Preprint ANL/MCS-P875–0401, April, 2001
Mathematics and Computer Science Division
Argonne National Laboratory

Jeff Linderoth · Stephen Wright

Decomposition Algorithms for Stochastic Programming on a Computational Grid

April 17, 2001

Abstract. We describe algorithms for two-stage stochastic linear programming with recourse and their implementation on a grid computing platform. In particular, we examine serial and asynchronous versions of the L-shaped method and a trust-region method. The parallel platform of choice is the dynamic, heterogeneous, opportunistic platform provided by the Condor system. The algorithms are of master-worker type (with the workers being used to solve second-stage problems), and the MW runtime support library (which supports master-worker computations) is key to the implementation. Computational results are presented on large sample average approximations of problems from the literature.

1. Introduction

Consider the following stochastic optimization problem:

$$\min_{x \in S} F(x) \stackrel{\text{def}}{=} \sum_{i=1}^{N} p_i f(x, \omega_i), \tag{1}$$

where $S \in \mathsf{R}^n$ is a constraint set, $\Omega = \{\omega_1, \omega_2, \ldots, \omega_N\}$ is the set of outcomes (consisting of N distinct scenarios), and p_i is the probability associated with each scenario. Problems of the form (1) can arise directly (in many applications, the number of scenarios is naturally finite), or as discretizations of problems over continuous probability spaces, obtained by approximation or sampling. In this paper, we discuss the *two-stage stochastic linear programming problem with fixed resource*, which is a special case of (1) defined as follows:

$$\min\ c^T x + \textstyle\sum_{i=1}^{N} p_i q(\omega_i)^T y(\omega_i), \quad \text{subject to} \tag{2a}$$

$$Ax = b, \quad x \geq 0, \tag{2b}$$

$$Wy(\omega_i) = h(\omega_i) - T(\omega_i)x, \quad y(\omega_i) \geq 0, \quad i = 1, 2, \ldots, N. \tag{2c}$$

The unknowns in this formulation are x and $y(\omega_1), y(\omega_2), \ldots, y(\omega_N)$, where x contains the "first-stage variables" and each $y(\omega_i)$ contains the "second-stage

Jeff Linderoth: Axioma Inc., 501-F Johnson Ferry Road, Suite 450, Marietta, GA 30068; `jlinderoth@axiomainc.com`

Stephen Wright: Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439; `wright@mcs.anl.gov`

Mathematics Subject Classification (1991): 90C15, 65K05, 68W10

variables" associated with the ith scenario. The ith scenario is characterized by the probability p_i and the data objects $(q(\omega_i), T(\omega_i), h(\omega_i))$.

The formulation (2) is sometimes known as the "deterministic equivalent" because it lists the unknowns for all scenarios explicitly and poses the problem as a (potentially very large) structured linear program. An alternative formulation is obtained by recognizing that each term in the second-stage summation in (2a) is a piecewise linear convex function of x. Defining the ith second-stage problem as a linear program (LP) parametrized by the first-stage variables x, that is,

$$\mathcal{Q}_i(x) \stackrel{\text{def}}{=} \min_{y(\omega_i)} \; q(\omega_i)^T y(\omega_i) \;\text{ subject to} \tag{3a}$$

$$W y(\omega_i) = h(\omega_i) - T(\omega_i)x, \;\; y(\omega_i) \geq 0, \tag{3b}$$

and defining the objective in (2a) as

$$\mathcal{Q}(x) \stackrel{\text{def}}{=} c^T x + \sum_{i=1}^{N} p_i \mathcal{Q}_i(x), \tag{4}$$

we can restate (2) as

$$\min_x \; \mathcal{Q}(x), \;\; \text{subject to } Ax = b, \; x \geq 0. \tag{5}$$

We note several features about the problem (5). First, it is clear from (4) and (3) that $\mathcal{Q}(x)$ can be evaluated for a given x by solving the N linear programs (3) separately. Second, we can derive subgradient information for $\mathcal{Q}_i(x)$ by considering dual solutions of (3). If we fix $x = \hat{x}$ in (3), the primal solution $y(\omega_i)$ and dual solution $\pi(\omega_i)$ satisfy the following optimality conditions:

$$\begin{aligned} q(\omega_i) - W^T \pi(\omega_i) \geq 0 \perp y(\omega_i) \geq 0, \\ W y(\omega_i) = h(\omega_i) - T(\omega_i)\hat{x}. \end{aligned}$$

