In Search of a Scalable, 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 such problems.**

In this paper, we explore the parallelization of a two-stage stochastic integer program solved using branch-and-bound. We present three design variations and describe the factors that shaped our quest for scalability. 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. These problems greatly incentivize short times to solution.

Unlike typical, iterative scientific applications, we encounter some very interesting characteristics that make it challenging to realize a scalable design. The total amount of computation required to find optima is not constant across multiple runs. This challenges traditional thinking about scalability and parallel efficiency. It also implies that reducing idle time does not imply quicker runs. The sequential grains of computation are quite coarse. They display a wide variation and unpredictability in sizes. The structure of the branch-and-bound search tree is sensitive to several factors, any of which can significantly alter the search tree causing longer times to solution. We explore the causes for this fragility and evaluate the trade-offs between scalability and repeatability.

Our attempts result in strong scaling to hundreds of cores for the small, sample datasets that we use. We believe our experiences will feed usefully into further research on this topic.

Optimization problems are of real-world relevance, often with multi-million dollar consequences. However, real-world problems often have a level of uncertainty in inputs, environmental factors, objectives etc. that should influence the optimal solution. Stochastic optimization captures this uncertainty by formulating problems as two-stage (or multi-stage) models that propose candidate solutions and evaluate them across a spectrum of potential scenarios.

In this paper we present our quest for the scalable, parallel solution of stochastic optimization problems. Specifically, we are interested in problems that require integer solutions, and hence, interested in Branch-and-Bound (BnB) approaches. BnB, a common, well-studied approach to finding optima for integer programs is notoriously hard to parallelize. Additionally, there has been little prior work in parallelizing or scaling two-stage, stochastic integer programs.

Our work is set in the context of a US airfleet management problem where aircraft are allocated to missions under uncertain cargo movement demands (II). However, the parallel scaffolding and the techniques we have developed are relevant elsewhere too.

We structure this paper in the form of a narrative that describes our findings and how they motivate successive modifications to our parallel designs. Once past the introductory sections (I–III), we discuss factors that influence the design of a parallel, stochastic integer program (IV). This sections makes for interesting reading and presents some of the factors that set this problem apart from typical computational science applications. We pick a programming model that enables the expression and management of the available parallelism (V). Finally, in sections VI–VIII, we present the motivations, structure and performance of each design.

I. Background

In two-stage stochastic optimization using Benders decomposition, candidate solutions are generated in Stage One (Stg1) (Eq. 1) and they are evaluated in Stage Two (Stg2) for every scenario. Stg2 (Eq. 1) provides feedback to Stg1 in the form of *cuts*, which are used by the Stg1 to improve the candidate solution. The process iterates unless no better candidate solutions can be found.

$$\min Cx + \sum_{k=1}^{K} p_k \theta_k \quad s.t. Ax \leq b,$$

In the objective function(1), x corresponds to the candidate solution, C is the cost coefficient vector, $\theta = \{\theta_k | k = 1, ..., k\}$ is the vector of Stg2 costs for the k scenarios and p_k are the probability of occurrence of scenario k,

$$\theta_k = \min {q_k}^T y \quad s.t. Wy \leq h_k - T_k x$$

Stochastic programming problems specify problem data as a probability distribution. While most research on stochastic programs assumes that the decision variables are continuous, our problem is made harder by the fact that some of the decision variables are integer-valued.

Louveaux and Schultz [1], Sahinidis [2], in the broader context of decision-making under uncertainty, give excellent overviews of stochastic integer programming problems. Much of the research on two-stage stochastic integer programs is concentrated on the calculation of the stage two value function (e.g. [1]). This is particularly challenging when the Stg2 problem has integer decision variables. In our case, the integer variables are confined to Stg1 of the problem which makes the problem theoretically easier, but still requires the use of enumerative search techniques such as the BnB method. The common method to solve such problems is an extension of the L-shaped method used to solve stochastic linear programs. This method by Laporte and Louveaux [3] is called the integer L-shaped method. We use a multi-cut variant of this method.

II. Case Study: Military Aircraft Allocation

The motivation and context for our study of stochastic integer programs comes from an airfleet management task performed by the US Air Mobility Command. The division 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 objective is to plan for an upcoming time period by accounting for the uncertainty in upcoming demands and to allocate aircraft to missions such that the overall costs of short-term aircraft leases or undelivered cargo is 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 small, but representative datasets that model this problem. The datasets are classified based on the number of time periods (typically, days) in the planning window and the number of possible scenarios that need to be evaluated to account for the uncertainty. In this work, we primarily use the 3t and 5t datasets with 120 possible scenarios in Stg2. For context, the 5t dataset has approximately 250 integer variables in the Stg1 Integer Program (IP), $1.6M$ variables in the Stg2 Linear Program (LP), and about $1M$ Stg2 constraints when evaluating 120 Stg2 scenarios.

III. Prior Work

Examples of the uses of stochastic integer programming can be found in literature. Bitran et al [4] model production planning of style goods as a stochastic mixed integer program. Dempster et al [5] consider heuristic solutions for a stochastic hierarchical scheduling problems. A comprehensive listing of work on stochastic IPs can be found here [6].

Large scale solvers for mixed integer programs have been studied by a number of authors [7], [8]. The difficulty in achieving high efficiencies has been documented. Kale et al [9] have studied the challenges of dynamic load balancing in parallel tree search implementations. Gurobi [10] has a state-of-the art mixed integer program solver that exploits multi-core architectures. However, Koch et al in [8] 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.

To the best of our knowledge, large-scale optimization of stochastic integer optimization has not been systematically studied. PySP [11], [12] is a generic decomposition-based solver for large-scale multistage stochastic mixed-integer programs. 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 from poor parallel efficiency because of MIP solve times. Escudero et al [13] note that MIP Solvers such as CPLEX [14] do not provide solution for even toy instances of two stochastic integer programs in a viable amount of time.

IV. Design Considerations

A. Coarse-Grained Decomposition

In our designs, we choose to delegate sequential LP solutions 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 grains and has profound consequences that have shaped our experiences and this narrative.

