by Library Memory Footprint

The lowest levels of the BnB tree that have not been pruned constitute the ``front" of exploration. The number of vertices on this front at any given instant represents the maximum available concurrency in exploring the tree. Each vertex on this front represents a unique combination of branching constraints. Since each vertex goes through multiple iterations (rounds), it is desirable to exploit warm starts for each vertex. This can be achieved by assigning one solver instance for each vertex that is currently being explored. However, LP solvers have large memory footprints. The memory usage required for a LP solver instance for 3t, 5t, 10t, 15t are 50MB, 100MB, 230MB, 950 MB, respectively in Stage 1 and 10MB, 15MB, 30MB, 45MB, respectively in Stage 2. This implies that the number of solver instances is limited by available memory, and can be substantially smaller than the number of vertices in a large BnB search tree.

The actual subset of vertices on the front that are currently being explored are known as ``active'' vertices. The parallel design should account for the memory usage by solver instances, carefully manage the number of active vertices, and expose as much parallelism as permitted by memory constraints.

G. Stage 2 Feedback Can Be Shared Across the BnB Tree

While the set of branching constraints for each vertex are unique to it, the cut constraints from Stage 2 are not. The branching constraints influences the candidate allocations that are generated in Stage 1. These, in turn, only affect the right hand sides in the Stage 2 LPs, which simply alters the objective function in dual of the Stage 2 LP. The dual polytope of the Stage 2 LPs remains the same across all the vertex in the BnB tree. This implies that the dual optimal solutions obtained in Stage 2 for a candidate solution from the Stage 1 LP of a given vertex, are all valid dual extreme points for any vertex in the BnB tree. Hence, the Benders cuts that are generated from the Stage 2 LPs remain valid irrespective of the branching constraints imposed on a vertex, implying that cuts generated from evaluating scenarios for a given vertex are also valid for all vertices in the BnB tree.

This observation provides a powerful solution to increasing the exposed parallelism while remaining within the memory usage constraints. Since cuts can be shared across vertices, two vertices only differ in the branching constraints unique to them. By applying this delta of branching constraints, a Stage 1 LP solver instance can be reused to solve a Stage 1 LP from another vertex. Solver libraries typically expose API to add / remove constraints. Hence, it becomes possible to reuse a single solver instance to interleave the exploration of multiple BnB vertices. We can simply remove branching constraints specific to the vertex that was just in a Stage 1 LP solve, and reapply constraints specific to another vertex that is waiting for such a Stage 1 solve. This permits exploring more vertices than the available number of solver instances, and also retains the ability to exploit warm starts for Stage 1 LP solves.

The reasoning presented here also implies that the same Stage 2 solver instance can evaluate scenarios across multiple vertices. Hence, we can share both Stage 1 and Stage 2 solvers.

H. Total Amount of Computation is Variable and Perturbable

The total amount of computation performed to complete the BnB exploration depends on the number of BnB vertices explored and the number of Stage 1--Stage 2 rounds for each vertex. Unlike traditional iterative HPC algorithms, this total work required is variable and not known a priori. This is compounded by the fact that the shape and size of the BnB tree is easily perturbed. The number of vertices explored depends on the branching and pruning decisions during the exploration. Any factor that affects these decisions can alter the time to solution.

1) Incumbent Ordering: A parallel exploration of the BnB tree implies that even if the explored trees are identical across two runs, the order in which incumbent solutions are generated can vary slightly because of LP solve times, system noise, network interference in message communication, etc. This order affects the pruning of vertices from the tree. Some cases might even cause a slightly worse incumbent to prune a vertex that would have yielded a slightly better incumbent (but within the pruning threshold) simply because the worse incumbent was generated slightly early on another processor.

