Performance Optimization of a Parallel, Two Stage Stochastic Linear Program

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

Steven Baker
MITRE Corporation
sbaker@mitre.org

***Abstract*—Stochastic optimization is used in several high impact contexts to provide optimal solutions in the face of uncertainties. This paper explores the parallelization of two-stage stochastic resource allocation problems that seek an optimal solution in the first stage, while accounting for sudden changes in resource requirements by evaluating multiple possible scenarios in the second stage. Unlike typical scientific computing algorithms, linear programs (which are the individual grains of computation in our parallel design) have unpredictable and long execution times. This confounds both a priori load distribution as well as persistence-based dynamic load balancing techniques. We present a master-worker decomposition coupled with a pull-based work assignment scheme for load balance. We discuss some of the challenges encountered in optimizing both the master and the worker portions of the computations, and techniques to address them. Of note are cut retirement schemes for balancing memory requirements with duplicated worker computation, and scenario clustering for accelerating the evaluation of similar scenarios.**

We base our work in the context of a real application: the optimization of US military aircraft allocation to various cargo and personnel movement missions in the face of uncertain demands. We demonstrate scaling up to 122 cores of an intel® 64 cluster; even for very small, but representative datasets.

Our decision to eschew problem-specific decompositions has resulted in a parallel infrastructure that should be easily adapted to other similar problems. Similarly, we believe the techniques developed in this paper will be generally applicable to other contexts that require quick solutions to stochastic optimization problems.

***Keywords*-stochastic optimization, parallel computing, large scale optimization, airfleet management**

I. Introduction

Stochastic optimization provides a means of coping with the uncertainty inherent in real-world systems; and with models that are nonlinear, of high dimensionality, or not conducive to deterministic optimization techniques. Deterministic approaches find optima for a fixed combination of inputs. However, the solutions obtained can be far from optimal even with small perturbations of the input data. This can be problematic because real-world systems often have many perturbations from mean values. Stochastic optimization allows the modeler to account for this randomness by looking for optimality across multiple possible scenarios [1]. Typically, the search for an optimum involves the evaluation of candidate solutions for many possible combinations or variations in input values (scenarios). Since, the number of likely or possible scenarios is typically quite large, there is a clear motivation to explore parallel computing to handle this large computational burden.

Stochastic optimization algorithms have applications in statistics, science, engineering, and business. Examples include making investment decisions in order to increase profit, transportation (planning and scheduling logistics), design-space exploration in product design, etc. There are other applications in agriculture, energy, telecommunications, military, medicine, water management etc.

In this paper, we describe our design for a parallel program to solve a 2-stage stochastic linear optimization model for an aircraft planning problem. We present our parallel decomposition and some interesting considerations in dealing with computation-communication granularity, responsiveness, and the lack of persistence of work loads in an iterative setting.

In Section II we briefly describe the aircraft allocation problem and its formulation as a two-stage stochastic program. In Section III we discuss our parallel program design for the Benders decomposition approach. In Section IV, we present challenges and strategies for optimizing the Stage 1 component of the computations while in Section V we present our study of the Stage 2 computations. Scalability results are presented in Section VI, while we summarize related work in Section VII.

II. Model Formulation & Approach

The United States Air Mobility Command (AMC) [1] manages a fleet of over 1300 aircraft [2] that operate globally under uncertain and rapidly changing demands. Aircraft are allocated at different bases in anticipation of the demands for several missions to be conducted over an upcoming time period (typically, fifteen days to one month). Causes of changes include demand variation, aircraft breakdown, weather, natural disaster, conflict, etc. The purpose of a stochastic formulation is to optimally allocate aircraft to each mission such that subsequent disruptions are minimized.

Aircraft are allocated by aircraft type, airlift wing, mission type and day. In situations when self-owned military aircraft are not sufficient for outstanding missions, civilian aircraft are leased. The cost of renting civilian aircraft procured in advance for the entire planning cycle is lower than the rent of civilian aircraft leased at short notice. Therefore, a good prediction of the aircraft demand prior to the schedule execution reduces the execution cost.

[1]http://www.amc.af.mil/

We model the allocation process as a two-stage stochastic linear program (LP) with Stage 1 generating candidate allocations and Stage 2 evaluating the allocations over many scenarios. This iterative method developed by Benders [3] has been widely applied to Stochastic Programming. Note that our formulation of the aircraft allocation model has complete recourse (i.e. all candidate allocations generated are feasible) because any demand (in a particular scenario) that cannot be satisfied by a candidate allocation is met by short term leasing of civilian aircraft at a high cost while evaluating that scenario. A detailed description of our model and the potential cost benefits of stochastic vs deterministic models is available elsewhere [4], [5]. To illustrate the size of the datasets of interest, Table I lists the sizes of various airlift fleet assignment models. $3t$ corresponds to an execution period of 3 days, $5t$ for 5 days, and so on.

