Chapter 1

Computational Grids for Stochastic Programming

***Jeff Linderoth*[*], *Stephen J. Wright*[†]**

Abstract

We discuss the use of a computational grid—a large, networked collection of computers—in solving many-scenario two-stage stochastic linear programs with recourse. We describe the characteristics of the grid platform in some detail, and then discuss parallel algorithms and their implementation on this platform. We illustrate the power of the resulting software tool by presenting computational results obtained on some very large sampled instances of standard problems from the literature.

[*] Axioma Inc., 501-F Johnson Ferry Road, Suite 450, Marietta, GA 30068; jlinderoth@axiomainc.com

[†] Computer Sciences Department, 1210 West Dayton Street, University of Wisconsin, Madison, WI 53706; swright@cs.wisc.edu

1.1 Introduction

Parallel supercomputers continue to increase in power and in ability to solve very large and complex problems in computational science. For many users, however, there are a number of practical limitations associated with these machines, including their high cost, the difficulty of obtaining access to them, and the difficulty of writing or procuring software tools that execute on them. In recent years, there has been a good deal of interest in alternative computing platforms known as *computational grids* [9], which are made up of large collections of geographically dispersed CPUs, storage, and visualization devices linked by local networks and the Internet. Of particular interest to the optimization community are computational grids that are made up of workstations, PCs and PC clusters, and supercomputer nodes, and which may be owned by a number of different individuals and institutions. Grids such as these grant access to compute cycles that would not otherwise be used by the owners of the machines of which they are composed, without interfering with the computing activities of the machine owners. As we discuss in the next section, computational grids have some obvious drawbacks that make them difficult to use. Nevertheless, the tremendous power and low cost of these platforms makes them appealing for large-scale computations of certain types, as has been demonstrated in the popular `SETI@home` project [36]. Recently, commercial ventures have arisen to build and harness computational grids [8].

Perhaps more than any other area of computational optimization, stochastic programming has made use of parallel computing to solve large problem instances; see [2, 7, 32, 20, 19, 3, 29, 12, 21, 10]. In this earlier work, different problem types and different algorithms were matched to particular parallel platforms in different ways, with decomposition according to scenario a recurring theme in many approaches. In this paper, we describe the characteristics of the type of grid outlined in the previous paragraph, and point out in broad terms why its properties are a good match for stochastic programming problems and algorithms. We focus in particular on grids based on the Condor system [**?**], and on the MW runtime support library for supporting master-worker computations on these grids [13]. We then discuss our asynchronous decomposition algorithm for two-stage stochastic linear programs with recourse, and describe in some detail its implementation on the grid, using Condor and MW. We present computational results obtained for some very large sampled instances of stochastic programs from the literature, including comparisons with results obtained for other software tools. Finally, we describe some future enhancements to grid infrastructure that will equip them to solve other classes of stochastic programs, including stochastic integer programs.

1.2 Computational Grids

The term "grid computing" (synonymously "metacomputing") is generally used to describe parallel computations performed on a geographically distributed, heterogeneous computing platform. The particular variant of this concept of interest to us is a parallel platform made up of workstations, PCs and PC clusters, and supercomputers, owned by various individuals and entities and distributed across a campus or other organization, or across the world. Although such platforms are powerful and inexpensive, they are difficult to harness for productive use, for the following reasons:

- Heterogeneity. Resources may vary widely in their operational characteristics: memory, swap space, processor speed, operating system, installed software.
- Poor communications properties. Latencies (times to pass messages between processors) may be high, variable, and unpredictable.
- Unreliability. Resources may disappear without notice; for example, a workstation may be reclaimed by its owner, resulting in termination of any computations being performed by us on that machine.
- Dynamic availability. The pool of available processors grows and shrinks during the computation, according to the changing priority of our job, the claims of other users, and scheduling considerations.

In all these respects, our target platform differs from conventional multiprocessor platforms (such as IBM-SP or SGI Origin machines) and from Beowulf clusters. However, applications developed to run on computational grids often can be executed efficiently on these more conventional platforms also.

In this section, we describe the Condor system that forms the basis of the computational grid we used in our project. We then discuss the suitability of stochastic optimization for solution on grids. Finally, we outline the software tool MW that enables the implementation of master-worker algorithms on grids.

1.2.1 Condor

The Condor system [6, 25] manages distributively-owned collections ("pools") of processors spread across a campus or other organization. The owner of each machine in a Condor pool specifies the conditions under which jobs of other users are allowed to run on their machine. Most owners require Condor to terminate any job it is running on their machine whenever they start to use the machine themselves. Thus, Condor intrudes minimally on the owner's use of their machine, while putting the computational cycles that would otherwise have been wasted to work for other users. Users have little to lose by donating their machines to a pool, and much to gain in terms of access to the Condor system when they want to make use of the pool for their own purposes. Organization-wide Condor pools have been assembled at a number of universities and research centers (see Table 1.2 for the locations used in one of our runs).