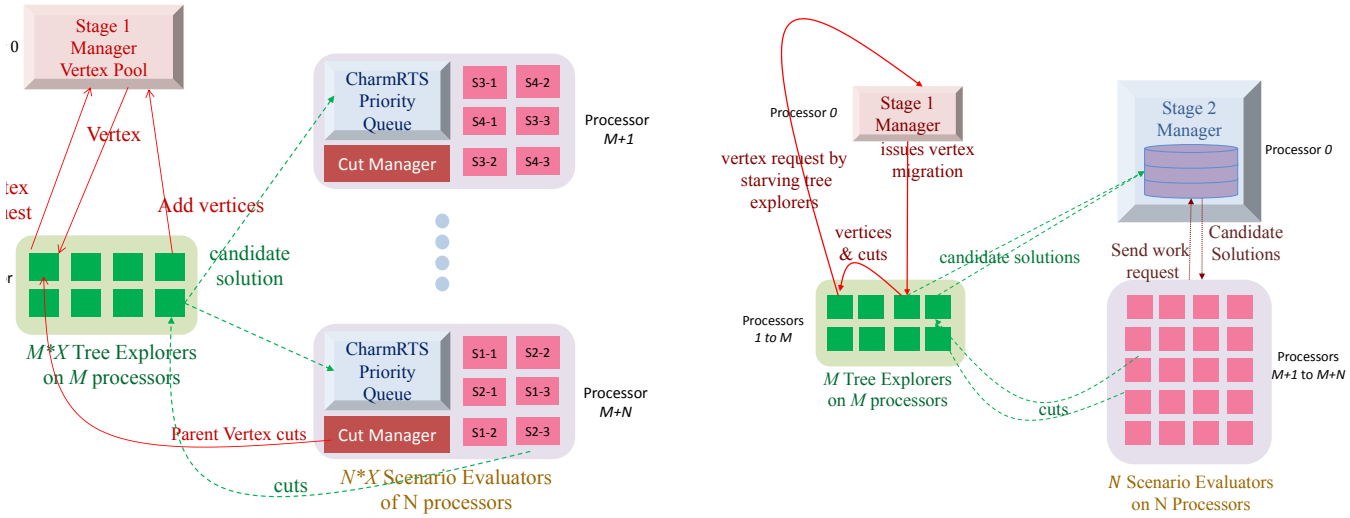
2) Degeneracy: Degeneracy occurs when the same extreme point on the feasible space polytope can be represented by several different bases. When this happens at the optimal extreme point, there can multiple dual optimal solutions. LPs often have degenerate solutions. While solving LPs, depending upon the starting point of the simplex method, one can end up with different solutions. If we share solver resources in an attempt to circumvent memory limitations, we cause an LP solve to start with an internal state that was the result of the previous LP solve for a different vertex. Thus, sharing solvers can yield different solutions to an LP depending on the order in which vertices use the shared LP solver instance. This can happen in both Stage 1 and Stage 2. Different LP solutions can impact the branching decisions under that vertex in the BnB tree. This reasoning implies that sharing LP solver instances can lead to different BnB tree structures.

I. Better Utilization $\neq$ Better Performance

For many parallel, HPC applications, load balance ensures minimal overall compute resource idle time, and hence results in better performance by maximizing the rate of computations. However, parallel, BnB search confounds such thinking. Indeed, reducing idle time by eagerly exploring as much of the tree as possible might be counter-productive by using compute resources for exploring sub-trees that might have been easily pruned later.

VII. Programming Model

The designs that we discuss here are implemented in an object-based, sender-driven parallel programming model called Charm++ [18], [19] . Charm++ is a runtime-assisted parallel programming framework in C++. Programs are designed using C++ constructs by partitioning the algorithm into classes. Charm++ permits elevating a subset of the classes and methods into a global space that spans all the processes during execution. Parallel execution then involves interacting collections of objects, with some objects and methods being invoked across process boundaries. Data transfer and messaging are all cast in the form of such remote method invocations. Such remote methods are always one-sided (only sender initiates the call), asynchronous (sender completes before receiver executes method), non-blocking (sender's side returns before messaging completion) and also do not return any values (remote methods are necessarily of void return type). Charm++ supports individual instances of objects, and also collections (or chare arrays of objects). Some features of Charm++ that enable the designs discussed in this paper:



(a) Design A: Every vertex in the BnB tree performs its own iterative, two-stage linear optimization in isolation from other vertices. There are X Tree Explorers on every Stage 1 processor. S1-3 corresponds to the Scenario Evaluator for scenario 1 of Tree Explorer 3, and similarly others.

(b) Design B: BnB vertices share resources (Stage 1 and Stage 2 solver objects) and constraints on the feasible solution space (cuts) while iteratively solving the nested two-stage linear programs.