From these two conditions we obtain that

$$\mathcal{Q}_i(\hat{x}) = q(\omega_i)^T y(\omega_i) = \pi(\omega_i)^T W y(\omega_i) = \pi(\omega_i)^T [h(\omega_i) - T(\omega_i)\hat{x}]. \tag{6}$$

Moreover, since $\mathcal{Q}_i$ is piecewise linear and convex, we have for any x that

$$\mathcal{Q}_i(x) - \mathcal{Q}_i(\hat{x}) \geq \pi(\omega_i)^T [-T(\omega_i)x + T(\omega_i)\hat{x}] = \left(-T(\omega_i)^T \pi(\omega_i)\right)^T (x - \hat{x}), \tag{7}$$

which implies that

$$-T(\omega_i)^T \pi(\omega_i) \in \partial \mathcal{Q}_i(\hat{x}), \tag{8}$$

where $\partial \mathcal{Q}_i(\hat{x})$ denotes the subgradient of $\mathcal{Q}_i$ at $\hat{x}$. By Rockafellar [20, Theorem 23.8], using polyhedrality of each $\mathcal{Q}_i$, we have from (4) that

$$\partial \mathcal{Q}(\hat{x}) = c + \sum_{i=1}^{N} p_i \partial \mathcal{Q}_i(\hat{x}), \tag{9}$$

for every $\hat{x}$ that lies in the domain of each $\mathcal{Q}_i$, $i = 1, 2, \ldots, N$.

Let $\mathcal{S}$ denote the solution set for (5); we assume for most of the paper that $\mathcal{S}$ is nonempty. Since (5) is a convex program, $\mathcal{S}$ is closed and convex, and the projection operator $P(\cdot)$ onto $\mathcal{S}$ is well defined. Because the objective function in (5) is piecewise linear and the constraints are linear, the problem has a *weak sharp minimum* (Burke and Ferris [7]); that is, there exists $\hat{\epsilon} > 0$ such that

$$\mathcal{Q}(x) - \mathcal{Q}^* \geq \hat{\epsilon}\|x - P(x)\|_\infty, \quad \text{for all } x \text{ with } Ax = b,\ x \geq 0, \tag{10}$$

where $\mathcal{Q}^*$ is the optimal value of the objective.

The subgradient information can be used by algorithms in different ways. Successive estimates of the optimal x can be obtained by minimizing over a convex underestimate of $\mathcal{Q}(x)$ constructed from subgradients obtained at earlier iterations, as in the L-shaped method described in Section 2. This method can be stabilized by the use of a quadratic regularization term (Ruszczyński [21], Kiwiel [16]) or by the explicit use of a trust region, as in the ℓ_∞ trust-region approach described in Section 3. Alternatively, when an upper bound on the optimal value $\mathcal{Q}^*$ is available, one can derive each new iterate from an approximate analytic center of an approximate epigraph. The latter approach has been explored by Bahn et al. [1] and applied to a large stochastic programming problem by Frangière, Gondzio, and Vial [8].

Because evaluation of $\mathcal{Q}_i(x)$ and elements of its subdifferential can be carried out independently for each $i = 1, 2, \ldots, N$, and because such evaluations usually constitute the bulk of the computational workload, implementation on parallel computers is possible. We can partition second-stage scenarios $i = 1, 2, \ldots, N$ into "chunks" and define a computational task to be the solution of all the LPs (3) in a single chunk. Each such task could be assigned to an available worker processor. Relationships between the solutions of (3) for different scenarios can be exploited within each chunk (see Birge and Louveaux [5, Section 5.4]). The number of second-stage LPs in each chunk should be chosen to ensure that the computation does not become communication bound. That is, each chunk should be large enough that its processing time significantly exceeds the time required to send the data to the worker processor and to return the results.

In this paper, we describe implementations of decomposition algorithms for stochastic programming on a dynamic, heterogeneous computational grid made up of workstations, PCs (from clusters), and supercomputer nodes. Specifically, we use the environment provided by the Condor system [17]. We also discuss the MW runtime library (Goux et al. [13,12]), a software layer that significantly simplifies the process of implementing parallel algorithms in Condor.