B. Solver Libraries Maintain Internal State

Unlike other numerical libraries, LP solvers maintain internal state across calls. This state represents a characterization of the feasible space of solutions for the previous problem that they solved, and also information about the last solution that was found. Most usage scenarios 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 or by reusing the previous characterization of the search space. Hence, it is highly advisable 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 of interest to us follow this pattern too. There are many iterations (rounds) to converge to a solution. In one stage, each iteration only modifies a few constraints on the feasible search space. In the other, the coefficient matrix maintained by the solver remains the same, and only the right-hand sides of all the constraint equations are modified across calls. A more detailed discussion on the impact of advanced starts can be found in [15].

Hence, its quite desirable 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

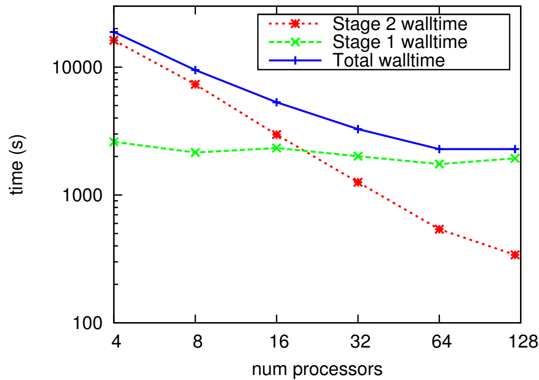

Fig. 1. A simple parallelization scheme that only exploits the readily available concurrency of evaluating multiple scenarios exhibits serial bottlenecks, and scaling limited by Amdahl's law. Results are for a 10t dataset with 1000 scenarios; and were obtained on the cluster Abe (dual quad-core 2.33GHz Intel Clovertowns nodes with GigE).

desirable to pick a parallel programming paradigm that will permit encapsulating and managing multiple solver instances per processor.

C. Exposing and Interleaving Nested Parallelism

Two-stage stochastic IPs have a natural expression as 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 [15], we focused on a simple iterative, master-worker design that tapped the readily available parallelism in Stg2 by evaluating multiple possible scenarios simultaneously. This captured much of the low-hanging, easily exploitable parallelism. However, the master-worker design was quickly limited by the serial bottleneck of performing Stg1 computations, as demonstrated in Figure 1.

The objective in our earlier attempt was the solution of a stochastic LP. However, the current objective is to solve stochastic IPs; which involves proposing candidate integer solutions in the first stage by solving IPs. This will magnify the serial bottleneck of the master-worker design so much that it becomes a completely untenable design. Thus, it is imperative to expose more parallelism than available in a simple master-worker decomposition.

Instead of delegating the whole IP to a sequential library, we decompose and parallelize the BnB search required to find integer solutions. BnB proceeds by branching on fractional parts of a solution and restricting each branch to disjoint portions of the search space, until gradually all components in the solution are integerized. 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. 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 in parallel. This nested parallelism has to be exploited for any reasonable

Dataset Name	Stg1 Memory Usage (MB)	Stg2 Memory Usage Per Scenario (MB)
3t	50	10
5t	110	15
10t	230	30
15t	950	45

TABLE I
MEMORY USED BY ONE INSTANCE OF THE GUROBI LIBRARY FOR STG1 AND STG2 OF SAMPLE DATASETS.

scalability.

A relevant observation that influences processor utilization is the mutual exclusivity of the two stages. For a given vertex, Stg1 cannot proceed while it is waiting for feedback from Stg2, and Stg2 is necessarily dependent on Stg1 for each new candidate solution. Ensuring high utilization of compute resources will therefore require interleaving the iterative two-stage evaluation of multiple BnB vertices.

D. Naive Stage One Parallelization is Limited by Memory

Each vertex in the BnB tree iterates through multiple rounds of Stg1 and Stg2 till it converges to a solution for the constraints at that vertex. When it converges to a solution, there are three possible outcomes for the fate of the vertex: (*a*) the solution components are all integers, in which case the vertex is said to be an incumbent. No further processing is required for the vertex, and the cost of the solution represents a floor that has to be bettered by other vertices to be considered candidates for optima; (*b*) the solution components still have fractions, but the cost of the solution is already worse than a known incumbent. In this case the vertex is "pruned", meaning its descendents are not explored any further; (*c*) the solution components still have fractions, and the cost of the solution still warrants exploring the subtree beneath the vertex for possible integer solutions. In this case additional branching constraints are imposed on the vertex to further integerize the solution, and the resulting child vertices are explored further.

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 single LP solver instance for the sample datasets of the aircraft allocation case study are shown in Table I. This implies that the number of solver instances can be substantially smaller than the number of vertices in a large BnB search tree. Hence, this first approach to parallelizing the Stg1 BnB is limited by the available memory.

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,

Fig. 2. LP solves, which are the smallest unit of computation, display sizeable variation in time to solution. This variation is not predictable too. Hence, the blocks of sequential computation do not display any persistent performance behavior that can be exploited. This plot shows a sample execution profile (on several processors) of evaluating multiple Stg2 scenarios for candidate Stg1 solutions. Colored bars represent an LP solve, while white stretches are idle times on that processor. Each processor (horizontal line) is assigned a specific Stg2 scenario, and evaluates multiple candidate solutions from Stg1 one after the other. Some loose synchronization is introduced so that every vertical grouping of bars represent the times taken for the evaluation of different scenarios for the same candidate solution from Stg1. We observe a wide variation and no persistence, both across scenarios and across candidate solutions. This illustrates the shortcomings of a static decomposition of the available work across processors.

carefully manage the number of active vertices, and expose as much parallelism as permitted by memory constraints.

E. Unpredictable Grain Sizes

Delegating the LPs to a library causes a very coarse-grained execution profile. We observe sizeable variation in the time taken for an LP. This is true for both Stg1 and Stg2 LPs. Additionally, we do not observe any persistence in the time taken for LP solves. A single Stg1 LP for a given vertex may take widely varying times as a result of the addition of a few cuts from Stg2. Likewise, we do not observe any persistence in Stg2 LP solve times either across different scenarios for a given Stg1 candidate solution, or for the same scenario across different candidate solutions. An illustrative execution profile is presented in Figure 2.

This complete unpredictability implies that a static a priori partition of work across different compute objects (or processors) will not ensure high utilization of the compute resources. The utter lack of persistence in the sizes of the sequential grains of computation also precludes the use of any persistence-based dynamic load balancing solutions. Hence, our designs explore 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.