In *Stage 1* , before a realization of the demands are known, decisions about long-term leasing of civilian aircraft are made, and the allocations of aircraft to different missions at each base location are also decided.

$$\min \quad Cx + \sum_{k=1}^{K} p_k \theta_k \tag{1}$$

$$s.t. \quad Ax \leq b, \tag{2}$$

$$E_l x + \theta \leq e_l \tag{3}$$

In the objective function(1), x corresponds to the allocations by the aircraft type, location, mission and time. C is the cost of allocating military aircraft and leasing civilian aircraft. $\theta = \{\theta_k | k = 1, ..., k\}$ is the vector of Stage 2 costs for the k scenarios, p_k are the probability of occurrence of scenario k, l corresponds to the iteration in which the constraint was generated and $E_l(e_l)$ are the coefficients (right hand sides) of the corresponding constraints. Constraints in (2) are the feasibility constraints, while constraints in (3) are cuts which represents an outer linearization of the recourse function.

In *Stage 2* , the expected cost of an allocation for each scenario in a collection of possible scenarios is computed by solving LPs for that scenario.

$$\min \quad {q_k}^T y \tag{4}$$

$$s.t. \quad Wy \leq h_k - T_k x \tag{5}$$

The second stage optimization helps Stage 1 to take the recourse action of increasing the capacity for satisfying an unmet demand by providing feedback in the form of additional constraints (cuts) on the Stage 1 LP (6). Here, π_k are the dual multipliers obtained from Stage 2 optimization and x^* is the allocation vector obtained from the last Stage 1 optimization.

$$\theta_k \leq \pi_k * (h_k - T_k x^*) - \pi_k T_k (x - x^*) \tag{6}$$

TABLE I
MODEL SIZES OF INTEREST (120 SCENARIOS)

Model Name	Num stg1 variables	Num stg2 variables	Num stg2 constraints
3t	255	1076400	668640
5t	345	1663440	1064280
10t	570	3068760	1988640
15t	795	4157040	2805000
30t	1470	7956480	5573400

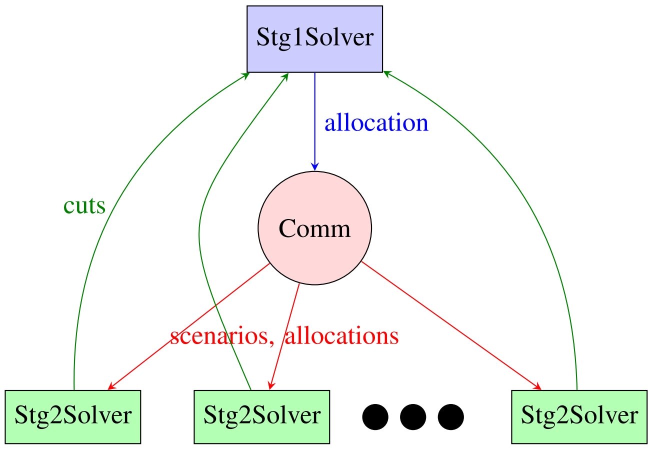


Fig. 1. Design Schematic

III. Parallel Program Design

a) Programming Model: We have implemented the program in Charm++ [6], [7], which is a message-driven, object-oriented parallel programming framework with an adaptive run-time system. It allows expressing the computations in terms of interacting collections of objects and also implicitly overlaps computation with communication. Messaging is one-sided and computation is asynchronous, sender-driven; facilitating the expression of control flow which is not bulk synchronous (SPMD) in nature.

b) Coarse Grained Decomposition: To exploit the state of the craft in LP solvers, our design delegates the individual LP solves to a library (Gurobi [8]). This allows us to build atop the domain expertise required to tune these numerically intensive algorithms. However, the same decision also causes a very coarse-grain of computation as the individual solves are not decomposed further. Parallel programs usually benefit from a medium or fine-grained decomposition as it permits a better overlap of computation with communication. In Charm++ programs, medium-sized grains allow the runtime system to be more responsive and give it more flexibility in balancing load. Adopting a coarse-grained decomposition motivates other mitigating design decisions described here. It also emphasizes any sequential bottlenecks and has been causative of some of our efforts in optimizing solve times.

c) Two-stage Design: Since the unit of sequential computation is an LP solve, the two-stage formulation maps readily onto a two-stage parallel design, with the first stage generating candidate allocations, and the second stage evaluating these allocations over a spectrum of scenarios that are of interest. Feedback cuts from the second stage LPs guides the generation of a new candidate allocation. There are many such iterations (rounds) until an optimal allocation is found. We express this as a master-worker design in Charm++ with two types (C++ classes) of compute objects. An *Allocation Generator*

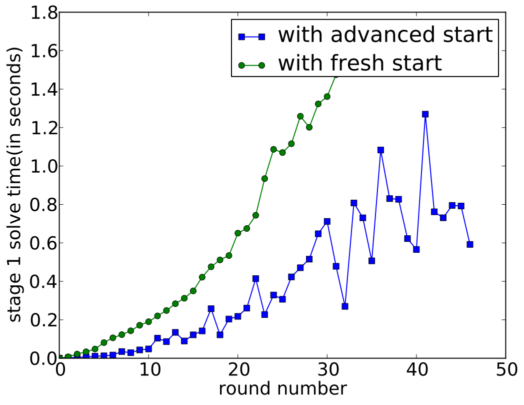


Fig. 2. Stage 1 LP solve times with and without advanced start on 2.67 GHZ Dual Westmere Xeon