Fig. 5. Schematics of the parallel components in the two design variants

a) One-sided messaging: helps express and exploit the synchronization-free parallelism found in parallel BnB. Extracting performance in a bulk synchronous programming model can be quite challenging.

b) Object-based expression: of designs facilitate the easy placement and dynamic migration of specific computations on specific processors. It also permits oversubscribing processors with multiple objects to hide work-starvation of one with available work in another.

c) Non-blocking reductions: for any required data collection, notifications etc avoids any synchronization that could be detrimental to performance. A programming model well suited to such problems, should unlock all the available parallelism without bridling it with synchronization constructs.

d) Prioritized execution: allows us to simply tag messages with appropriate priorities and allow the Charm++ runtime system to pick the highest priority tasks from the available pool.

VIII. Design A: Each BnB Vertex is an Isolated Two-Stage LP

A. Stage 1 Tree Explorers

A collection of compute objects (chare array in Charm++) explore the BnB tree in parallel. Each Tree Explorer hosts an instance of the Gurobi LP library. Tree Explorers are constrained to explore only one vertex at a time. Whenever a new vertex is picked, the library instance is reset and reloaded with a known collection of cuts from an ancestor vertex. When the vertices are waiting on Stage 2 feedback, the Tree Explorer idles. The processors dedicated to exploring the tree are oversubscribed by placing multiple Tree Explorers on each. The Charm++ runtime automatically overlaps idle time in one object with computation in another object by invoking any objects which are ready to compute. In the situation when multiple objects on a processor are ready to compute, execution is prioritized according to the search policy. This is indicated to the Charm++ runtime by tagging the messages with a priority field. This field can be an integer (tree depth), a fraction (bounds / cost), or a bitvector (vertex identifier).

B. Cut Dump Manager

Solving stochastic LP at each vertex from scratch can be very expensive as this potentially repeats a lot of avoidable Stage 1--Stage 2 rounds to regenerate all the cuts that would have been generated by vertex's ancestors. Each vertex, therefore, starts with the cuts of its parent. This significantly reduces the number of rounds required to optimize the stochastic LPs.

We precompute the available memory on the system and corral a portion of it for storing dumps of cut collections. Whenever a vertex converges, we extract its collection of cuts from the library instance and store it in the available memory. The dump is tagged with the bitvector id of the vertex. Whenever an immediate child of this vertex is picked for exploration, the parent's cut collection is retrieved and applied to the library instance. Once both children of a vertex are explored, the parent's dump is discarded. Hence, at any given time, the number of cut dumps stored is a linear function of the number of vertices on the tree frontier. The cut collection dumps are managed by a third chare collection called the Cut Manager. Objects of this collection are not placed on processors with Tree Explorers in order to keep them reasonably responsive to requests.

C. Scenario Evaluators

Akin to the Tree Explorers, the Scenario Evaluators are a collection of compute objects each of which hosts an LP

instance. These evaluate the candidate solutions for one or more scenarios and send the generated cuts directly back to the Tree Explorer that hosts the specific BnB vertex. We dedicate a collection of Scenario Evaluators to each Tree Explorer. Each Tree Explorer object interacts directly with its collection of Scenario Evaluators. We place these multiple collections of Scenario Evaluators on the same subset of processors. Idle time in one is overlapped with computation in another. The execution of Stage 2 computations for the most important vertices is again achieved by simply tagging the messages with the priorities of the corresponding vertices.

D. Load Balancing

When a Tree Explorer converges to an LP solution on a vertex, on its currently assigned vertex, further work is generated only if the vertex branches. In this case, the children are deposited with the Stage 1 Manager vertex queue. After every Stage 1 LP convergence, the Tree Explorer requests the Stage 1 Manager for a new vertex to work on. The Stage 1 Manager dequeues the highest priority vertex from its vertex queue and sends it to requesting Tree Explorer. Thus all Tree Explorers always pull from a global pool of available work. This effectively balances Stage 1 load and also ensures a globally prioritized tree exploration.

IX. Design B: BnB Vertices Share Cut Constraints, Tree Explorers and Scenario Evaluators

A. Stage 1 Tree Explorers

Each Tree Explorer object stores and explores several vertices. The vertices are divorced from the library instance by separately storing the set of branching constraints specific to each vertex. Every object maintains a set of private vertex queues to manage the vertices in different stages of their lifespan. When the LP library completes a solve, the next vertex is picked from a ``ready'' queue. This queue is prioritized according to the search policy (depth-first, most-promising-first, etc). The delta of branching constraints between the previously solved vertex and the currently picked vertex is applied to the LP library to reconstruct the Stage 1 LP for the newly selected vertex. The Stage 1 LP is then solved to yield a new candidate solution for the current vertex. This candidate solution is sent for evaluation against the set of Stage 2 scenarios and the vertex is moved to a ``waiting'' queue. The compute object repeats the process as long as there are vertices waiting to be solved in the ready queue. Vertices move back from the waiting queue into the ready queue when the cuts from evaluating all the scenarios for the generated candidate allocation are sent back to the Tree Explorer. When a vertex ``converges'', that is, when the optimal fractional solution to the stochastic LP described by the vertex is found, it is ``retired'' by either pruning it or branching further.