F. 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 two 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 usually a middle phase when there are a large, but fluctuating number of vertices on the exploration front depending on branching and pruning rates. Towards the end, as solutions close to optima are found, pruning starts to dominate and the front of exploration shrinks rapidly. This tends to reduce the tree to a few strands, that finally shrinks to one path leading to the optimum. Any parallel design has to necessarily cope with, and harness these varying levels of available concurrency.

G. Better Utilization $\neq$ Better Performance

Due to the varying degrees of parallelism and the unpredictable grain sizes, the primary metric of performance of such an application is the time to solution. For many high performance computing 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.

V. Programming Model

The designs that we discuss here are implemented in an object-based, sender-driven parallel programming model called Charm++ [16], [17] . Charm++ is a runtime-assisted parallel programming framework based on 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 method invocations 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) 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 parallelis without bridling it with synchronization constructs.

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

VI. Design 1-A: Vertices Share Constraints on Feasible Space

A. Design Motivations

1) Vertices in Stg1 Can Share State: While the set c branching constraints for each vertex are unique to it, th characterization of the feasible space in the form of cuts fro Stg2 is not. The addition of a branching constraint influenc the candidate allocations that are generated in Stg1. These, i turn, only affect the right hand sides in the Stg2 LPs, which simply alters the objective function in Stg2. The dual polytope (which is the characterization of the feasible space) of the Stg2 LPs for all scenarios remains unchanged. Indeed, regardless of the vertex in the BnB tree, the dual polytope for any Stg2 problem remains constant. This implies that the dual optimal solutions obtained in Stg2 for a candidate solution from the Stg1 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 Stg2 LPs remain valid irrespective of the branching constraints imposed on a vertex. This implies two facts that can both be exploited in the program design: 1) each vertex of the BnB tree can inherit the Benders cuts from its parent vertex. 2) cuts generated from evaluating scenarios for a given vertex are also valid for all vertices in the BnB tree.

This forms the core principles underlying our first design. Since every cut generated from the evaluation of any scenario in Stg2 for any vertex is valid across the whole BnB tree, we share the cuts across vertices wherever possible. The concurrent evaluation of multiple BnB vertices generates a large set of cuts, which should greatly accelerate the characterization of the feasible space; and hence shortens the number of iterations required for any vertex to converge.

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 Stg1 LP solver instance can be reused to solve a Stg1 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 constraints specific to the vertex that was just in a Stg1 LP solve, and reapply constraints specific to another vertex that is waiting for such a Stg1 solve. This permits exploring more vertices than the available number of solver instances, and also retains the ability to exploit warm starts for each Stg1 LP solve.

The reasoning presented here also implies that the same Stg2 solver instance can be used to evaluate multiple scenarios across multiple vertices. Hence, our first design aims to maintain a few library solver instances for Stg1 and Stg2

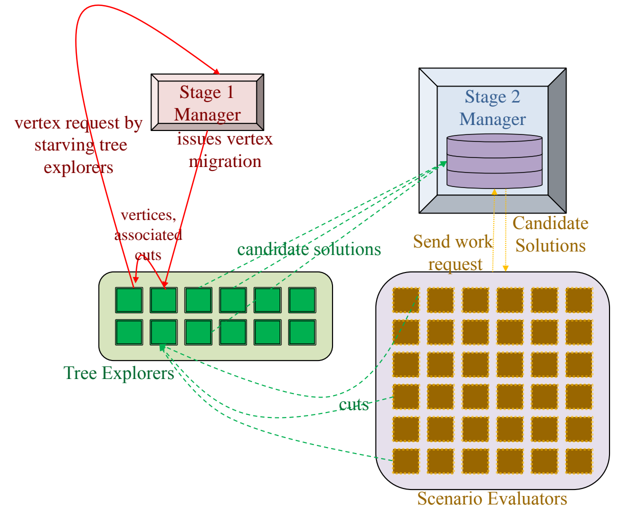

Fig. 3. Schematic for design 1-A. Enables cut sharing across vertices of the BnB tree

respectively, and interleave the LPs for multiple vertices on these solvers.

B. Parallel Structure

The considerations described earlier lead to a two-stage design with two primary collections of interacting compute objects that host solver instances. There are also a few other objects required for orchestrating the execution (Figure 3). The overall parallel structure is described here.

1) Stg1 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. However, each 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 Stg1 LP for that picked vertex. The Stg1 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 Stg2 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 far lesser than the number of vertices in the search tree. We also find from experiments that it is sufficient for the number of such Tree

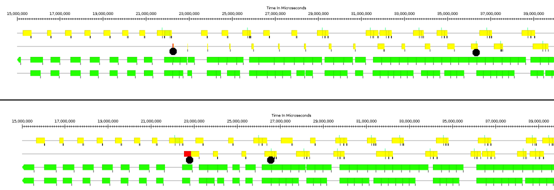

Fig. 4. Execution traces that demonstrate vertex migration in the ramp-up phase. Work, as it becomes available is distributed to more Tree Explorers. The first two processors in the traces (with execution events colored yellow) host Tree Explorers. The traces also illustrate the benefits of maintaining cut locality. The traces on the top half of the figure demonstrate the time required for a migrated vertex to converge while building up a collection of cuts at the new Tree Explorer. The traces in the bottom half demonstrate the shortened walltime to convergence when the cuts are migrated too. The migration and convergence events are marked with black dots for clarity. The short red stretch in the execution profile with cut migration represents the migrated cuts being applied to the local LP solver instance.

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 Stg1 LPs. However, we have found that this results in a deluge of cuts added to the Stg1 library instances. In earlier work [15], 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 cuts might no longer actively constrain the polytope, but may only be loose constraints, these can be safely discarded. 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 [15].

2) Stg2 Manager and the Vertex Queue: Candidate solutions from the Tree Explorers are sent to a Stg2 Manager object. The raison d'etre for this object is the unpredictability of the Stg2 LP solve times. This object helps implement a pull-based work assignment scheme across all Scenario Evaluators. To do this, it maintains a fair 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.

Since even the Stg1 LP solve times display some variation, each Tree Explorer can produce candidate solutions at differing rates. The Stg2 Manager ensures that each Tree Explorer gets an equal share of Stg2 evaluation resources by picking candidates from Tree Explorers in round-robin fashion.