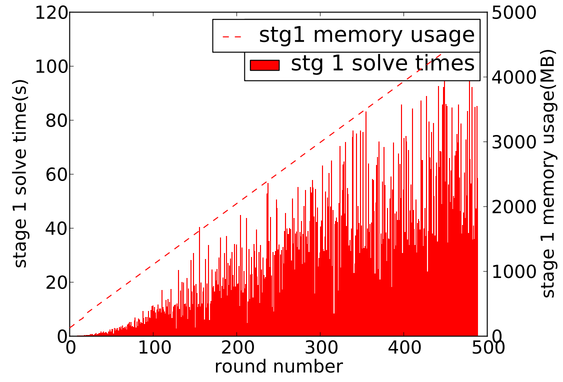


Fig. 3. Stage 1 memory usage and LP solve times for 15 time period model on Dell 2.6 GHz Lisbon Opteron 4180

object acts as the master and generates allocations, while a collection of *Scenario Evaluator* objects are responsible for the evaluation of all the scenarios.

d) Unpredictable Grain Sizes: Experiments show that LP solves for different scenarios take different amounts of time. Hence, an a priori static distribution of scenarios across all the *Scenario Evaluators* will not achieve a good load balance. Unlike typical algorithms in parallel, scientific computing, the time taken for an individual grain of computation (LP solve) is also devoid of any persistence across different iterations (rounds). This precludes the use of any persistence-based dynamic load balancers available in Charm++. To tackle this fundamental unpredictability in the time taken for a unit of computation we adopt a work-request or pull-based mechanism to ensure load-balance. We create a separate work management entity, *Work Allocator* object(*Comm* in Figure 1), that is responsible for doling out work units as needed. As soon as a *Scenario Evaluator* becomes idle, it sends a work request to the *Work Allocator* which assigns it an unevaluated scenario. Figure 1 is a schematic representing our design.

e) Maintaining Responsiveness: A pull-based mechanism to achieve load balance requires support from a very responsive *Work Allocator*. Charm++ provides flexibility in the placement of compute objects on processors. We use this to place the *Allocation Generator* and the *Work Allocator* objects on dedicated processors. This ensures a responsive *Work Allocator* object and allows fast handling of work requests from the *Scenario Evaluators*; unimpeded by the long, coarse-grained solves that would otherwise be executing.

IV. Optimizing Stage 1

f) Advanced Starts: The two-stage design yields an allocation that is iteratively evolved towards the optimal. Typically, this results in LPs that are only incrementally different from the corresponding LPs in the previous round as only a few additional constraints may be added every round. LP solvers can exploit such situations by maintaining internal state from a call so that a later call may start its search for an optimum from the previous solution. This is called advanced start (or warm start), and can significantly reduce the time required to find a solution to an LP. We enabled advanced starts for the Stage 1 LP and observed sizable performance benefits (Figure 2).

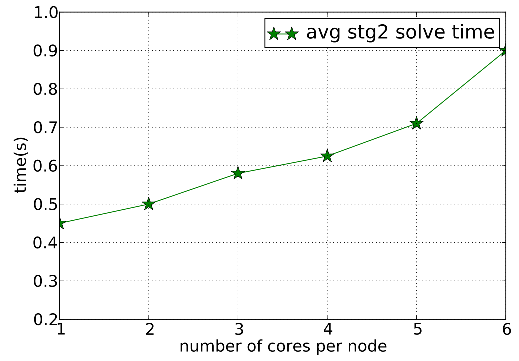


Fig. 4. The impact of artificially constraining memory bandwidth available for an LP solve (10 time period model) on a system with Intel 64(Clovertown) 2.33 GHz dual socket quad core processor with 1333MHz front size bus (per socket), 2x4MB L2 cache and 2 GB/core memory.

g) Memory Footprint and Bandwidth: An observation from Figure 2 is that the Stage 1 solve time increases steadily with the round number irrespective of the use of advanced

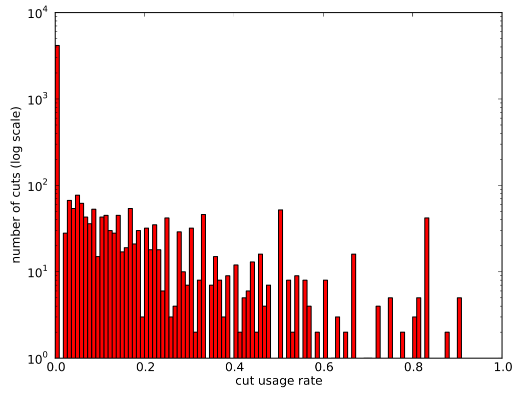


Fig. 5. Cut usage rate for a 5 time period model

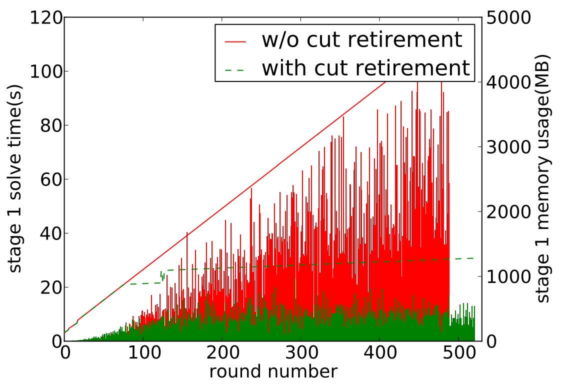


Fig. 6. Stage 1 LP solve times and memory usage for the 15 time period model solved to 1% convergence with *Cut Window* of 75 (run on 8 cores of 2.6 GHz Lisbon Opteron 4180)

starts. Our investigation pointed to an increasing solver memory footprint as the cause for such behavior.