The number of Tree Explorer objects is smaller than the number of vertices in the search tree. We also find from experiments that it is sufficient for the number of such Tree Explorers to be a small fraction of the number of processors in a parallel execution.

Cuts generated from a scenario evaluation can be used in all the Stage 1 LPs. However, we have found that this results in a deluge of cuts added to the Stage 1 library instances. In earlier work [17], we have observed a strong correlation between the number of cuts added to a library instance and the time taken for the LP solve. Hence, instead of sharing the cuts across the entire BnB tree, we share cuts only across vertices hosted by a single Tree Explorer. Cuts generated from the evaluation of a candidate solution are hence messaged directly to the solver hosting the corresponding vertex. However, the collection of cuts accumulated in a library instance continues to grow as more vertices are explored. Since some of these may be loose constraints, we discard them to make space for newer constraints. If these constraints are required again later on, they will be regenerated by the algorithm. We implement bookkeeping mechanisms that track the activity of cuts and retires cuts identified as having low impact (longest-unused, most-unused, combination of the two, etc). This maintains a fixed window of recent cuts that are slowly specialized to the collection of active vertices sharing that library instance. The impact of cut retirement on solve times is illustrated in [17].

B. Stage 2 Manager

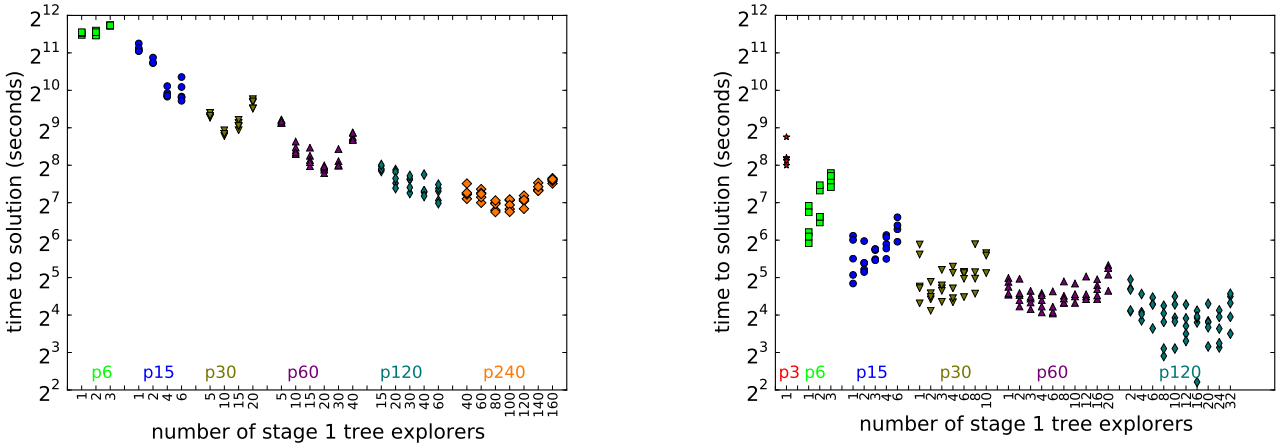
Candidate solutions from the Tree Explorers are sent to a Stage 2 Manager object. This object helps implement a pull-based work assignment scheme across all Scenario Evaluators. To do this, it maintains a queue of such candidate solutions and orchestrates the evaluation of all scenarios for each candidate. In order to remain responsive and ensure the quick completion of pull requests, the object is placed on its own dedicated core and other compute objects (which invoke, long, non-preempted LP solves) are excluded from that core. The Stage 2 Manager ensures that each Tree Explorer gets an equal share of Stage 2 evaluation resources by picking candidates from Tree Explorers in round-robin fashion.