3) Stg2 Scenario Evaluators: Akin to the Tree Explorers, the Scenario Evaluators are a collection of compute objects each of which hosts an LP instance. They request the Stg2 Manager for candidate Stg1 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. Since the solve times are typically much larger than the time for the roundtrip messaging required to obtain a candidate solution from the Manager object, this ensures good utilization of the processors hosting Scenario Evaluators, and also balances the scenario evaluation workload across all the Stg2 compute objects.

The total number of Stg2 LPs is the product of the number of candidate allocations and the number of scenarios. There are some interesting choices in how these LPs are assigned to different Scenario Evaluators. Similar scenarios could be clustered together into packets of work based on the assumption that their optima would lie in the same neighborhood of the dual polytope. A single compute object could then evaluate all the scenarios in the cluster, exploiting warm starts to the fullest. LPs could also be grouped based on the candidate solution they are evaluating. A discussion of the effects of such clustering and warm starts can be found in [15].

C. Work Distribution and Load Balancing

As described earlier, we observe three primary phases in BnB search: the ramp-up, middle and ramp-down phases. During the ramp-up and ramp-down phases there is insufficient parallelism available, and we need techniques to exploit any additional concurrency as soon as it becomes available. We achieve this by migrating vertices across Tree Explorers. In the middle phase, typically there is abundant work to be done. However, due to the unpredictability of solve times its difficult to pick a good balance between Stg1 and Stg2 compute resources. We combat this by automatically adjusting this balance based on collected performance statistics. Both these techniques are described in this section. The required capabilities are implemented as part of a Stg1 Manager object. Again, for the sake of responsiveness, compute objects are excluded from the core that hosts the Stg1 Manager.

1) Vertex Migration: BnB starts at a root vertex and ends at a vertex that represents the optimum. In the neighborhood of both these end points, there is either insufficient branching, or aggressive pruning. Both result in fewer vertices than available Tree Explorers, resulting in potentially idle Tree Explorers. Idle Tree Explorers can also occur in the middle phase if a Tree Explorer prunes all the vertices that it owned. To automatically exploit any parallelism as it becomes available, and keep all Tree Explorers busy with useful work, we implement a mechanism loosely similar to work stealing.

We employ a single work request mechanism in all situations when a Tree Explorer finds that it has no vertices to explore. The object then registers a request with the Stg1 Manager. The Stg1 Manager periodically collects load information from all the Tree Explorers, and hence can identify the most overloaded object. Here load can refer to the (total or unexplored) number of vertices that are owned. The Manager then instructs the migration of one or more vertices from the

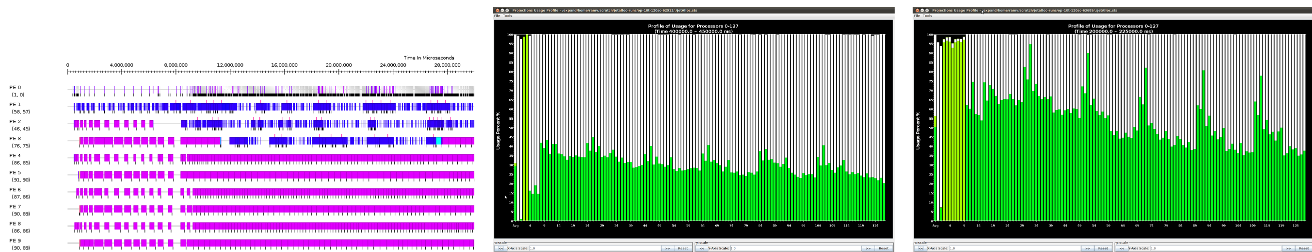

Fig. 5. Execution traces from a few relevant processors of a larger execution that captures the adaptive shrink / expand of the number of Tree Explorers to maximize rate of vertex exploration. Each processor is a horizontal line in the execution traces. Blue bars represent Tree Explorer computations, while pink bars represent Scenario Evaluator computations. Based on the performance and exploration throughput statistics, the number of processors hosting Tree Explorers is automatically adjusted. The impact of such reassignment can be observed in the "before-and-after" processor utilization profiles. Both plot utilization during a small portion (25-50s) of the overall execution and are obtained on 128 processors. The first profile is before the dynamic adjustment kicks in, and shows just two processors hosting Tree Explorers, while the second shows increased utilization because a greater number of processors (8) are automatically dedicated to hosting Tree Explorers. Such introspective adjustment occurs periodically throughout the execution.

overloaded object to the idle object. These vertices are used to seed a new sub-tree of exploration at the idle object. This mechanism successfully doles out any available units of work to idle Tree Explorers and keeps them occupied as far as possible.

As described earlier, each Tree Explorer gradually specializes its collection of cuts for the currently active vertices. However, a migrated vertex moves to a Tree Explorer whose collection of cuts is tailored to exploring a different region of the BnB tree. Our experiments reveal that this results in significantly more rounds for the newly busy Tree Explorer object to converge to a solution for the migrated vertex. The walltime to convergence on the migrated vertex is even greater than it would have been without the migration, making the scheme counter-productive. This is especially true when migrations occur during the ramp-down phase when there is not much benefit to reconstructing a specialized collection of cuts.

In order to maintain the locality of the cuts to the region of the tree being explored, we also implement mechanisms to migrate the collection of cuts along with the migrated vertex. The idle Tree Explorer discards all its existing cuts and loads the newly received cuts into its library instance. We observe that this greatly accelerates convergence on newly acquired vertices, and restores the benefits of vertex migration. Vertex migration, and the benefit of cut migration are illustrated in Figure 4.

2) Automatically Adjusting Stg1-Stg2 Processor Allocation: The program starts with an initial user-specified number of active Stg1 and Stg2 objects. The number of Tree Explorers determines how many Stg1 LP solves can happen concurrently, while the the number of Scenario Evaluators determines how many Stg2 scenarios can be evaluated concurrently. Setting the number of Tree Explorers too low might not explore vertices fast enough to keep all the Scenario Evaluators busy. Setting it too high might generate and push candidate solutions into Stg2 Manager's queues faster than can be drained by all the Scenario Evaluators. This will cause a high turnaround time for the cuts from each candidate solution to return to its Tree Explorer object; resulting in either Tree Explorers idling, or causing a breadth-first search because the Tree Explorers start exploring more and more vertices while they wait for a response from Stg2. Since the amount of parallelism varies over time, it is quite difficult to predict a good balance that will achieve the highest rate of exploration of BnB tree vertices.