During each round, the *Allocation Generator* incorporates feedback from the evaluation of each scenario into the Stage 1 model. This feedback is in the form of constraints (cuts) which are additional rows added to a matrix maintained internally by the library. The number of cuts added to the model grows monotonically with the number of rounds; requiring an increasing amount of memory to store and solve an LP. Figure 3 captures this trend by plotting memory utilization for the *Allocation Generator* object (which includes LP library memory footprint) and the time taken for the Stage 1 solves by round number. The memory usage is as high as 5 GB and the solve time for a single grain of Stage 1 computation can reach 100s.

To improve the characterization of the LP solves, we designed an experiment that artificially limits the memory bandwidth available to a single LP solver instance by simultaneously running multiple, independent LP solver instances on a multicore node. Our results (Figure 4) show that for the same problem size, the time to solution of an LP is increased substantially by limiting the available memory bandwidth per core. As the Stage 1 model grows larger every round, it becomes increasingly limited by the memory subsystem and experiences dilated times for LP solves.

h) Curbing Solver Memory Footprint: For large Stage 1 problems, which take many iterations to converge, the increasing Stage 1 solve times and the increasing memory demands exacerbate the serial bottleneck at the *Allocation Generator*, and pose a threat to the very tractability of the Benders approach. However, an important observation in this context is that not all the cuts added to a Stage 1 problem may actually constrain the feasible space in which the optimum solution is found. As new cuts are added, older cuts may no longer be binding or active. They may become active again in a later round or maybe rendered redundant if they are dominated by newer cuts. Such cuts simply add to the size of the Stage 1 model and its solve time, and can be safely discarded. Figure 5 plots a histogram of the cut usage rate (defined by equation 7) for the cuts generated during the course of convergence of a 5 time period model. Most of the cuts have very low usage rates while a significant number of the cuts are not used at all. This suggests that the size of the Stage 1 problem may be reduced noticeably without diluting the description of the feasible space for the LP solution.

$$\text{Cut Usage Rate} = \frac{\text{num rounds in which cut is active}}{\text{num rounds since its generation}} \qquad (7)$$

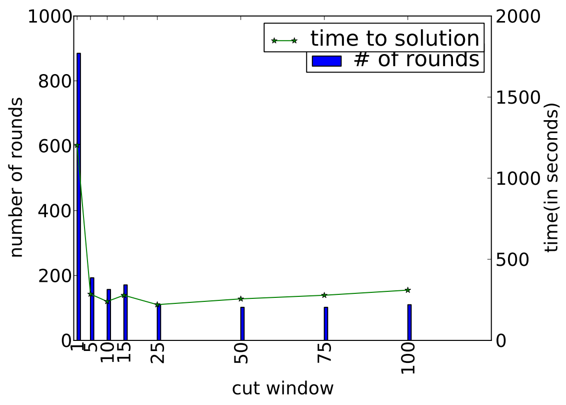
We therefore implemented a cut retirement scheme that discards/retires cuts whenever the total number of cuts in the Stage 1 model exceeds a configurable threshold. After every round of the Benders method, the cut score is updated based on it's activity in that round. Cuts with small usage rates (defined by Equation 7) are discarded. The desired number of lowest scoring cuts can be determined using a partial sort that runs in linear time.

Discarding a cut that may be required during a later round only results in some repeated work. This is because the Benders approach will cause any necessary cuts to be regenerated via scenario evaluations in future rounds. This approach could increase the number of rounds required to reach convergence, but lowers execution times for each Stage 1 LP solve by limiting the required memory and access bandwidth. Figure 6 demonstrates these effects and shows the benefit of cut management on the Stage 1 memory usage and solve times of the 15 time period model solved to 1% convergence tolerance. The time to solution reduced from 19025s without cut retirement to 8184s with cut retirement - a 57% improvement.

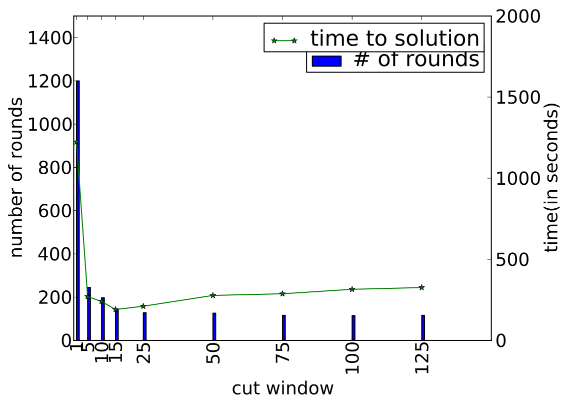
We define a *Cut Window* as the upper limit on the number of cuts allowed in the Stage 1 model, expressed as the maximum number of cuts divided by the number of scenarios. Figure 7(a) and 7(b) describe the effect of different *Cut Window*s on the time and number of rounds to convergence. Smaller *Cut Window*s reduce the individual Stage 1 solve times, leading to an overall improvement in the time to solution even though it takes more rounds to converge. However, decreasing the *Cut Window* beyond a certain limit, leads to a significant increase in the number of rounds because several useful cuts are discarded and have to be regenerated in later rounds. Further reducing the *Cut Window* makes it impossible to converge because the collection of cuts is no longer sufficient. These experiments demonstrate the need to make an informed choice of the *Cut Window* to get the shortest time to solution, e.g. for the 5 time period model with 120 scenarios, an optimal *Cut Window* size is close to 25 while for the 10 time period model with 120 scenarios it is close to 15.