C. Stage 2 Scenario Evaluators

In this design variant, all Tree Explorers share the same collection of Scenario Evaluators. A Scenario Evaluator request the Stage 2 Manager for candidate Stage 1 solutions and evaluate these solutions for one or more scenarios. Upon evaluation, they send the generated cuts directly back to the Tree Explorer that hosts the specific BnB vertex. This pull-based scheme ensures good utilization of the processors hosting Scenario Evaluators, and also balances the scenario evaluation workload across all the Stage 2 processors. Given that the Stage 2 LP solve times are typically much larger than the messaging overhead to obtain work, the pull-based approach has negligible overhead.

D. Load Balancing

A Tree Explorer maintains a private list of vertices. It regularly updates the Stage 1 Manager of the total number of vertices that it currently has. Whenever a Tree Explorer runs



(a) Design A

(b) Design B

Fig. 6. Strong Scaling for the 3t-120 model. Colors correspond to the scale (number of processors) e.g. p6 is for 6 processors, p15 for 15, and so on. At each scale, runs for performed for varying number of Tree Explorers. For each configuration 5 trials are performed to measure the variability

TABLE II
Average Stage 1 LP solve time comparison between Design A and Design B

Model	Stage 1 LP solve time (s)	
	Design A	Design B
3t-120	0.38	0.04
3t-240	0.98	0.08
5t-120	2	0.25

out of work i.e. has evaluated all its vertices, it requests the Stage 1 Manager for work. Stage 1 Manager selects the most loaded Tree Explorer and sends it a request to offload half of its workload to the starving Tree Explorer. The max loaded Tree Explorer sends half of its vertices and its LP solver state (cuts) to the starving Tree Explorer.

X. Performance and Analysis

All experiments were performed on the 268 node (3216 cores) Taub cluster installed at University of Illinois. Each node has Intel HP X5650 2.66 GHz 6C processors and 24GB of memory. The cluster has a QDR Infiniband network communications with a Gigabit Ethernet control network. We used Gurobi [8] as the LP solver.

As noted in 5a and 5b, Stage 1 Manager and Stage 2 Manager (in Design B) are placed on processor 0. Tree Explorer and Scenario Evaluator are place on disjunct set of processors, with Tree Explorer objects placed on processors 1 through M, and Scenario Evaluator objects placed on processors $M+1$ through N, where $M+N+1$ is the total number of processors. We use depth first search as the BnB vertex prioritization policy, where depth is determined by the total number of branching decisions taken on the path from the root node to that vertex. For vertices with the same depth, one with a smaller lower bound is given higher priority.

A. Variability in Execution time

As discussed in subsection VI-H, both designs suffer from variability in execution times across runs with identical configurations. Design A ensures that the branching order remains the same across all runs of the same model. However, as discussed in VI, the chronology of incumbent discoveries might vary slightly across runs, thereby causing different pruning decisions and different BnB tree sizes. Design B, in addition, has another source of variation. The order in which the Stage 1 and Stage 2 solves are done can alter the LP solutions to the same problem because of the combined effect of advanced start and degenerate Stage 1, Stage 2 LPs. This changes the branching decisions and hence different trees are generated. This can cause significant variation in the time to solution.

Figure 6 plots the performance of the two designs for 3t-120. On x-axis is the number of Tree Explorers. Each color corresponds to a scale e.g. p3 is for 3 processors, p6 for 6, and so on. At each scale, we measured the performance for varying number of Tree Explorers. For every configuration, we did 5 trials to measure the variability. The time to solution in these trials is plotted with markers in the same vertical line. Design A has much less variability as the markers are very close to each other as compared to the Design B, where performance varies even by an order of magnitude in some cases. In Figure 7, we plot the BnB trees explored in two identically configured executions of Design B on the 5t-120 model. This explains the large variation in performance of Design B.

B. Performance Comparison

The number of Tree Explorers at any given execution scale has a significant effect on the performance. Expectedly, increasing the number of Tree Explorers too much inundates Stage 2 with work and deteriorates performance. We have also ascertained that the concurrent execution of several Stage 1 LPs on the same compute node of the machine increases the individual solve times because of memory bandwidth limitations. From Figure 6 it is clear that Design B, despite having high variability has significant advantage in terms of