For the dimensions of problems and parallel platforms considered in this paper, evaluation of the functions $\mathcal{Q}_i(x)$ and their subgradients at a single x often is insufficient to make effective use of the available processors. Moreover, "synchronous" algorithms—those that depend for efficiency on all tasks completing in a timely fashion—run the risk of poor performance in an environment such as ours, in which failure or suspension of worker processors while they are processing a task is not an infrequent event. We are led therefore to "asynchronous" approaches that consider different points x simultaneously. Asynchronous variants of the L-shaped and ℓ_∞ trust-region methods are described in Sections 2.2 and 4, respectively.

Other parallel algorithms for stochastic programming have been devised by Birge et al. [4], Birge and Qi [6], and Frangière, Gondzio, and Vial [8]. In [4], the focus is on multistage problems in which the scenario tree is decomposed into subtrees, which are processed independently and in parallel on worker processors. Dual solutions from each subtree are used to construct a model of the first-stage objective (using an L-shaped approach like that described in Section 2), which is periodically solved by a master process to obtain a new candidate first-stage solution x. Parallelization of the linear algebra operations in interior-point algorithms is considered in [6], but this approach involves significant data movement and does not scale particularly well. In [8], the second-stage problems (3) are solved concurrently and inexactly by using an interior-point code. The master process maintains an upper bound on the optimal objective, and this bound along with the subgradients obtained from the second-stage problems yields a polygon whose (approximate) analytic center is calculated periodically to obtain a new candidate x. The approach is based in part on an algorithm described by Gondzio and Vial [11]. The numerical results in [8] report solution of a two-stage stochastic linear program with 2.6 million variables and 1.2 million constraints in three hours on a cluster of 10 Linux PCs.

2. L-Shaped Methods

We now describe the L-shaped method, a fundamental algorithm for solving (5), and an asynchronous variant.

2.1. The Multicut L-Shaped Method

The L-shaped method of Van Slyke and Wets [25] for solving (5) proceeds by finding subgradients of partial sums of the terms that make up $\mathcal{Q}$ (4), together with linear inequalities that define the domain of $\mathcal{Q}$. The method is essentially Benders decomposition [2], enhanced to deal with infeasible iterates. A full description is given in Chapter 5 of Birge and Louveaux [5]. We sketch the approach here and show how it can be implemented in an asynchronous fashion.

We suppose that the second-stage scenarios indexed by $1, 2, \ldots, N$ are partitioned into T clusters denoted by $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_T$. Let $\mathcal{Q}_{[j]}$ represent the partial sum from (4) corresponding to the cluster $\mathcal{N}_j$:

$$\mathcal{Q}_{[j]}(x) = \sum_{i \in \mathcal{N}_j} p_i \mathcal{Q}_i(x). \tag{11}$$

The algorithm maintains a model function $m^k_{[j]}$, which is a piecewise linear lower bound on $\mathcal{Q}_{[j]}$ for each j. We define this function at iteration k by

$$m^k_{[j]}(x) = \inf\{\theta_j \mid \theta_j e \geq F^k_{[j]} x + f^k_{[j]}\}, \tag{12}$$

where $F^k_{[j]}$ is a matrix whose rows are subgradients of $\mathcal{Q}_{[j]}$ at previous iterates of the algorithm, and $e = (1, 1, \ldots, 1)^T$. The rows of $\theta_j e \geq F^k_{[j]} x + f^k_{[j]}$ are referred to as *optimality cuts*. Upon evaluating $\mathcal{Q}_{[j]}$ at the new iterate x^k by solving (3) for each $i \in \mathcal{N}_j$, a subgradient $g_j \in \partial \mathcal{Q}_{[j]}$ can be obtained from a formula derived from (8) and (9), namely,

$$g_j = - \sum_{i \in \mathcal{N}_j} p_i T(\omega_i)^T \pi(\omega_i), \tag{13}$$

where each $\pi(\omega_i)$ is an optimal dual solution of (3). Since by the subgradient property we have

$$\mathcal{Q}_{[j]}(x) \geq g_j^T x + (\mathcal{Q}_{[j]}(x^k) - g_j^T x^k),$$

we can obtain $F^{k+1}_{[j]}$ from $F^k_{[j]}$ by appending the row g_j^T, and $f^{k+1}_{[j]}$ from $f^k_{[j]}$ by appending the element $(\mathcal{Q}_{[j]}(x^k) - g_j^T x^k)$. In order to keep the number of cuts reasonable, the cut is not added if $m^k_{[j]}$ is not greater than the value predicted by the lower bounding approximation (see (17) below). In this case, the current set of cuts in $F^k_{[j]}$, $f^k_{[j]}$ adequately models $\mathcal{Q}_{[j]}$. In addition, we may also wish to delete some rows from $F^{k+1}_{[j]}$, $f^{k+1}_{[j]}$ corresponding to facets of the epigraph of (12) that we do not expect to be active in later iterations.