i) Evaluating Cut-Retirement Strategies: We investigate cut management further to study it's performance with different cut scoring schemes. Three cut scoring schemes are discussed here namely, the least frequently used, the least recently used and the least recently/frequently used. Each of these are briefly discussed here: leftmargin=3mm

- *Least Frequently Used (LFU)* A cut is scored based on it's rate of activity since it's generation (equation7).



(a) 5 time period model (solved to 0.1% convergence on 8 cores of 2.26 GHz Dual Nehalem)



(b) 10 time period model (solved to 1% convergence on 32 cores of 2.67 GHz Intel Xeon hex-core processors)

Fig. 7. Performance of 5 and 10 time period models with different *Cut Windows*

This scoring method was used for results presented in Figure 7(a) and 7(b).

- *Least Recently Used (LRU)* - In this scheme, the recently used cuts are scored higher. Therefore, a cut's score is simply the last round in which it was active.

$$LRU\_Score = \text{Last active round for the cut}$$

- *Least Recently/Frequently Used (LRFU)* This scheme takes both the recency and frequency of cut activity into account. Each round in which the cut was active contributes to the cut score. The contribution is determined by a weighing function $\mathcal{F}(x)$, where x is the time span from the activity in the past to current time.

$$LRFU\_Score = \sum_{i=1}^{k} \mathcal{F}(t_{base} - t_i)$$

where $t_1, t_2, ..., t_k$ are the active rounds of the cut and $t_1 < t_2 < ... < t_k \leq t_{base}$. This policy can demand a large amount of memory if each reference to every cut has to be maintained and also demands considerable computation every time the cut retirement decisions are made. Lee, et. al. [9] have proposed a weighing function $\mathcal{F}(x) = (\frac{1}{p})^{\lambda x}$ $(p \geq 2)$ which reduces the storage and computational needs drastically. They tested it for cache replacement policies and obtained competitive results. With this weighing function, the cut score can be calculated as follows:

$$S_{t_k} = \mathcal{F}(0) + \mathcal{F}(\delta) S_{t_{k-1}},$$

where S_{t_k} is the cut score at the kth reference to the cut, $S_{t_{k-1}}$ was the cut score at the $(k-1)$th reference and $\delta = t_k - t_{k-1}$. For more details and proofs for the weighing function refer to [9]. We use $p = 2$ and $\lambda = 0.5$ for our experiments.

Figure 8 compares the result of these strategies. LRFU gives the best performance of the three. The *cut windows* used for these experiments were the optimal values obtained from experiments in Figure 7(a) and 7(b).

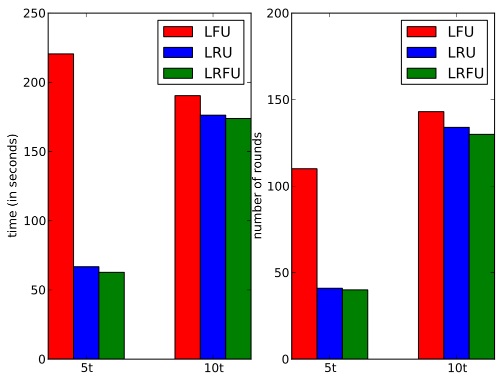


Fig. 8. Performance of different cut scoring strategies for the 5 time period model(8 cores, cut-window=25, 0.1% convergence) and the 10 time period model(32 cores, cut-window=15, 1% convergence)

V. Optimizing Stage 2

j) Advanced Starts: In every iteration, there are as many Stage 2 LP solves as there are scenarios. This constitutes the major volume of the computation involved in the Benders approach because of the large number of scenarios in practical applications. Even a small reduction in the number of rounds or average Stage 2 solve times can have sizable payoffs. In this section, we analyze different strategies to reduce the amount of time spent in Stage 2 work.