When a user submits a job to Condor, the system finds an available processor in the pool with the right architecture and capabilities, and starts executing the job on that machine. Condor periodically checkpoints the state of the job, so that if the current host becomes unavailable (for example, if the machine is reclaimed by its owner), then job can be migrated and restarted from the latest checkpoint on a different processor in the pool. Condor-managed processes can communicate with the workstation from which they were submitted through a Condor-enabled version of the popular message passing library PVM [11], or by writing and reading shared files. The submitting workstation can submit more than one job to the Condor pool at a time—a fact which makes it possible to perform parallel computing on this platform.

Parallel algorithms with the following characteristics are in principle well suited for execution on the Condor system.

- *Algorithms that can be expressed in the master-worker framework.* The Condor setup supports this framework naturally by executing the master process on the submitting workstation and the worker processes on different hosts in the Condor pool. Workers communicate with the master process via the mechanisms mentioned above.

- *Algorithms that are compute-intensive rather than data-intensive.* Computations of the latter type need to communicate large quantities of data between processors and may be affected seriously if the communications properties of the platform are poor.

- *Algorithms with weak synchronicity requirements.* Algorithms that require strict coordination between the different parts of the overall computation may run inefficiently when the platform is highly heterogeneous or has poor latency properties, or if the worker processors are constantly being reclaimed by their owners. In such algorithms, the entire computation can be suspended while waiting for a single slow or suspended worker processor to complete its task.

- *Algorithms in which the size of the computational task can be varied.* In these algorithms, task sizes can be adapted to the capabilities of the machines in the pool and of the communication networks.

- *Algorithms for problems that are large and significant enough to justify the programming and debugging effort.* Although significant advances have been made in tools to facilitate programming for complex applications in this environment (see Section 1.2.3 below), the effort required to implement algorithms on grid computing platforms is certainly not trivial at present.

Although the master-worker model may seem restrictive, we have found that a surprising range of algorithms in optimization can be expressed in terms of this model. For example, branch-and-bound algorithms can quite naturally be implemented as master-worker algorithms by using the master process to store global information and manage the branch-and-bound tree, and worker processes to solve individual nodes or subtrees of relaxed problems.

We are convinced that inexpensive, powerful platforms such as those managed by the Condor system can be put to highly effective use by many optimization researchers and practitioners. In an academic setting, this platform has already been used to solve large instances of the quadratic assignment problem [1], linear integer programming [5], and nonlinear integer programming [14], as well as the stochastic programming problems with recourse discussed in this report. In an industrial setting too, we believe that the use of idle time on office workstations and PCs to solve complex problems associated with scheduling and logistics is an appealing alternative to the purchase and operation of a dedicated computing server.

1.2.2 Stochastic Programming on Grids

The platform we have outlined above is well suited to solving stochastic programming problems of various types. In the next section, we discuss one particular problem—two-stage stochastic linear programming with recourse over a finite set of scenarios. More generally, however, we can note the following relevant properties of stochastic optimization problems.

- Stochastic problems are typically compute-intensive rather than data-intensive. The often huge set of scenarios usually can be expressed in a compact fashion (for example, by listing the possible values of the independent random variables in a problem and their associated probabilities). Problems with integer variables are especially compute-intensive.

- Stochastic programming algorithms often have weak synchronicity requirements. For instance, in sampling-based methods [27, 37], the different samples can be evaluated independently with no synchronization—and example of *embarrassingly parallel* or *pleasantly parallel* computation. In the L-shaped algorithm for stochastic programs with recourse, second-stage problems associated with the different scenarios can be solved independently, though synchronicity enters the algorithm in that the master processor waits for all scenarios to be evaluated before solving a new master problem. (We discuss in a later section how this requirement can be relaxed.)

- Stochastic programming algorithms can easily vary the size of their computational tasks. One can partition by scenarios, defining each task to be a cluster of scenarios, and varying the task size by varying the number of scenarios in the task. The fact that we often can predict the time required to perform each task with some accuracy makes it easier to schedule the application efficiently on a parallel platform. Load balancing is more difficult in most other areas of optimization, for example, in integer programming, where it is difficult to predict the time required to process each subtree of the branch-and-bound tree (see [26]).

- Stochastic programming algorithms can benefit from the computational power offered by a grid platform. For problems with continuous variables, the number of scenarios that we can handle grows roughly in proportion to the computing power available. The ability to solve problems with more scenarios, and to solve multiple sampled instances of the same problem, can significantly improve the quality of the computed solution.

1.2.3 MW: A Tool For Implementing Master-Worker Algorithms on Grids

MW (see Goux, Linderoth, and Yoder [15] and Goux et al. [13]) is a runtime support library that facilitates implementation of master-worker algorithms on computational grids. The application programming interface takes the form of a set of three C++ abstract classes, containing ten methods, that must be reimplemented by the applications programmer to fit their particular application. Users are of course free to define additional methods as needed for their algorithms. In Section 1.3.3, we outline the reimplementation of these classes in the case of our asynchronous algorithms for two-stage stochastic linear programming with recourse.

MWDriver. The MWDriver class is responsible for initializing the algorithm and for performing the actions required by the application whenever one of the worker processors completes its assigned computational task. The base class contains additional functionality whose complexity is hidden from the user, including the handling of worker processes that join and leave the computation, assignment of tasks to appropriate workers, rescheduling of tasks when their host workers disappear without warning, and keeping track of performance data for the run.