The algorithm also maintains a collection of *feasibility cuts* of the form

$$D^k x \geq d^k, \tag{14}$$

which have the effect of excluding values of x that were found to be infeasible, in the sense that some of the second-stage linear programs (3) are infeasible for these values of x. By Farkas's theorem (see Mangasarian [18, p. 31]), if the constraints (3b) are infeasible, there exists $\pi(\omega_i)$ with the following properties:

$$W^T \pi(\omega_i) \leq 0, \quad [h(\omega_i) - T(\omega_i)x]^T \pi(\omega_i) > 0.$$

(In fact, such a $\pi(\omega_i)$ can be obtained from the dual simplex method for the feasibility problem (3b).) To exclude this x from further consideration, we simply add the inequality $[h(\omega_i) - T(\omega_i)x]^T \pi(\omega_i) \leq 0$ to the constraint set, by appending the row vector $\pi(\omega_i)^T T(\omega_i)$ to D^k and the element $\pi(\omega_i)^T h(\omega_i)$ to d^k in (14).

The iterate x^k of the multicut L-shaped method is obtained by solving the following approximation to (5):

$$\min_x \; m_k(x), \;\text{ subject to } D^k x \geq d^k, \; Ax = b, \; x \geq 0, \tag{15}$$

where

$$m_k(x) \stackrel{\text{def}}{=} c^T x + \sum_{j=1}^{T} m^k_{[j]}(x). \tag{16}$$

In practice, we substitute from (12) to obtain the following linear program:

$$\min_{x,\theta_1,\dots,\theta_T} \; c^T x + \sum_{j=1}^{T} \theta_j, \quad \text{subject to} \tag{17a}$$

$$\theta_j e \geq F^k_{[j]} x + f^k_{[j]}, \quad j = 1, 2, \dots, T, \tag{17b}$$

$$D^k x \geq d^k, \tag{17c}$$

$$Ax = b, \;\; x \geq 0. \tag{17d}$$

The L-shaped method proceeds by solving (17) to generate a new candidate x, then evaluating the partial sums (11) and adding optimality and feasibility cuts as described above. The process is repeated, terminating when the improvement in objective promised by the subproblem (15) becomes small.

For simplicity we make the following assumption for the remainder of the paper.

Assumption 1.

(i) The problem has complete recourse; that is, the feasible set of (3) is nonempty for all $i = 1, 2, \dots, N$ and all x, so that the domain of $\mathcal{Q}(x)$ in (4) is R^n.
(ii) The solution set $\mathcal{S}$ is nonempty.

Under this assumption, feasibility cuts of the form (14), (17c) do not appear during the course of the algorithm. Our algorithms and their analysis can be generalized to handle situations in which Assumption 1 does not hold, but since our development is complex enough already, we postpone discussion of these generalizations to a future report.

Using Assumption 1, we can specify the L-shaped algorithm formally as follows:

Algorithm LS
choose tolerance ϵ_{tol};
choose starting point x^0;
define initial model m_0 to be a piecewise linear underestimate of $\mathcal{Q}(x)$
  such that $m_0(x^0) = \mathcal{Q}(x^0)$ and m_0 is bounded below;
$\mathcal{Q}_{\min} \leftarrow \mathcal{Q}(x^0)$;
for $k = 0, 1, 2, \dots$
  obtain x^{k+1} by solving (15);
  if $\mathcal{Q}_{\min} - m_k(x^{k+1}) \leq \epsilon_{\text{tol}}(1 + |\mathcal{Q}_{\min}|)$
    STOP;
  evaluate function and subgradient information at x^{k+1};
  $\mathcal{Q}_{\min} \leftarrow \min(\mathcal{Q}_{\min}, \mathcal{Q}(x^{k+1}))$;
  obtain m_{k+1} by adding optimality cuts to m_k;
end(for).