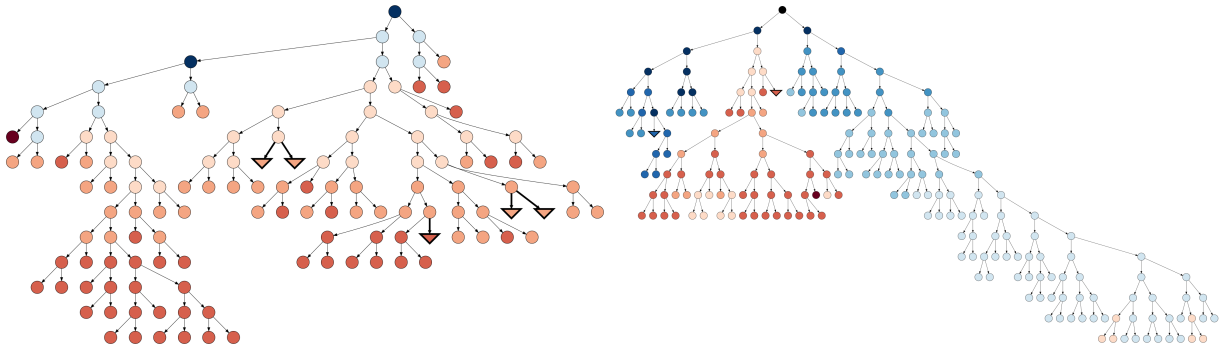
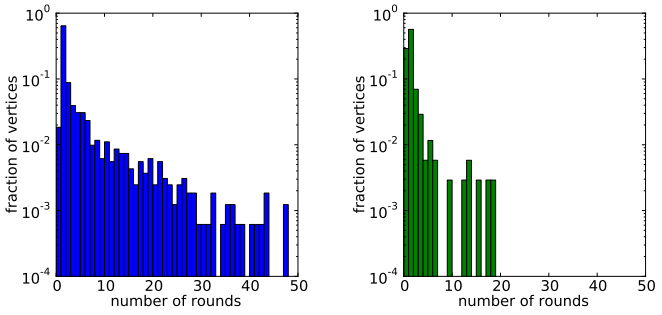


Fig. 7. The Branch-and-Bound trees from two identically configured executions of Design B for the 3t-120 dataset. The trees from the two trials are significantly different because of branching on different variables in different orders. This explains the large variation in performance across trials. Triangles represent integer solutions (incumbents), while the vertices are colored by the value of bound



(a) Design A (b) Design B

Fig. 8. Analyzing the cause of slower performance of Design A as compared to Design B. (a) and (b) plot the histogram of the number of rounds taken to solve the stochastic LP at the BnB tree vertices in the 5t-120 model.

solution speed over Design A. This advantage is two-fold. First, the number of rounds to achieve convergence at the tree vertices is much smaller in Design B. This effect is shown in 8a, 8b in which we plot a histogram of the number of rounds vertices take to converge in the two designs. This difference can be attributed to the difference in the set of Benders cuts that are maintained by the two designs. In an effort to maintain repeatability, Design A always starts with the cuts from the parent vertex. On the other hand, Design B uses the most current set of cuts resident on the processor being used. This means that Design B has access to cuts generated in different parts of the tree and is therefore likely to have more cuts that are binding and thus speed up convergence. Secondly, the stage 1 linear programs also take less time to solve in Design B (Table II). Since in Design A, every new vertex starts with a fresh start of the Gurobi library instance, a significant number of simplex iterations are required to optimize the LP in the first round for each vertex. Conversely, Design B always uses advanced start and the most recent cut set. The LPs differs from the previous vertex LP only in the few branching constraints and thereby, the LP solves very quickly using advanced start.

Even though Design A has better repeatability, the worst performance using Design B is better than the best performance using Design A. Therefore, Design B is the design of choice because of quicker time to solutions. Additionally,

TABLE III
Cumulative distribution of trials of Design B for a target minimum parallel efficiencies on 3t-120 (baseline: 3 cores)

Efficiency(%) >	**Number of processors**				
	6	**15**	**30**	**60**	**120**
100	1.0	0.9	0.95	0.066	0.0
90	1.0	1.0	0.95	0.066	0.2
80	1.0	1.0	0.95	0.466	0.4
70	1.0	1.0	1.0	0.733	0.4
60	1.0	1.0	1.0	0.866	0.6
40	1.0	1.0	1.0	1.0	1.0

TABLE IV
Cumulative distribution of trials of Design B for a target minimum parallel efficiency on 5t-120 (baseline: 3 cores)

Efficiency(%) >	**Number of processors**				
	6	**15**	**30**	**60**	**120**
100	0.95	0.7	0.8	0.2	0.0
90	0.95	0.75	0.85	0.4	0.0
80	0.95	0.8	0.85	0.4	0.0
70	1.0	0.8	0.9	1.0	0.2
60	1.0	0.85	0.9	1.0	0.6
40	1.0	0.85	1.0	1.0	0.8

Design A suffers from large memory requirements for cut dump collection, which can become a bottleneck for larger data sets in which the tree frontier becomes very large before the solution is found.