MWTask. This class is the abstraction of a single computational task. It holds both the data describing that task and the results obtained by executing the task. The user must reimplement methods for packing and unpacking the task data into simple data structures to be passed between master and workers, using communications primitives appropriate to the particular grid platform. (When using Condor's implementation of PVM, for instance, PVM communication primitives are used.) Similar routines for packing and unpacking the results of the task are also required.

MWWorker. The MWWorker class is the core of the executable that runs on each worker. After initializing itself, using information passed from the master, the worker process sits in a loop, waiting for tasks to be sent to it. When a task is received, an MWWorker method executes the task, and then invokes an MWTask method to pass results back to the master. The worker process then returns to its wait loop.

Besides its interface to the application, MW also has an abstract *infrastructure-programming interface* (IPI) that allows it to be implemented on different grid platforms. To implement MW on their grid, a grid programmer must reimplement a few functions that facilitate communications between processors and management of computational resources. To date, there are four concrete implementations of the MW IPI. Three of these implementations use Condor to manage the computational resource pool, but differ in the manner in which they pass messages between master and workers; they use PVM, shared files, and UNIX sockets, respectively. The fourth implementation runs both master and worker as a single process and is used for debugging.

1.3 Decomposition Methods for Stochastic Programming on Grids

We now discuss algorithms for two-stage stochastic linear programming with recourse and their implementation on a grid platform based on Condor and MW. A more complete discussion of these algorithms, their implementations, and computational results can be found in [24].

Our target problem has the following form:

$$\min_x \ \mathcal{Q}(x) \stackrel{\text{def}}{=} c^T x + \sum_{i=1}^{N} p_i \mathcal{Q}_i(x), \ \text{ subject to } Ax = b, \ x \geq 0, \tag{1.1}$$

where each $\mathcal{Q}_i(x)$ is the value function for a second-stage linear program:

$$\mathcal{Q}_i(x) \stackrel{\text{def}}{=} \min_{y(\omega_i)} \ q(\omega_i)^T y(\omega_i) \ \text{ subject to} \tag{1.2a}$$

$$W y(\omega_i) = h(\omega_i) - T(\omega_i)x, \ \ y(\omega_i) \geq 0. \tag{1.2b}$$

Each ω_i, $i = 1, 2, \ldots, N$ represents a scenario index, possibly obtained by sampling from a large (possibly infinite) set of scenarios. The function $\mathcal{Q}$ is convex and piecewise linear. We assume for purposes of this discussion that $\mathcal{Q}$ is defined for all $x \geq 0$ satisfying $Ax = b$; that is, our problem has relatively complete recourse. The algorithms use subgradient information about $\mathcal{Q}$ to construct a lower-bounding model function which becomes the basis for choosing a succession of iterates x^k, $k = 0, 1, 2, \ldots$ of the first-stage variables. These iterates converge to the solution set for the problem (1.1).

We denote the subdifferential of $\mathcal{Q}$ and its component functions $\mathcal{Q}_i$ by $\partial\mathcal{Q}$ and $\partial\mathcal{Q}_i$, respectively. If $\pi(\omega_i)$ is a vector of Lagrange multipliers for the constraints $Wy(\omega_i) = h(\omega_i) - T(\omega_i)x$ in (1.2), then we have

$$-T(\omega_i)^T \pi(\omega_i) \in \partial\mathcal{Q}_i(x). \tag{1.3}$$

Provided that x lies in the domain of every $\mathcal{Q}_i$, it follows from (1.1) that

$$\partial\mathcal{Q}(x) = c + \sum_{i=1}^{N} p_i \partial\mathcal{Q}_i(x). \tag{1.4}$$

Therefore, we can both evaluate the function $\mathcal{Q}(x)$ and generate subgradient information by finding a primal-dual solution of each second-stage problem (1.2).

1.3.1 Parallel L-Shaped Algorithm

The first algorithm we implemented for solving (1.1) is a multicut version of the L-shaped algorithm of Van Slyke and Wets [39]. It works by partitioning the N scenarios of (1.1) into T clusters $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_T$ and by using function and subgradient information for each partial sum defined by

$$\mathcal{Q}_{[j]}(x) = \sum_{i \in \mathcal{N}_j} p_i \mathcal{Q}_i(x), \quad j = 1, 2, \ldots, T, \tag{1.5}$$

to define a lower-bounding piecewise-linear approximation $m_{[j]}$ to $\mathcal{Q}_{[j]}$. At the kth iteration, we solve the following model problem to obtain a new iterate x^{k+1}:

$$\min_x m_k(x), \quad \text{subject to } Ax = b,\ x \geq 0, \tag{1.6}$$

where

$$m_k(x) \stackrel{\text{def}}{=} c^T x + \sum_{j=1}^{T} m_{[j]}^k(x). \tag{1.7}$$

Here, $m_{[j]}^k$ denotes the model function for cluster j after subgradient information from the first k iterates $x^0, x^1, \ldots, x^k$ has been incorporated. The problem (1.6) can be formulated as a linear program whose optimal value provides a lower bound on the solution of (1.1).