2.2. An Asynchronous Parallel Variant of the L-Shaped Method

The L-shaped approach lends itself naturally to implementation in a master-worker framework. The problem (17) is solved by the master process, while solution of each cluster $\mathcal{N}_j$ of second-stage problems, and generation of the associated cuts, can be carried out by the worker processes running in parallel. This approach can be adapted for an asynchronous, unreliable environment in which the results from some second-stage clusters are not returned in a timely fashion. Rather than having all the worker processors sit idle while waiting for the tardy results, we can proceed without them, re-solving the master by using the additional cuts that were generated by the other second-stage clusters.

We denote the model function simply by m for the asynchronous algorithm, rather than appending a subscript. Whenever the time comes to generate a new iterate, the current model is used. In practice, we would expect the algorithm to give different results each time it is executed, because of the unpredictable speed and order in which the functions are evaluated and subgradients generated. Because of Assumption 1, we can write the subproblem

$$\min_x \; m(x), \quad \text{subject to } Ax = b, \; x \geq 0. \tag{18}$$

Algorithm ALS, the asynchronous variant of the L-shaped method that we describe here, is made up of four key operations, three of which execute on the master processor and one of which runs on the workers. These operations are as follows:

- `partial_evaluate`. This is the routine for evaluating $\mathcal{Q}_{[j]}(x)$ defined by (11) for a given x and j, in the process generating a subgradient g_j of $\mathcal{Q}_{[j]}(x)$. It runs on a worker processor and returns its results to the master by activating the routine `act_on_completed_task` on the master processor.
- `evaluate`. This routine, which runs on the master, simply places T tasks of the type `partial_evaluate` for a given x into the task pool for distribution to the worker processors as they become available. The completion of these T tasks is equivalent to evaluating $\mathcal{Q}(x)$.
- `initialize`. This routine runs on the master processor and performs initial bookkeeping, culminating in a call to `evaluate` for the initial point x^0.
- `act_on_completed_task`. This routine, which runs on the master, is activated whenever the results become available from a `partial_evaluate` task. It updates the model and increments a counter to keep track of the number of clusters that have been evaluated at each candidate point. When appropriate, it solves the master problem with the latest model to obtain a new candidate iterate and will call `evaluate`.

In our implementation of both this algorithm and its more sophisticated cousin Algorithm ATR of Section 4, we may define a single task to consist of the evaluation of more than one cluster $\mathcal{N}_j$. We may bundle, say, 5 or 10 clusters into a single task, in the interests of making the task large enough to justify the master's effort in packing its data and unpacking its results, and to maintain the ratio of compute time to communication cost at a high level. For purposes of simplicity, however, we assume in the descriptions both of this algorithm and of ATR that each task consists of a single cluster.

The implementation depends on a "synchronicity" parameter σ which is the proportion of clusters that must be evaluated at a point to trigger the generation of a new candidate iterate. Typical values of σ are in the range 0.25 to 0.9. A logical variable $\texttt{speceval}_k$ keeps track of whether x^k has yet triggered a new candidate. Initially, $\texttt{speceval}_k$ is set to `false`, then set to `true` when the proportion of evaluated clusters passes the threshold σ.

We now specify all the methods making up Algorithm ALS.

ALS: $\texttt{partial\_evaluate}(x^q, q, j, \mathcal{Q}_{[j]}(x^q), g_j)$
Given x^q, index q, and partition number j, evaluate $\mathcal{Q}[j](x^q)$ from (11)
    together with a partial subgradient g_j from (13);
Activate $\texttt{act\_on\_completed\_task}(x^q, q, j, \mathcal{Q}_{[j]}(x^q), g_j)$ on the master processor.

ALS: $\texttt{evaluate}(x^q, q)$
for $j = 1, 2, \ldots, T$ (possibly concurrently)
    $\texttt{partial\_evaluate}(x^q, q, j, \mathcal{Q}_{[j]}(x^q), g_j)$;
end (for)

ALS: `initialize`
choose tolerance ϵ_{tol};
choose starting point x^0;
choose threshold $\sigma \in (0, 1]$;
$\mathcal{Q}_{\min} \leftarrow \infty$;
$k \leftarrow 0$, $\texttt{speceval}_0 \leftarrow \texttt{false}$, $t_0 \leftarrow 0$;
$\texttt{evaluate}(x^0, 0)$.