We automate this process by using measured performance statistics to decide whether the number of Tree Explorers need to be adjusted. The Stg1 Manager object periodically collects data from all the Tree Explorers. Non-blocking reductions (available in Charm++) permit such reductions to proceed without imposing an implicit synchronization barrier on the inherently unsynchronized Tree Explorers. The collected data includes vertex queue lengths, average LP solve times, idle times on processors hosting compute objects etc. This data is used to compute the LP solve rates in Stg1 and Stg2, and eventually a decision on whether there are sufficient processors allocated to explore the BnB tree. Any decision on the number of processors to shrink / expand the Stg1 resources by, is followed by spawning the appropriate compute objects on these processors. The newly launched compute objects then use the existing work request mechanisms (Stg1 vertex migration or Stg2 pull requests) to get work. We observe that this automation significantly boosts processor utilization and increases the rate at which the tree is explored (Figure 5).

D. Performance and Analysis

Figure 6 shows the performance of this design for the 5t* dataset. All performance data in this plot is using one Tree Explorer. Its worth remembering that, although a single Tree Explorer object may not yield the fastest vertex exploration rates, it does not preclude a parallel exploration of the tree because each solver can interleave the exploration of several vertices. However, we deliberately constrain the number of Tree Explorers to illustrate noteworthy performance characteristics.

The large variation in performance at any given scale is evident from the plot. The performance data presented here are the result of several iterations of design improvements and tuning. This includes much of the design discussed thus far:

pull-based work distribution; adaptive processor redistribution; vertex and cut migrations to shorten the ramp-up and ramp-down phases; several studies of cut locality, the impact of cut sharing, cut retirement windows etc. Owing to space constraints, we do not present the performance data for each of these individual design iterations, but only the end results. However, despite observing some scalability, the results are not encouraging because of the large variations across multiple identically configured runs.

VII. Design 1-B: Ensuring a Globally Prioritized Search

A. Design Motivations

The large variation in performance that we observe with the previous design is of concern. Our next explorations attempt to understand and mitigate this. We discuss two phenomena to motivate the second design.

Our experiments prior to implementing adaptive Stg1-Stg2 resource distribution showed that sometimes the number of Tree Explorers overwhelmed the available Stg2 resources. This resulted in long turnaround times for any candidate solution that was sent to Stg2 for evaluation; causing each Tree Explorer to start exploring as many vertices as possible to remain busy during this long turnaround period. This further inundated Stg2 with more work, eventually leading to a breadth-first search on the tree. The time to solution in such cases is enormously worse and is tenable only for the smallest of problems.

Even with an adaptive adjustment of the number of Tree Explorers, we noticed that even small increases in this number caused wide performance fluctuations. To study the sensitivity to the number of Tree Explorers, we performed an illustrative experiment that shaped our hypothesis for this design. We gradually stepped to a higher number of Tree Explorers as we increased the scale of execution. We conducted multiple trials of this experiment and the results are plotted in Figure 7. Despite ensuring that there were always sufficient compute resources for the Stg2 portion of the computation, we noticed some very pathological scaling behavior. Adding more Tree Explorers was markedly counter-productive.

However, we did observe some scaling when using just one Tree Explorer. This led us to the suspicion that fragmenting the BnB search across multiple Tree Explorers was somehow degrading performance.

Each Tree Explorer explores BnB tree without feedback from, or interaction with, others. This is equivalent to fragmenting the BnB tree into as many pieces as Tree Explorers and then performing each subtree search separately. Each of these individual subtree searches then adaptively explores as many vertices as required to ensure high utilization. Each Tree Explorer locally follows the appropriate search strategy that was requested (depth-first, most-promising vertex-first, etc). However, there is no global coordination that enforces this search policy across all Tree Explorers. The second design stems from this realization and seeks to ensure that all Tree Explorers spend their time on the next most *globally* important

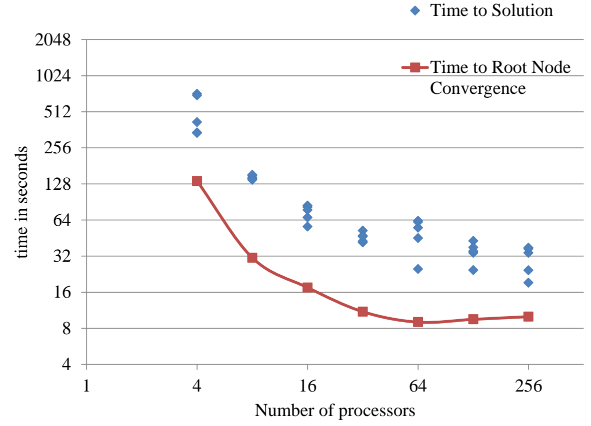

Fig. 6. Performance data for design 1-A for the 5t* dataset. This plots the time to solution at different scales with multiple trials at each scale. A single Tree Explorer is used at all scales. Note the large variation in performance at any given scale. The dataset used for this plot is different from that used in the rest of this paper and, unfortunately, does not permit a direct comparison with the other designs.

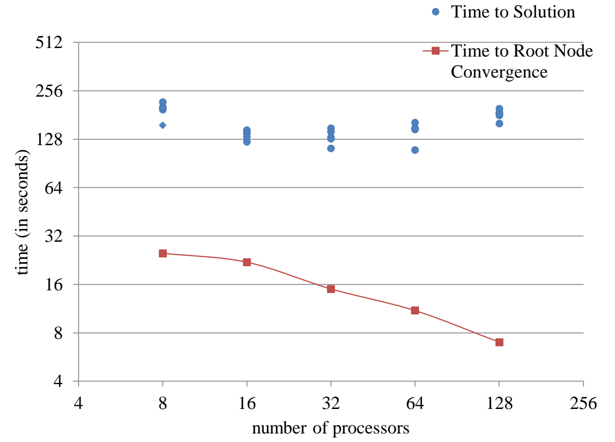

Fig. 7. Performance of design 1-A with gradually increasing number of Tree Explorers at each scale. We observe that the performance is sensitive to the number of Tree Explorers and that adding more Tree Explorers is markedly counter-productive. This is despite ensuring reasonable turnaround times from the Stg2 compute resources. The results, combined with those from Figure 6 seem to imply that fragmenting a global view of the BnB tree is counter-productive. Results are for the same 5t* dataset as used in Figure 6.

vertices that are available for exploration. Consistently exploring the most interesting portions of the search tree might make the best use of the available Tree Explorers.