Even if we define a computational task to be the evaluation of scenarios in just one cluster, a maximum of T workers can be utilized at any one time. The master processor stores all the subgradient information needed to define the problem (1.6) and solves the resulting linear program to obtain a new iterate x^{k+1} when evaluation of the function and subgradient for (1.5) has been completed for $x = x^k$.

This basic approach has the disadvantage of being too synchronous for a grid platform. If one of the tasks is delayed or lost, the master process and the entire worker pool are left without useful work to do. Ultimately, Condor and MW will reschedule the task on another worker, but the delay may seriously affect the performance of the algorithm.

Fortunately, it is easy to modify the L-shaped approach to reduce its synchronicity. The master simply waits until some *fraction* $\sigma \in (0, 1]$ of the evaluation tasks at x^k are completed before re-solving the master problem (1.6) (using all the current information in the model) to obtain x^{k+1}. In this asynchronous variant, the algorithm is not seriously delayed by a few tardy tasks. For values of σ nearer zero, the list of tasks waiting to be processed at any given time may include tasks from a number of past iterates, and the number of tasks being processed at any given time may be significantly greater than T. Hence, the number of workers that can be put to work on the problem is potentially greater than in the basic algorithm. However, the algorithm typically requires more iterations to converge to a solution of (1.1) as we decrease σ. This phenomenon is not surprising, since we are evaluating x^{k+1} using less information about the function $\mathcal{Q}$ than is available at the corresponding iteration of the synchronous algorithm ($\sigma = 1$). Hence, there is a tradeoff between total workload and improved asynchronicity / parallelism. In our tests, we tried values of the synchronicity parameter σ between .25 and .9, and found that values of σ in the range $[.75, .9]$ appeared to yield the best wall clock times on a typical grid configuration.

1.3.2 Parallel Trust-Region Algorithm

Both the practical performance of the L-shaped method and its theoretical properties leave something to be desired. The L-shaped method tends to take large steps on early iterations, when the model function m_k is only a crude approximation to $\mathcal{Q}$. Further, although one can devise heuristics for deleting uninteresting subgradient information from the model (to prevent the number of constraints in the equivalent linear program for (1.6) becoming excessively large), there is no theoretically reliable way of doing so.

Regularization by means of a trust region can be used to improve the properties of the method. A trust-region algorithm adds the following basic features to the L-shaped approach.

- In solving for the new candidate iterate x, we add the following trust-region constraint to the problem (1.6),

$$\|x - x^k\|_\infty \leq \Delta^k, \tag{1.8}$$

  where Δ^k is the trust-region radius at iteration k. The resulting problem still can be formulated as a linear program.

- The point obtained by solving (1.6), (1.8) is not accepted as the new iterate unless it yields a "sufficient decrease" in the value of $\mathcal{Q}$ over the current point x^k. If it fails to do so, the trust-region radius may be reduced and a new subproblem of the form (1.6), (1.8) solved. Note that we do not *necessarily* reduce the trust region radius; even without changing the trust region, the additional function and subgradient information obtained at the failed candidate may enhance the model sufficiently to produce a successful step.

- After stepping to a new iterate, the algorithm may delete subgradient information generated earlier in the algorithm. In particular, subgradients that have been inactive for a number of successive iterations may be removed.

The sequence of iterates generated by this trust-region (TR) algorithm can be shown to converge to the solution set $\mathcal{S}$ of (1.1), if $\mathcal{S}$ is nonempty; see [24, Theorem 2]. The approach has the advantage that if started from a point x^0 close to $\mathcal{S}$ (a situation that frequently arises in our tests), the trust region ensures that the initial iterates stay close to this point. In fact, the algorithm does not step away from x^0 at all until the model m_k becomes sufficiently good to generate a point with a significantly better value of $\mathcal{Q}$. The overall algorithm is related to those proposed by Ruszczyński [31], Kiwiel [22], and Hirart-Urruty and Lemaréchal [18, Chapter XV], who use quadratic penalty terms to restrict the distance from x^k to the next candidate iterate.

Parallel implementation of the TR algorithm on the grid is similar to implementation of the synchronous L-shaped method, and the main drawback is the same: A delay in the evaluation of one task can cause the workers and master to be left idle for long periods. However, devising an asynchronous variant of the TR approach that retains the appealing theoretical properties of the synchronous is more difficult than for the L-shaped method. Our asynchronous trust-region (ATR) method maintains an incumbent point x^I and a basket $\mathcal{B}$ containing a set of candidate points for which function and subgradient information is currently being calculated by the workers. When evaluation of one of the points in $\mathcal{B}$ is completed, a "sufficient decrease" test is applied to determine whether this point should replace x^I as the incumbent. Whether this happens or not, the trust region radius may be adjusted, and cuts may be deleted from the model. A trust-region subproblem similar to (1.6), (1.8) is solved to find a new candidate point. The model function used in the trust-region subproblem uses whatever subgradient information is currently available, including information from partially completed evaluations of points in the basket $\mathcal{B}$. The trust region is centered at the incumbent x^I, rather than the latest iterate x^k as in (1.8).