C. Performance of Design B

Using large-scale parallel computing for an application is advantageous when it is guaranteed that running the application on more processors will give faster times to solution. Unlike typical scientific iterative applications, Design B for this application suffers from large variability in execution times for runs with identical configurations, which makes it difficult to measure its parallel efficiency. We therefore need a different method to quantify its parallel efficiency in the wake of variation. Our method is to measure the probability of getting a certain parallel efficiency. To measure the performance of Design B with this metric, we did 20 trials of Design B with each of 3t-120 and 5t-120 datasets. In Table III and Table IV, first column has the parallel efficiencies. Rest of the columns report, at different scales, the fraction of trials that achieved greater efficiency than the corresponding entry in the first column. For example, for 3t-

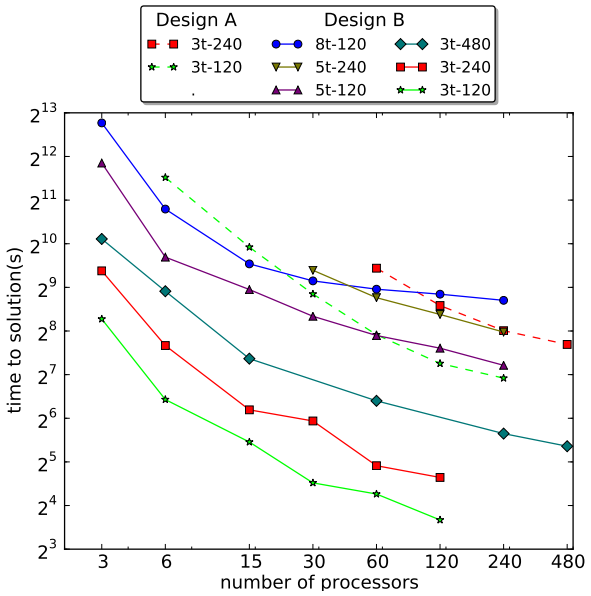


Fig. 9. Scaling of various models for Design A and Design B

120, the parallel efficiency was greater than 90% in 95% of the trials at 6 processors and in 75% of the trials at 15 processors. These results show that in majority of the cases efficiency was greater than 40% at all scales for both the datasets. Also note the super linear speedup in some cases. As compared to Gurobi's typical efficiency of 10% for IPs [6], our algorithms yield significantly higher parallel efficiencies even at larger scales.

We further report the scaling of Design A and Design B in Figure 9. We identify the best performing Tree Explorer count at each scale by comparing the average time to solution across 5 trials. Average times to solutions for these Tree Explorer counts are presented in Figure 9 for several datasets. We get very good incremental speedups on up to 480 processors for several datasets. The scaling at large scales is limited by the root vertex optimization, which takes many rounds to converge as compared to the other vertices. During root node optimization there is only 1 vertex and hence no Stage 1 parallelism. Scaling at large scales is additionally limited by the critical path to reach the optimal solution.

XI. Summary

We have discussed and presented several factors that influence the design and performance of parallel, two-stage stochastic integer programs solved using Branch-and-Bound. We have also presented two designs that prioritize different factors: 1) a nested parallel decomposition that solves each BnB vertex in isolation and 2) a design variant that shares LP library solvers as well as Stage 2 feedback across BnB vertices. The interplay between some of the factors like memory usage, solver sharing, degeneracy and tree structure are borne out by the performance results for both these designs on multiple datasets. Sharing solvers and cuts results in more variable, yet better performance. We also show strong scaling from 6 cores up to 480 cores of a dual hex-core, 2.67 GHz, Intel Xeon cluster. Because of the inherent variability in the amount of computation required, we also report the spread in performance by tabulating the fraction of trials that achieved various parallel efficiencies. We believe these are noteworthy results for strong scaling such an unconventional problem.