B. Parallel Structure

There are several means to ensure a globally prioritized search, so that all the Tree Explorers collectively explore the most interesting portions of the BnB tree, rather than simply partitioning the tree amongst themselves. We pick a relatively simple solution that trades cut locality for global prioritization, and is sufficient for the scales at which we test our designs.

Our design simply introduces a global "ready" queue of BnB vertices that is prioritized according to the search policy. Any Tree Explorer that is idling can pull a vertex from this global queue and start exploring it. Upon branching, both child vertices are pushed back into the global queue

Num processors	Average	Min	Max
60	474.88	291.53	653.22
120	306.27	60.79	478.36
240	544.27	269.07	903.21

TABLE II
DESIGN 1-B STILL EXHIBITS PERFORMANCE VARIATION. DATA BASED ON FIVE TRIALS WITH 3T DATASET.

and the currently most important vertex is pulled for further exploration. This global ready queue is naturally embedded in the Stg1 Manager object. Vertices pulled from the global queue alternate between local "ready" and "pending" queues private to each Tree Explorer. Partially explored vertices in the local ready queue are given preference over those in the global ready queue.

The fair-share policy for sharing Stg2 compute resources across Tree Explorers is also replaced with a policy that prioritizes the Stg2 evaluations for the most important vertices. The design aspires to ensure that the most important vertices are always explored the fastest, in hope that this will quickly lead to an integer solution that will help prune the tree.

The description of cut sharing in section IV explicitly states that cuts are valid across the entire BnB tree. This design exploits that to the fullest by permitting vertices to be pulled by solvers which have an unrelated collection of cuts. The cost of enabling a global work pool is that cut locality is not maintained any more.

A central global work queue is a questionable choice for ensuring a global steering of the BnB search. The obvious objection in a parallel execution context is that requiring all Tree Explorers to pull work from a single Stg1 Manager object can cause scaling bottlenecks. However, this concern is easily assuaged by a few observations. A relatively small number of Tree Explorers can generate enough candidate solutions to keep the Stg2 compute resources highly utilized. Hence the number of Tree Explorers will be too small to cause a performance bottleneck at the Stg1 queue. The coarse-grained Stg1 LP solves that necessarily separate two pulls by the same Tree Explorer from the global queue will also prevent the bottleneck. Finally, transitioning from a single global queue to a hierarchical queue is possible, but worth considering only in the case of evident bottlenecks.

Its worth noting that this design introduces a symmetry between the two stages. Both stages now maintain a global pool of work, and each compute object requests a manager object for units of work.

C. Performance and Analysis

Table II presents results we obtained from multiple trials of solving a 5t dataset at different scales. We observe that the large variation in performance persists. We also observe no discernible scaling when we increase the number of processors. This variation persists despite ensuring a globally enforced search policy. To investigate we plotted the BnB trees from two identically configured trials on a very small dataset. These trees are plotted in Figure 8 and demonstrate that despite a global work queue there is a large variation in the actual search tree. This obviously affects the amount of work done until a solution is found, and hence the time to solution.

VIII. DESIGN 2: IN SEARCH OF REPEATABILITY AND SCALABILITY

A. Design Motivations

Design 1-B demonstrates that enforcing a global search policy is insufficient in maintaining consistent search trees across multiple trials. The experiments also demonstrate that identically configured trials can yield vastly different search trees. We summarize a set of diagnostic experiments that we constructed to isolate the cause(s) of this variation.

If we momentarily ignore the Stg2 component of the formulation, there is one significant difference between a regular BnB search and the designs for tree exploration proposed thus far. This is the fact that the vertices in our designs share state. The Tree Explorers, which are responsible for exploring the BnB tree, have to necessarily (memory constraints) hold and explore several vertices during any span of time.

a) Diagnostic Experiment 1: As a first experiment, we isolate the vertices from each other in the following manner. We modify the Tree Explorers from the previous design and constrained them to operate on just one vertex at a time. While a candidate solution for that vertex is undergoing Stg2 evaluation, the Stg1 object simply idles. It makes another request for a vertex only when the current vertex has converged. The other modification is to ensure that LP library instances do not port state across solves for two different vertices. To realize this, we store a dump of all the cuts generated for the root vertex, reset the library instance and reload this same set of cuts whenever a Tree Explorer starts working on a new vertex. These actions completely eliminate the sharing of cuts across vertices and ensure that each vertex starts with the same deterministic characterization of the feasible space of solutions. However, experiments reveal that we still observe some variation across multiple identical trials.

We furthered the previous reasoning and proposed the hypothesis that sharing state in the Scenario Evaluators could affect the cuts that were generated for a candidate solution. Since, the Stg2 LP is degenerate, it has several possible solutions. A different start point could cause the LP to converge to a different solution. This would lead to different cuts being generated. Hence, the cuts for any given candidate solution potentially depended on the internal state of the LP. This meant that even if a vertex started with a deterministic collection of cuts, the cuts that were generated during its lifetime were influenced by the Stg2 LP solves of other vertices.

b) Diagnostic Experiment 2: A second experiment evaluated this hypothesis, by isolating even the scenario evaluations for a candidate solution from other Stg2 computations. Since this was purely diagnostic, we achieved this by inserting reset calls to the Stg2 LP library instances. This greatly increased the solve times, but ensured that each candidate solution was evaluated devoid of any external influences. Experiments revealed that the combination of the above two modifications