The basket is filled initially by solving the trust-region subproblem after a certain fraction $\sigma \in (0, 1]$ of the evaluation tasks for the first few iterates is completed. However, during most of the execution, the basket is in steady state, with each completed evaluation resulting in a point leaving the basket and being replaced by a new candidate obtained by solving a trust-region subproblem.

Techniques for adjusting the trust-region radius and managing the subgradient information in the model are closely related to those used in the synchronous case. The theoretical convergence results are also similar. Under the assumption that all evaluation tasks complete in finite time, the sequence of incumbents x^I converges to the optimal set $\mathcal{S}$ (see [24, Theorem 5]).

The ATR algorithm clearly has greater potential for parallel implementation than its synchronous cousin TR. A larger number of tasks are available for evaluation at any given time. It also is less synchronous in that a delay in processing one task holds up the evaluation of one point in the basket $\mathcal{B}$, but not the entire algorithm. Not unless nearly all points in $\mathcal{B}$ are blocked in this way do we expect serious degradation in parallel performance. On the other hand, as in the asynchronous L-shaped approach, ATR typically performs more total computation than the synchronous TR algorithm; the total number of iterates (that is, the number of times the trust-region subproblem is solved) grows with basket size. Once again, there is a tradeoff between total work on one hand, and improved parallelism and asynchronicity on the other.

1.3.3 MW Implementations

In this section, we elaborate on our discussion of the MW runtime support library in Section 1.2.3. We describe how each of the key methods in MW is implemented in the case of the ATR application of Section 1.3.2.

MWDriver.

- `get_userinfo`: This function, which is executed at the start of ATR, reads a command file to set parameters such as convergence tolerances, number of scenarios, number of partial sums to be evaluated in each task, maximum number of worker processors to be requested, and initial trust region radius. It calls the routines that read and store the problem data files, and reads the initial point, if one is supplied by the user. This routine also sets important algorithmic parameters. In particular, it decides how many of the clusters defined in (1.5) to place in one computational task, to be assigned subsequently to a worker. Each task is chosen large enough to ensure that it requires at least five seconds to execute on the fastest processor in the pool. (In our largest examples, considerably larger tasks were used.)

- `setup_initial_tasks`: Populates the task pool with the tasks for evaluating the objective at the initial point x^0.

- `pack_worker_init_data`: Sends the information in the input files for the stochastic programming problem to each new new worker as it joins the pool. When the worker subsequently receives a task, it uses the original input data to generate the particular second-stage data for its assigned set of scenarios.

- `act_on_completed_task`: When a worker completes its task, this routine adds the partial sums $\mathcal{Q}_{[j]}$ evaluated by this task to the objective function and places its subgradient information in a buffer. When this task is the one that completes the evaluation of the objective $\mathcal{Q}(x)$ at some point x, the method decides whether to accept x as the new incumbent, performs any necessary adjustments to the trust region radius, adds cuts from the current buffer into the model, possibly deletes obsolete cuts, solves the trust-region subproblem to generate a new candidate point, and adds the evaluation tasks for the new candidate to the list of tasks.

MWTask. In our applications, each task evaluates the partial sum $\mathcal{Q}_{[j]}(x)$ and a subgradient for a certain number of clusters. The task is described compactly by a range of scenario indices for each of its clusters and by the value of the first-stage variables x. The results consist of the value of $\mathcal{Q}_{[j]}$ and its subgradient for each clusters making up the task. The functions in this class simply pack and unpack the data and results into data structures suitable for transmission between master and worker.

MWWorker. The `execute_task` method of this class formulates the second-stage linear programs in the cluster for the given task, using the task definition information received from the master. It then calls the linear programming solver SOPLEX to solve these problem, and uses their dual solutions to calculate the subgradients.

1.3.4 Computational Results

We now present a selection of computational results obtained with the solvers described above. For a fuller picture, in particular the dependence of the results on various parameter choices, we refer the reader to [24]. For extensive sampling studies performed with these solvers, see [23].

run	points evaluated	size of $\mathcal{B}$	# tasks	# clusters	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	master problem solve time (min)	wall clock time (min)
ALS	282	-	50	50	100	26	.81	5562	24	254
TR	46	-	25	100	25	22	.43	4032	1	56
TR	43	-	50	100	50	33	.48	3922	1	35
ATR	96	3	25	100	50	37	.55	6682	2	47
ATR	100	3	50	100	100	41	.64	8083	2	30
ATR	171	6	25	100	87	56	.80	7653	5	37
ATR	170	6	50	100	175	44	.90	9143	5	36

Table 1.1. *SSN trial with best parameter combinations, $N = 10000$ scenarios, Algorithms ALS, TR, and ATR*

Our first test problem is known as **SSN**, a telecommunications design application described in [35]. This problem has 89 first-stage variables and a single constraint (a budget constraint), while the second stage problems have 706 unknowns, 175 constraints, and a constraint matrix with 2284 nonzeros. This problem contains 86 independent random variables and a total of 10^{70} scenarios. It has a reputation of being difficult, in that algorithms typically require many iterations and, even when variance reduction techniques are applied, good quality upper and lower bounds on the optimal objective value are hard to calculate (see [17, 27]). Computational experience with this problem has been reported with the serial Regularized Decomposition (RD) code of Ruszczyński and Świetanowski [33]. In [27], 30 sampled instances with 1000 scenarios each were solved using RD on a (what machine?) in an average of 81 minutes per instance. In 1996, the authors of the RD code reported solution times of 64 minutes for an instance of size 500 and 163 minutes for an instance of size 1000, and on a single processor of a Cray 6400 SuperServer, which is roughly equivalent in speed to a SparcStation 5. More recently, we obtained an execution time for RD of 220 minutes on a Sun UltraSparc-2 with 128MB of memory for an instance of size 5000. (This version of RD has an upper limit of 5000 on the number of scenarios.)