However, there is still a need for further characterizing the behavior of parallel stochastic integer programs; and for further research into techniques for improved scalability. We feel our experiences and findings are a useful addition to the literature and can seed further work in this direction.

Acknowledgments

This research was supported by MITRE Research Agreement Number 81990 with UIUC. We acknowledge the use of the Taub compute resource provided by the Computational Science and Engineering Program at the University of Illinois. We thank Wayne L. Hoyenga (UIUC) for his help with licenses, configuration and access to the compute resources. We used Gurobi [8] solvers under a license that permits academic use at no cost.

References

[1] J. Benders. Partitioning Procedures for Solving Mixed Variables Programming Problems. *Numerische Mathematik 4*, pages 238--252, 1962.

[2] J.R. Birge and F.V. Louveaux. A Multicut Algorithm for Two-stage Stochastic Linear Programs. *European Journal of Operational Research*, 34(3):384--392, 1988.

[3] F.V. Louveaux and R. Schultz. Stochastic Integer Programming. *Handbooks in operations research and mgmt science*, 10:213--266, 2003.

[4] N.V. Sahinidis. Optimization Under Uncertainty: State-of-the-art and Opportunities. *Computers & Chemical Engg*, 28(6):971--983, 2004.

[5] Y. Xu, T.K. Ralphs, L. Ladányi, and M.J. Saltzman. Computational Experience with a Software Framework for Parallel Integer Programming. *INFORMS Journal on Computing*, 21(3):383--397, 2009.

[6] T. Koch, T. Ralphs, and Y. Shinano. Could we use a Million Cores to Solve an Integer Program? *Mathematical Methods of Operations Research*, pages 1--27, 2012.

[7] Amitabh Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In *Workshop on Dynamic Object Placement and Load Balancing, in co-operation with ECOOP's 92*, Utrecht, The Netherlands, April 1992.

[8] Gurobi Optimization Inc. Software, 2012. http://www.gurobi.com/.

[9] G.R. Bitran, E.A. Haas, and H. Matsuo. Production Planning of Style Goods with High Setup Costs and Forecast Revisions. *Operations Research*, 34(2):226--236, 1986.

[10] M.A.H. Dempster, ML Fisher, L. Jansen, BJ Lageweg, JK Lenstra, and A.H.G.R. Kan. Analysis of Heuristics for Stochastic Programming: Results for Hierarchical Scheduling Problems. *Mathematics of Operations Research*, 8(4):525--537, 1983.

[11] Maarten H. van der Vlerk. Stochastic Integer Programming Bibliography. http://www.eco.rug.nl/mally/biblio/sip.html, 1996-2007.

[12] L.F. Escudero, M. Araceli Garín, G. Pérez, and A. Unzueta. Scenario Cluster Decomposition of the Lagrangian Dual in Two-stage Stochastic Mixed 0-1 Optimization. *Computers & Operations Research*, 2012.

[13] IBM CPLEX Optimization Studio. Software, 2012. http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/.

[14] J.P. Watson, D.L. Woodruff, and W.E. Hart. PySP: Modeling and Solving Stochastic Programs in Python. *Mathematical Programming Computation*, pages 1--41, 2011.

[15] PySp: Python-based Stochastic Programming Modeling and Solving Library, 2012. https://software.sandia.gov/trac/coopr/wiki/PySP.

[16] Miles Lubin, Kipp Martin, Cosmin Petra, and Burhaneddin Sandıkçı. On Parallelizing Dual Decomposition in Stochastic Integer Programming. *Operations Research Letters*, 2013.

[17] Akhil Langer, Ramprasad Venkataraman, Udatta Palekar, Laxmikant V. Kale, and Steven Baker. Performance Optimization of a Parallel, Two Stage Stochastic Linear Program: The Military Aircraft Allocation Problem. In *Proceedings of the 18th International Conference on Parallel and Distributed Systems (ICPADS 2012). To Appear*, Singapore, December 2012.

[18] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91--108. ACM Press, September 1993.

[19] Laxmikant Kale, Anshu Arya, Nikhil Jain, Akhil Langer, Jonathan Lifflander, Harshitha Menon, Xiang Ni, Yanhua Sun, Ehsan Totoni, Ramprasad Venkataraman, and Lukasz Wesolowski. Migratable Objects + Active Messages + Adaptive Runtime = Productivity + Performance A Submission to 2012 HPC Class II Challenge. Technical Report 12-47, Parallel Programming Laboratory, November 2012.