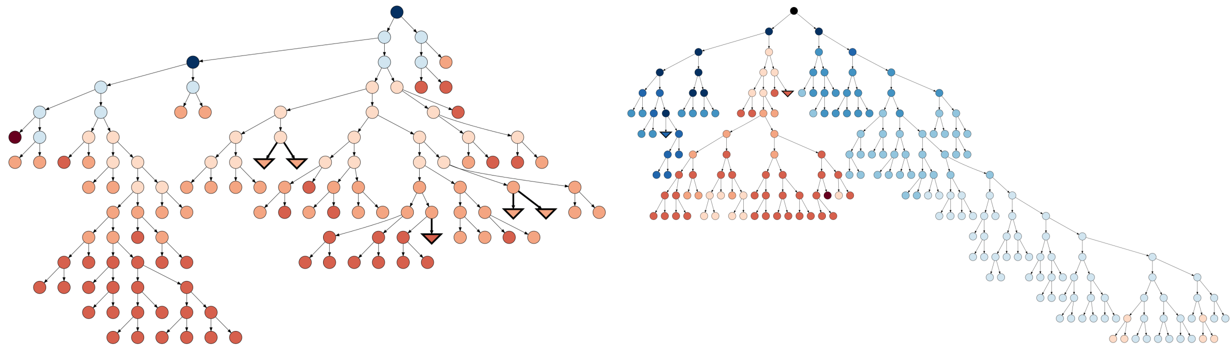

Fig. 8. The Branch-and-Bound trees from two identically configured executions of the same 5t dataset. The trees from the two trials area significantly different and explain the large variation in performance across trials. Triangles represent integer solutions (incumbents), while the vertices are colored by the value of bound

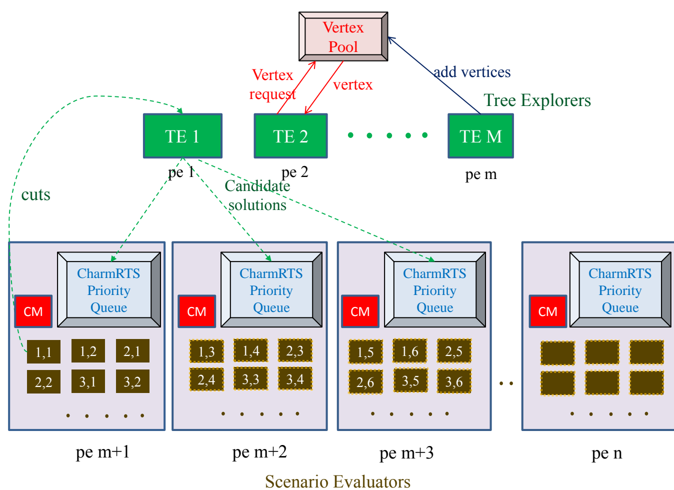

Fig. 9. Schematic of design that isolates BnB vertex explorations in order to reduce the large variations in performance observed with the previous two designs. This design is enabled by the availability of prioritized method invocation available in Charm++. Messages are tagged with the priority of the vertex they correspond to. The runtime system automatically picks the highest priority messages and invokes the corresponding methods.

resulted in the same (or very similar) search trees being generated across multiple trials.

These findings illustrate that a large part of the variation in search trees is simply because of the sharing of cuts and solver internal states. This motivates our final design.

B. Parallel Structure

The final design that we present is again obtained as a set of modifications to our previous design. We isolate computations to gain repeatability, and then oversubscribe objects to processors to combat the ensuing idle time. The set of modifications required atop the previous design are described below.

1) Isolated Tree Explorers: The new design seeks to isolate vertex explorations in a manner similar to that described in the diagnostic experiments. 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 Stg2 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 (depth-first), a fraction (bounds / cost), or a bitvector that identifies the vertex.

2) Cut Dump Manager: Our diagnostic experiments simply stored a dump of the cut collection from the root vertex and used that as a starting point for every vertex. However, this potentially repeats a lot of avoidable Stg1–Stg2 rounds to regenerate all the cuts that would have been generated by vertex's ancestors. This would dilate the time taken for a single vertex to converge. Our experiments do demonstrate that this is the case. Hence, in order to mitigate the effects of isolation we attempt to share cuts; but share them in a deterministic manner.

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. This should significantly offset the detrimental effects of isolating vertices. 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 front. 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.

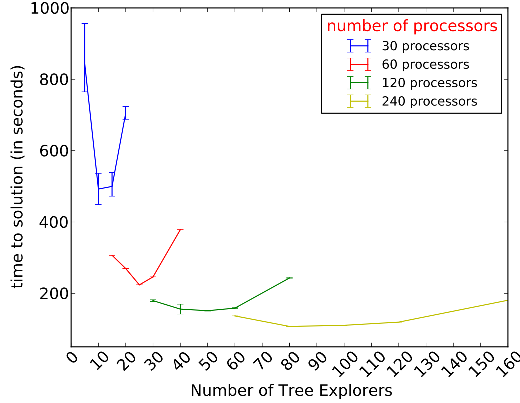

Fig. 10. The amount of Stg1 parallelism has an impact on solve times. Performance plots for a 3t dataset with 120 scenarios with different number of Tree Explorers at each scale. Each curve reflects the variation in performance at a given scale as we increase the number of Stg1 compute resources. Error bars reflect the variation in time to solution at each data point across multiple trials. The new design seems to exhibit lesser variability.

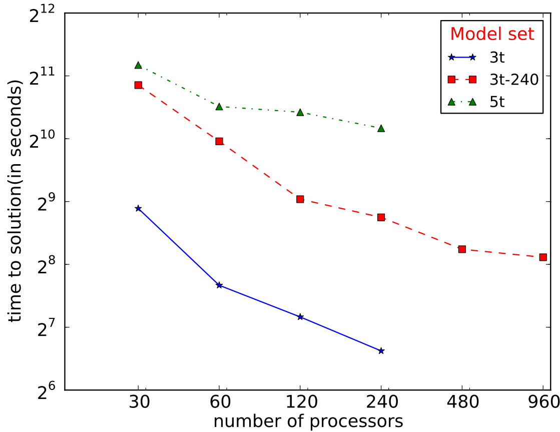

Fig. 11. Scaling plot of different model sets. Experiments were performed with varying number of Tree Explorers and the average of the various trials at the best performing distribution is plotted.

3) Isolated Scenario Evaluators: In order to isolate the Stg2 computations for each candidate solution, we dedicate a collection of Scenario Evaluators to each Tree Explorer. We remove the Stg2 Manager object and its pool of candidate solutions. Instead 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 Stg2 computations for the most important vertices are again achieved by simply tagging the messages with the priorities of the corresponding vertices.