Table 1.1 shows results obtained from a sampled instance of SSN with $N = 10000$ scenarios, which can be formulated as a linear program with approximately 1.75×10^6 constraints and 7.06×10^6 variables. The table shows results from the asynchronous version of the L-shaped method (ALS), the trust-region method (TR) and the asynchronous trust-region method (ATR) with various choices of basket size ($|\mathcal{B}|$), and varying numbers of tasks and clusters. The second column tabulates the number of points at which the function $\mathcal{Q}$ was evaluated, and the last column shows the total wall clock time for the jobs. The number of workers requested from the pool, the average number of workers used during the computation, and the parallel efficiency (the proportion of time for which the owned workers were kept busy) are also shown. The final columns show the maximum number of cuts (subgradients) stored in the master-problem model during the computation, and the total time spent solving the problem (1.6), (1.8) on the master processor.

Number	Type	Location
184	Intel/Linux	Argonne
254	Intel/Linux	New Mexico
36	Intel/Linux	NCSA
265	Intel/Linux	Wisconsin
88	Intel/Solaris	Wisconsin
239	Sun/Solaris	Wisconsin
124	Intel/Linux	Georgia Tech
90	Intel/Solaris	Georgia Tech
13	Sun/Solaris	Georgia Tech
9	Intel/Linux	Columbia U.
10	Sun/Solaris	Columbia U.
33	Intel/Linux	Italy (INFN)
1345		**TOTAL**

Table 1.2. *Machines Available for Storm Problem, with* $N = 10^7$ *Scenarios.*

The table shows that the ALS code is generally slower than the trust region variants, which all perform fairly well for the choices of parameters shown here. In general, the performance depends somewhat on conditions on the Condor pool at the time the job is run. The priority assigned to the job affects the number of workers it can obtain, and the overall run time can be seriously affected if workers are suspended repeatedly. Poor performance of the synchronous algorithm TR has been observed on some runs, when two or three workers are significantly slower than the others in the pool.

run	points evaluated	size of $\|\mathcal{B}\|$	# tasks	# clusters	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	master problem solve time (hr)	wall clock time (hr)
ATR	38	4	1024	1024	800	433	.668	39647	1.9	31.9

Table 1.3. *Storm Problem, with $N = 10^7$ Scenarios.*

To demonstrate the scale of problem that can be solved on a computational grid platform, we solved large instances of a cargo flight scheduling problem known as **storm**. This problem was described by Mulvey and Ruszczyński [28], who also reported considerable computational testing with the algorithm they proposed in their paper, along with several other codes and an earlier implementation of Ruszczyński's regularized decomposition algorithm [31]. In experiments performed in 1992-93, the latter was found to perform best; a 1000-scenario instance was solved in about 6 hours of CPU time on a SUN SPARCstation 2. (The other codes were not able to handle more than 200 scenarios in a reasonable amount of time.) Recently, we tested the current implementation of RD on a Sun UltraSparc-2 and solved a 1000-scenario instance in about 20 minutes (45 iterations) and a 5000-scenario instance in about 260 minutes (33 iterations).

In [24], we report solving a sampled instance of storm with $N = 250000$ in 116 minutes using ATR. The average size of the worker pool during the run (drawn from the Condor pool at Wisconsin) was 106. In our largest run, we solved an instance of storm with $N = 10^7$ scenarios, for which the equivalent linear program has approximately 10^9 constraints and 10^{10} variables. A starting point was calculated from a smaller sampled instance. For this run, we used ATR with a basket size of 4, and executed on a pool of more than 1300 workers from seven different institutions running three different operating systems (see Table 1.2). The job ran for a total of almost 32 hours; the number of machines in use over the course of the run is graphed in Figure 1.1. The master process crashed after 8 hours of execution, but Condor's checkpointing mechanism allowed it to be restarted 2 hours later and then run an additional 24 hours to completion.

The run is profiled in Table 1.3. As noted there, an average of 433 workers were present during the run, and 556 workers were used altogether. A total of 40837 tasks were generated during the run, representing 3.99×10^8 second-stage linear programs. (At this rate, an average of 3472 second-stage linear programs were being solved per second during the run.) The average time to solve a task was 774 seconds. The total cumulative CPU time spent by the worker pool was 9014 hours, or slightly more than one year.