In contrast to the Stage 1 LP solves, Stage 2 solves take more time with the advanced start feature as compared to a fresh start. This can happen because the initial basis from the previous scenario solve can be a bad starting point for the new scenario. Despite the slower solves with advanced starts, our experiments show that runs with advanced start take fewer rounds to converge than with a fresh start. This indicates that starting from the previous solves gives us better cuts. This behavior was seen across several input datasets. We do not yet have data to back any line of reasoning that can explain

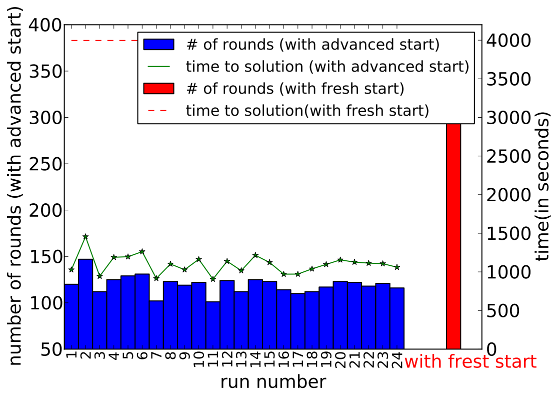


Fig. 9. Variation across runs with advanced-start and their comparison with fresh start ($10t$ model) on 8 cores of 2.67 GHz Dual Nehalem

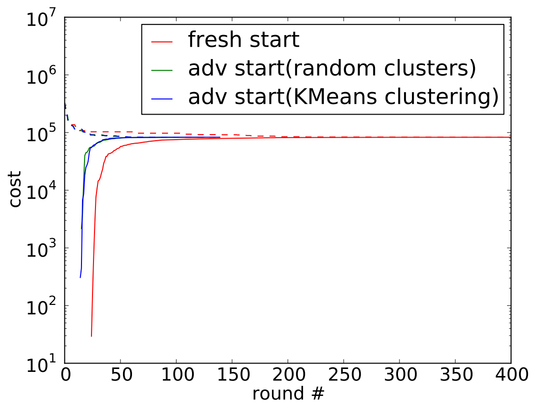


Fig. 11. Bound convergence rate comparison between Stage 2 fresh start, advanced start with clustering and advanced start without clustering

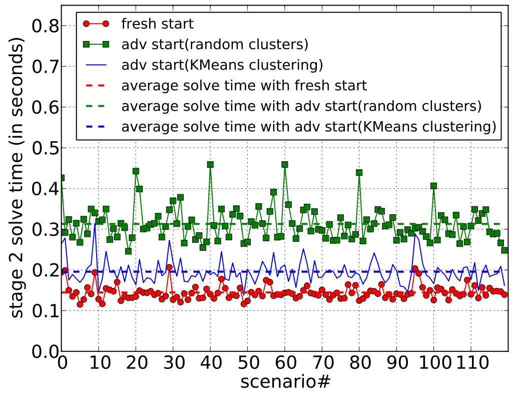


Fig. 10. Comparison of average Stage 2 solve time between Stage 2 fresh start, advanced start with clustering and advanced start without clustering on 2.6 GHz AMD Lisbon Opteron

this. Figure 9 shows that runs with Stage 2 fresh starts took 300 rounds to converge as compared to just $100 - 150$ rounds with advanced start. Consequently, the total time to solution with advanced start is much less than with Stage 2 reset despite slower Stage 2 LP solves.

k) Variability Across Runs: A Note: Figure 9 also shows the number of rounds and time to solution for 25 runs on the same model. An interesting note is the variability across various runs of the same program.

Scenarios are assigned to *Scenario Evaluators* in the order in which work requests are received. This varies across different runs because of variable message latencies and variable LP solve times. With advanced starts, this results in different LP library internal states as starting points for a given scenario evaluation; yielding different cuts for identical scenario evaluations across different runs. This variation in cuts affects the next generated allocation from Stage 1 and the very course of convergence of the execution.

Variation across different runs make it difficult to measure the effect of different optimization strategies. Additionally in some situations, obtaining an optimal solution in predictable time can be more important than obtaining it in the shortest possible time. Therefore, mitigating the variability can be an important consideration.

Note that to verify that multiple solutions are not the artifact of a loose termination criteria, we solved the problems to very tight convergence criteria (up to 0.00001%). Identical runs resulted in different solutions implying that the problem is degenerate.

l) Clustering Similar Scenarios: Turning off the advanced start feature can significantly increase the time to solution and hence is not a viable approach. However, the scenario evaluation order can be pre-determined by assigning a fixed set of scenarios to each solver. This approach can potentially decrease the efficiency of the work-request mechanism at balancing Stage 2 load because work is now assigned in larger clusters of scenarios.

However, since some scenarios may exhibit similarities, it may be possible to group similar scenarios together to increase the benefits of advanced starts. It may be beneficial to trade coarser units of work-assignment (poorer load balance) for reduced computation grain sizes. We explore this by implementing scenario clustering schemes and cluster-based work assignment. Similarity between scenarios can be determined either by using the demands in each scenario, the dual variable values returned by them, or by a hybrid of demands and duals. Our current work has used the demands to cluster scenarios because they are known a priori– before the stochastic optimization process begins. We use a k-means [10] algorithm for clustering scenarios. Since, the clusters returned from k-means can be unequal in size, we use a simple approach (described in Algorithm 1) to migrate some scenarios from over-sized clusters to the under-sized clusters. We also implement random clustering for reference. Figure 10 compares the improvement in average Stage 2 solve times when scenarios are clustered using Algorithm 1.