C. Performance and Analysis

In Figure 10, we plot the performance of a 3t dataset at different execution scales and with different number of Tree Explorers. Each point is the average of multiple trials. Two observations are apparent. First, the amount of variation across multiple trials of a single configuration is significantly lower than in the previous designs. However it has not vanished completely. Second, the number of Tree Explorers at any given execution scale has a significant effect on the performance. However, with the reduced performance variation, it is now possible to reason about this effect. Expectedly, increasing the number of Tree Explorers too much inundates Stg2 with work and deteriorates performance. We have also ascertained that the concurrent execution of several Stg1 LPs on the same compute node of the machine increases the individual solve times because of memory bandwidth limitations.

Dataset Name	Design 1-B	Design 2
3t	0.053	0.442
5t	1.037	2.613

TABLE III
THE IMPACT OF CUT SHARING ON AVERAGE STG1 LP SOLVE TIMES (S)

Figure 11 presents the performance of this design when strong scaling several datasets. The data points represent the average times across multiple trials of the best performing configuration at each scale. The design is able to successfully deliver better performance at increasing core counts. For instance, the 3t-240 dataset scales perfectly from 30 to 120 cores, and then with an efficiency of 50% from 120 to 480 cores. We observe similar efficiencies for the other 3t dataset.

The behavior of the 5t dataset in the scaling experiments is of some concern. Again contrary to conventional wisdom, the larger dataset does not seem to scale better. However, scaling is limited by depth of the optimal vertex in the tree. The length of the path from the root to the optimal vertex defines a lower bound on the critical path length. The exploration of the path leading to the optimum cannot be parallelized further. For some datasets, we believe that the discovery of the optimum leads to all remaining vertices being pruned and hence, causes the optimal vertex to define the critical path in the computation. This places a lower bound on the time to solution and limits scaling.

Finally we evaluate the impact of isolating vertices in Stg1 at the expense of cut sharing. Figure 12 plots a histogram of the number of rounds taken by different vertices to converge. We compare the convergence behavior for: (*a*) design 1-A, where cuts are shared within a Tree Explorer, and cut locality is ensured (*b*) design 1-B, where cuts are shared, but cut locality is sacrificed for global prioritization (*c*) design 2 without the Cut Manager, where Stg1 solves always start from the root vertex cuts (*d*) design 2, where vertices are isolated, but cut dumps restore some cut sharing and locality. The original design permits vertices to converge in the fewest rounds because of the very good cut locality. Design 1-B takes more rounds to converge on vertices, as it sacrifices locality. The third plot shows the worst convergence behavior because it sacrifices both cut sharing and locality. This variant performs a lot of needless computation in regenerating cuts. The final design, which restores limited amounts of cut sharing and locality produces repeatable search trees, and displays convergence behavior only slightly worse than design 1-B. Thus the final design, in comparison to the original, trades

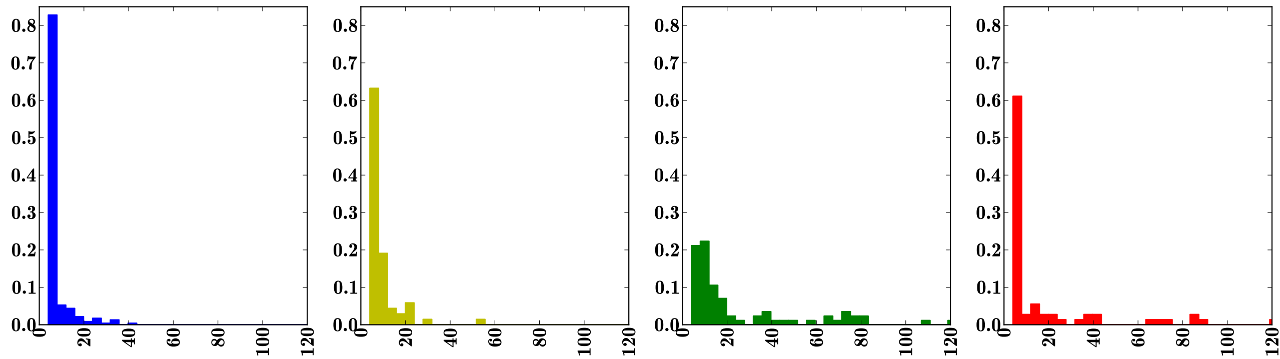

Fig. 12. Number of rounds to explore a vertex. Histogram. From left: design 1-A; design 1-B; design 2 without cut manager; design 2 with cut manager

extra computation for repeatability.

Table III shows the average Stg1 LP solve times for design 1-B and 2. We observe that the solve times are significantly larger for the final design. This implies that even if vertices take approximately the same number of rounds to converge in both designs, cut sharing across vertices helps find candidate solutions in lesser time. Hence the total Stg1 walltime required for a vertex to converge is lesser for design 1-B.

The previous histogram and table attempt to convey that the final design potentially suffers dilated times to solution in order to gain repeatability. However, we have insufficient data to make this claim with greater confidence. The choice between designs may depend on the application usage situations.

IX. Summary

We have presented three designs for a parallel, stochastic integer program using Branch-and-Bound. Our designs are the result of an evolutionary exploration and possess differing strengths. However, the need for scalable solutions to stochastic integer programs is best met by the third of our designs, which presents both repeatability and scalability. The design demonstrates sufficient scaling, and yields solutions in incrementally faster times upto 960 cores of a dual hex-core, 2.67 GHz, Intel Xeon cluster. We believe these are noteworthy results for strong scaling such an unconventional problem.

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

References

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

[2] N.V. Sahinidis. Optimization under uncertainty: state-of-the-art and opportunities. *Computers & Chemical Engineering*, 28(6):971–983, 2004.

[3] G. Laporte and F.V. Louveaux. The Integer L-shaped Method for Stochastic Integer Programs with Complete Recourse. *Operations research letters*, 13(3):133–142, 1993.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] Gurobi Optimization Inc. Gurobi Optimizer. Software, 2012. http://www.gurobi.com/welcome.html.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] Laxmikant Kale, Anshu Arya, Abhinav Bhatele, Abhishek Gupta, Nikhil Jain, Pritish Jetley, Jonathan Lifflander, Phil Miller, Yanhua Sun, Ramprasad Venkataraman, Lukasz Wesolowski, and Gengbin Zheng. Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge. Technical Report 11-49, Parallel Programming Laboratory, November 2011.