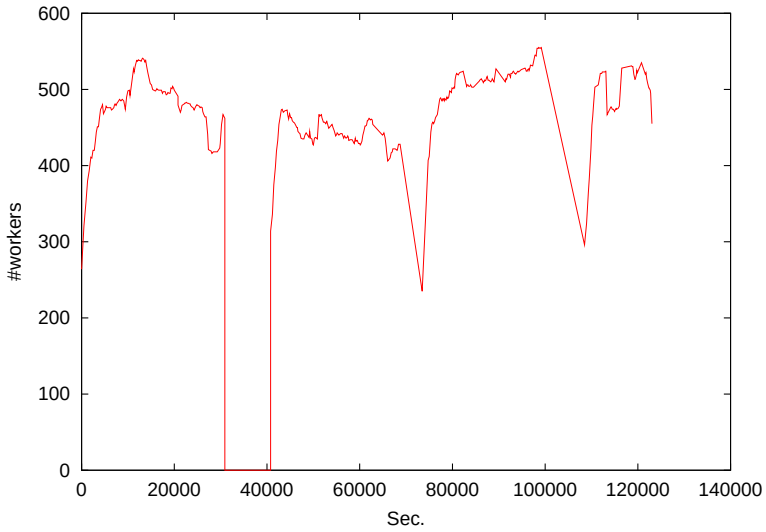

Figure 1.1. *Machines in Use during Execution of Storm Problem with* $N = 10^7$ *Scenarios.*

1.4 Future Directions

Our goal in this paper has been to introduce the stochastic programming community to a platform that seems well-suited to the needs of algorithms for solving large practical instances of problems in the area. In this section we outline some future plans for algorithms and software for other stochastic problems, and relevant plans for the underlying grid environment.

We plan to improve the ATR code further by improving its algorithmic features, possibly by using a Euclidean-norm trust region (which necessitates solving a quadratic program at each iteration) and by improving the management of the subgradient buffer. Other improvements will come from the use of performance modeling techniques to better understand the behavior of the ATR code in dynamic, heterogeneous computational environments. At present, our understanding of the effect on performance of different choices of the algorithmic parameters (such as basket size and numbers of chunks and tasks) is limited, especially when the pool contains machines of very different speeds. We are working with experts in performance modeling (see [30]) to understand better these issues. Ultimately, we hope to use performance models within the ATR code, enabling it to adapt its behavior to changing pool conditions during a run. Adaptations that may be made include changing the basket size; changing the number of tasks or clusters; removing an unpromising point from the basket before its evaluations are complete; or assigning different amounts of work to different workers.

Approaches that require solution of a number of independent sample-average approximations are ideal for implementation on a computational grid. The jobs can be run independently, possibly from different masters to avoid interference with each other. Batch approaches such as this are useful in computing estimates of the upper and lower bounds on the objective value. We refer the reader to [27] for verification experiments that rely on this kind of computation. The recent report [23] describes a large set of empirical experiments carried out with ATR.

Many of our planned adaptations to the algorithms will require enhancements to the capabilities of MW. Future versions of MW will include improved performance information that can be exploited by the performance modeling techniques described above, such as better measurement of task execution times, communication times, times required to obtain new workers; and worker suspension rates. Improved logging tools will enable us to improve performance and eliminate sources of inefficiency in the algorithms. Other planned features for MW will allow the application code greater control over its worker pool; including the ability to remove a worker, to interrupt or terminate execution of tasks on specific workers, and to dynamically change the number of workers requested during the computation. These enhancements to MW capabilities will not benefit our stochastic programming application alone, but will also be useful in parallel algorithms for other compute-intensive numerical applications.

Longer-term goals involve stochastic integer programming. The complexity of parallel algorithms for this problem and their high computational demands will require much larger worker pools and new ways to manage these pools. We anticipate extending MW to support a hierarchical master-worker framework in which a second layer of master processors will direct computations on subtrees of the branching tree. Stochastic integer programming is recognized as a difficult class of problems (see for example [4]) and relatively little research on algorithms has been performed to date, especially not on general-purpose algorithms (see the bibliography [38]). We believe that computational grids of the type discussed in this paper open the door to solution of interesting problems in this area, and we hope that this possibility will spur new work on algorithm development and implementation.

Acknowledgments

We thank Alex Shapiro for interesting discussions and advice over the course of this project. We also thank other members of the metaNEOS, Condor, and POEMS teams for their ongoing work on the grid platform described here and for their continuing inspiration and advice.

Bibliography

[1] K. Anstreicher, N. Brixius, J.-P. Goux, and J. T. Linderoth. Solving large quadratic assignment problems on computational grids. To appear in Mathematical Programming, available at `http://www.optimization-online.org/`, 2001.

[2] K. A. Ariyawansa and D. D. Hudson. Performance of a benchmark parallel implementation of the Van Slyke and Wets algorithm for two-stage stochastic programs on the Sequent/Balance. *Concurrency Practice and Experience*, 3:109–128, 1991.

[3] J. R. Birge, C. J. Donohue, D. F. Holmes, and O. G. Svintsiski. A parallel implementation of the nested decomposition algorithm for multistage stochastic linear programs. *Mathematical Programming*, 75:327–352, 1996.

[4] J. R. Birge and F. L. Louveaux. *Introduction to Stochastic Programming.* Springer, 1997.

[5] Q. Chen, M. C. Ferris, and J. T. Linderoth. Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research*, 103:17–32, 2001.