However, our results show that random clustering gives faster convergence rates than k-means clustering (Figure 11). Despite faster Stage 2 solves, clustering does not give us shorter times to solution (Figure 12). These experiments point to a need for further studies to understand the impact of advanced starts on cut quality, to develop more robust

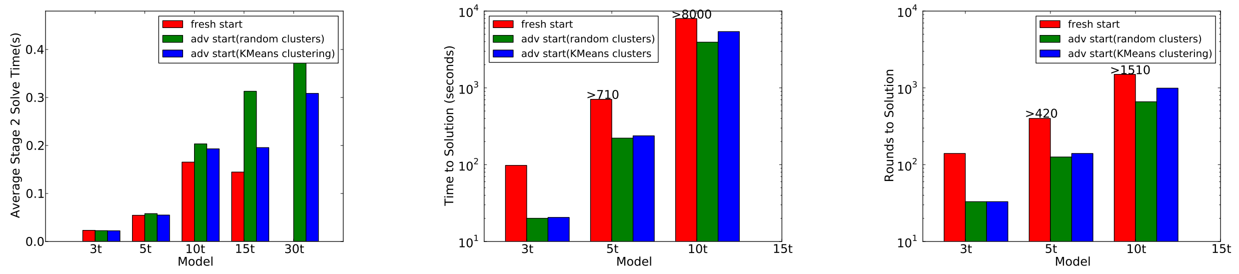


Fig. 12. Comparing the effect of advanced start and clustering on performance of different models on 8 cores of 2.67 GHz AMD Lisbon Opteron

Algorithm 1 The Scenario Clustering Algorithm

```
Input
$D_i$- Demand set for scenario $i$ ($i = 1, 2, ...., n$)
$k$ - number of clusters
Output
$k$ equally sized clusters of scenarios
Algorithm
{label, centroids} = kMeans({$D_1$, $D_2$, $D_3$, ..., $D_n$}, k)
IdealClusterSize = $\frac{n}{k}$
$size_i$ = size of cluster $i$
{Identify Oversized clusters}
$\mathcal{O} = \{c \in Clusters \mid size_c > IdealClusterSize\}$
{Identify Undersized clusters}
$\mathcal{U} = \{c \in Clusters \mid size_c < IdealClusterSize\}$
$\mathcal{S}$: set of adjustable points
for $c \in \mathcal{O}$ do
    Find $(size_i - IdealClusterSize)$ points in cluster $c$ that
    are farthest from $centroid_c$ and add them to the set $\mathcal{S}$
end for
while $size(\mathcal{S}) > 0$ do
    Find the closest pair of cluster $c \in (U)$ and point $p \in \mathcal{S}$
    Add $p$ to cluster $c$
    Remove $p$ from $\mathcal{S}$
    if $size_c == IdealClusterSize$ then
        Remove $c$ from $\mathcal{U}$
    end if
end while
```

scenario similarity metrics; and to investigate other clustering algorithms that may yield better results.

VI. Scalability

With the optimizations described above, we were able to scale medium-sized problems up to 122 cores of an Intel-64 Clovertwon (2.33 GHz) cluster with 8 cores per node. For 120 scenarios, an execution that uses 122 processors represents the limit of parallel decomposition using the described approach: one Stage 1 object, one *Work Allocator* object, and 120 *Scenario Evaluators* that each solve one scenario. Figure 13(a) and 13(b) show the scalability plots with Stage 1 and Stage 2 wall time breakdown. The plots also demonstrate Amdahl's effect as the maximum parallelism available is proportional to the number of scenarios that can be solved in parallel, and scaling is limited by the sequential Stage 1 computations. It must be noted that real-world problems may involve several hundreds or thousands of scenarios, and our current design should yield significant speedups because of Stage 2 parallelization.

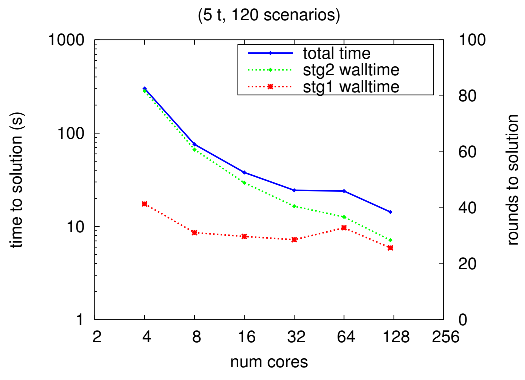
VII. Related Work

Stochastic linear programs can be solved using the extensive formulation(EF) [1]. Extensive formulation of a stochastic program is its deterministic equivalent program in which constraints from all the scenarios are put together in a single large scale linear program. e.g. the extensive formulation for a stochastic program corresponding to Stage 1 given in equations 1, 2, 3 and Stage 2 in equations 4, 5 can be written as:

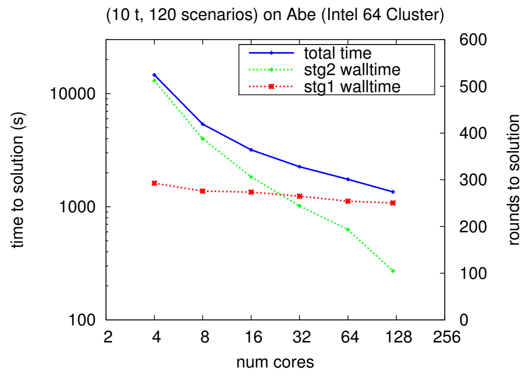
$$\begin{array}{ll} \min & c^T x + \sum_{k=1}^{K} p_k q_k^T y_k \\ s.t. & Ax = b, \\ & T_k x + W_{y_k} = h_k, \qquad k = 1, ..., K \\ & x \geq 0, y_k \geq 0, \qquad k = 1...., K \end{array}$$