[6] Condor Project: `http://www.cs.wisc.edu/condor`.

[7] G. Dantzig, J. Ho, and G. Infanger. Solving stochastic linear programs on a hypercube multicomputer. Technical Report SOL 91-10, Department of Operations Research, Stanford University, August 1991.

[8] Entropia, 2001. `http://www.entropia.com/`.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[10] E. Fragnière, J. Gondzio, and J.-P. Vial. Building and solving large-scale stochastic programs on an affordable distributed computing system. *Annals of Operations Research*, 99:167–187, 2000.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, MA, 1994.

[12] J. Gondzio and R. Kouwenberg. High performance computing for asset liability management. Technical Report MS-99-004, Department of Mathematics and Statistics, The University of Edinburgh, May 1999.

[13] J.-P. Goux, S. Kulkarni, J. T. Linderoth, and M. Yoder. Master-Worker : An enabling framework for master-worker applications on the computational grid. *Cluster Computing*, 4:63–70, 2001.

[14] J.-P. Goux and S. Leyffer. Solving large MINLPs on computational grids. Numerical Analysis Report NA/200, Mathematics Department, University of Dundee, 2001.

[15] J.-P. Goux, J. T. Linderoth, and M. Yoder. Metacomputing and the master-worker paradigm. Preprint MCS/ANL-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., February 2000.

[16] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[17] J. Higle. Variance reduction and objective function evaluation in stochastic linear programs. *INFORMS Journal on Computing*, 10:236–247, 1998.

[18] J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms II*. Comprehensive Studies in Mathematics. Springer-Verlag, 1993.

[19] E. Jessup, D. Yang, and S. Zenios. Parallel factorization of structured matrices arising in stochastic programming. *SIAM Journal on Optimization*, 4:833–846, 1994.

[20] A. J. King. SP/OSL V1.0, Stochastic Programming Interface Library User's Guide. Technical report, IBM Research Division, Mathematical Sciences Department IBM T. J. Watson Research Center, 1994.

[21] A. J. King and S. E. Wright, A Flexible-partition, nested L-shaped method for linear programming, Technical report, IBM Research Division, Mathematical Sciences Department IBM T. J. Watson Research Center, 1999.

[22] K. Kiwiel. Proximity control in bundle methods for convex nondifferentiable minimization. *Mathematical Programming*, 46:105–122, 1990.

[23] J. T. Linderoth, A. Shapiro, and S. J. Wright. The Empirical behavior of sampling methods for stochastic programming. Optimization Technical Report 02-01, Computer Sciences Department, University of Wisconsin-Madison, January, 2002. Available at `http://www.optimization-online.org/`.

[24] J. T. Linderoth and S. J. Wright. Decomposition algorithms for stochastic programming on a computational grid. Preprint ANL/MCS-P875-0401, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., April 2001. Available at `http://www.optimization-online.org/`.

[25] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11, 1997. Available from `http://www.cs.wisc.edu/condor/doc/htc_mech.ps`.

[26] R. Lüling and B. Monien. Load balancing for distributed branch and bound algorithms. In *Proceedings of the International Parallel Processing Symposium*, pages 543–549, Beverly Hills, CA, 1992.

[27] W. K. Mak, D. P. Morton, and R. K. Wood. Monte Carlo bounding techniques for determining solution quality in stochastic programs. *Operations Research Letters*, 24:47–56, 1999.

[28] J. M. Mulvey and A. Ruszczyński. A new scenario decomposition method for large scale stochastic optimization. *Operations Research*, 43:477–490, 1995.

[29] S. S. Nielsen and S. A. Zenios. Scalable parallel Benders decomposition for stochastic linear programming. *Parallel Computing*, 23:1069–1089, 1997.

[30] POEMS Project. `http://www.cs.utexas.edu/users/poems/`.

[31] A. Ruszczyński. A regularized decompoisition method for minimizing a sum of polyhedral functions. *Mathematical Programming*, 35:309–333, 1986.

[32] A. Ruszczyński. Parallel decomposition of multistage stochastic programming problems. *Mathematical Programming*, 58:201–228, 1993.

[33] A. Ruszczyński and A. Świetanowski. On the regularized decomposition method for two-stage stochastic linear problems. Working paper WP-96-014, IIASA, Laxenburg, Austria, 1996.

[34] A. Ruszczyński and A. Świetanowski. Table at `http://users.iems.nwu.edu/~jrbirge//html/dholmes/POSTresults.html`

[35] S. Sen, R. D. Doverspike, and S. Cosares. Network planning with random demand. *Telecommunications Systems*, 3:11–30, 1994.

[36] `SETI@home`, 2001. `http://setiathome.ssl.berkeley.edu/`.

[37] A. Shapiro. Stochastic programming by Monte Carlo simulation methods. Technical report, School of ISYE, Georgia Institute of Technology, 1999.

[38] M. H. van der Vlerk. Stochastic Integer Programming Bibliography, 1996-2001. `http://mally.eco.rug.nl/biblio/SIP.HTML`

[39] R. Van Slyke and R. J-B. Wets. L-shaped linear programs with applications to control and stochastic programming. *SIAM Journal on Applied Mathematics*, 17:638–663, 1969.