EF results in a large linear program that quickly becomes too large to be solved by a single computer. However, liner program solvers are hard to parallelize, and other parallelization approaches become necessary. Recently, there has been some work on parallelization of the simplex algorithms for linear programs with dual block-angular structure [11]. Lubin et al [12] demonstrated how emerging HPC architectures can be used to solve certain classes of power grid problems, namely, energy dispatch problems. Their PIPS solver is based on the interior-point method and uses a Schur's complement to obtain a scenario-based decomposition of the linear algebra. However, in our work we choose not to decompose the LP solves, but instead delegate them to a library. This reuses domain expertise encapsulated in the library and allows performance specialists to focus just on parallel performance. Using a library also allows the implementation to remain more general with the ability to use it for other problems.

Linderoth, et. al. [13] have studied the performance of two-stage stochastic linear optimizations using the L-shaped algorithm on distributed grids. Unlike modern supercomputers, grids have high communication latencies and availability of



(a) 5 time period problem solved to 0.1% convergence



(b) 10 time period problem solved to 0.1% convergence

Fig. 13. Scalability plots for 5 and 10 time period models

nodes is sporadic. Hence, their work focuses on performance of an asynchronous approach to the Benders decomposition. In contrast, our work is based on a synchronous approach where a new iteration is initiated only after completion of all the scenario solves from the previous iteration.

VIII. Summary

Most stochastic programs incorporate a large number of scenarios to hedge against many possible uncertainties. Therefore, Stage 2 work constitutes a significant portion of the total work done in stochastic optimizations. For stochastic optimization with Benders approach, the vast bulk of computation can be parallelized using a master-worker design described in this paper. We have presented experiments, diagnoses and techniques that aim to improve the performance of each of the two stages of computation.

We presented an *LRFU* based cut management scheme, that completely eliminates the memory bottleneck and significantly reduces the Stage 1 solve time, thus making the optimization of large scale problems tractable. We analyzed different aspects of the Stage 2 optimization and have presented some interesting avenues for further studies in improving Stage 2 performance. With our techniques, we were able to obtain a speedup of about 21 and 11 for the 5 and 10 time period problems, respectively with 120 scenarios each as we scaled from 4 cores to 122 cores. Much higher speedups can be obtained for real-world problems which present much more Stage 2 computational loads. In our current design, Stage 1 still presents a serial bottleneck that inhibits the efficiency of any parallel implementation. We are currently exploring methods such as Lagrangean decomposition to alleivate this. We believe that some of our strategies can be applied to other stochastic programs too; and that this work will be of benefit to a larger class of large, commercially relevant, high impact stochastic problems.

IX. Acknowledgments

The research was supported by MITRE Research Agreement Number 81990 with UIUC. The Gurobi linear program solver is licensed at no cost for academic use. Runs on *A*be cluster were done under the TeraGrid [14] allocation grant ASC050040N supported by NSF.

References

[1] J. Birge and F. Louveaux. *Introduction to stochastic programming*. Springer, 2011.
[2] Air Mobility Command. Air Mobility Command Almanac 2009. Retrieved 12 Sep 2011. *http://www.amc.af.mil/shared/media/document/AFD-090609-052.pdf.*
[3] J.F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische mathematik*, 4(1):238–252, 1962.
[4] Steven Baker, Udatta Palekar, Gagan Gupta, Laxmikant Kale, Akhil Langer, Mark Surina, and Ram Venkataraman. Parallel Computing for DoD Airlift Allocation. MITRE Technical Report, 2012. www.mitre.org/work/tech_papers/2012/11_5412/.
[5] Akhil Langer. Enabling Massive Parallelism for Two-Stage Stochastic Integer Optimizations: A Branch and Bound Based Approach. Master's thesis, Dept. of Computer Science, University of Illinois, 2011. http://charm.cs.uiuc.edu/media/11-57.
[6] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.
[7] 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.
[8] Gurobi Optimization Inc. Gurobi Optimizer. Software, 2012. http://www.gurobi.com/welcome.html.
[9] C.S. Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 50(12), 2001.
[10] JA Hartigan and MA Wong. Algorithm AS 136: A K-means clustering algorithm. *Applied Statistics*, pages 100–108, 1979.
[11] Parallel Distributed-Memory Simplex for Large-scale Stochastic LP Problems. Preprint ANL/MCS-P2075-0412, Argonne National Laboratory, Argonne, IL. April 2012. www.optimization-online.org/DB_HTML/2012/04/3438.html.
[12] Miles Lubin, Cosmin G. Petra, Mihai Anitescu, and Victor Zavala. Scalable Stochastic Optimization of Complex Energy Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 64:1–64:64, New York, NY, USA, 2011. ACM.
[13] J. Linderoth and S.J. Wright. Computational Grids for Stochastic Programming. *Applications of stochastic programming*, 5:61–77, 2005.
[14] Charlie Catlett et al. TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications. In Lucio Grandinetti, editor, *HPC and Grids in Action*, volume 16, pages 225–249, Amsterdam, 2007. IOS Press.