ALS: $\texttt{act\_on\_completed\_task}(x^q, q, j, \mathcal{Q}_{[j]}(x^q), g_j)$
$t_q \leftarrow t_q + 1$;
add $\mathcal{Q}_{[j]}(x^q)$ and cut g_j to the model m;
if $t_q = T$
    $\mathcal{Q}_{\min} \leftarrow \min(\mathcal{Q}_{\min}, \mathcal{Q}(x^q))$;
else if $t_q \geq \sigma T$ **and** not $\texttt{speceval}_q$
    $\texttt{speceval}_q \leftarrow \texttt{true}$;
    $k \leftarrow k + 1$;
    solve current model problem (18) to obtain x^{k+1};
    if $\mathcal{Q}_{\min} - m(x^{k+1}) \leq \epsilon_{\text{tol}}(1 + |\mathcal{Q}_{\min}|)$
        STOP;
    $\texttt{evaluate}(x^k, k)$;
end (if)

We present results for Algorithm ALS in Section 6. While the algorithm is able to use a large number of worker processors on our opportunistic platform, it suffers from the usual drawbacks of the L-shaped method, namely, that cuts, once generated, must be retained for the remainder of the computation to ensure convergence and that large steps are typically taken on early iterations before a sufficiently good model approximation to $\mathcal{Q}(x)$ is created, making it impossible to exploit prior knowledge about the location of the solution.

3. A Bundle-Trust-Region Method

Trust-region approaches can be implemented by making only minor modifications to implementations of the L-shaped method, and they possesses several practical advantages along with stronger convergence properties. The trust-region methods we describe here are related to the regularized decomposition method of Ruszczyński [21] and the bundle-trust-region approaches of Kiwiel [16] and Hirart-Urruty and Lemaréchal [14, Chapter XV]. The main differences are that we use box-shaped trust regions yielding linear programming subproblems (rather than quadratic programs) and that our methods manipulate the size of the trust region directly rather than indirectly via a regularization parameter.

When requesting a subgradient of $\mathcal{Q}$ at some point x, our algorithms do not require particular (e.g., extreme) elements of the subdifferential to be supplied. Nor do they require the subdifferential $\partial\mathcal{Q}(x)$ to be representable as a convex combination of a finite number of vectors. In this respect, our algorithms contrast with that of Ruszczyński [21], for instance, which exploits the piecewise-linear nature of the objectives $\mathcal{Q}_i$ in (3). Because of our weaker conditions on the subgradient information, we cannot prove a finite termination result of the type presented in [21, Section 3]. However, these conditions potentially allow our algorithms to be extended to a more general class of convex nondifferentiable functions. We hope to explore these generalizations in future work.

3.1. A Method Based on ℓ_∞ Trust Regions

A key difference between the trust-region approach of this section and the L-shaped method of the preceding section is that we impose an ℓ_∞ norm bound on the size of the step. It is implemented by simply adding bound constraints to the linear programming subproblem (17) as follows:

$$-\Delta e \le x - x^k \le \Delta e, \tag{19}$$

where $e = (1, 1, \ldots, 1)^T$, Δ is the trust-region radius, and x^k is the current iterate. During the kth iteration, it may be necessary to solve several problems with trust regions of the form (19), with different model functions m and possibly different values of Δ, before a satisfactory new iterate x^{k+1} is identified. We refer to x^k and x^{k+1} as *major iterates* and the points $x^{k,\ell}$, $\ell = 0, 1, 2, \ldots$ obtained by minimizing the current model function subject to the constraints and trust-region bounds of the form (19) as *minor iterates*. Another key difference between the trust-region approach and the L-shaped approach is that a minor iterate $x^{k,\ell}$ is accepted as the new major iterate x^{k+1} only if it yields a substantial reduction in the objective function $\mathcal{Q}$ over the previous iterate x^k, in a sense to be defined below. A further important difference is that one can delete optimality cuts from the model functions, between minor and major iterations, without compromising the convergence properties of the algorithm.

To specify the method, we need to augment the notation established in the previous section. We define $m_{k,\ell}(x)$ to be the model function after ℓ minor iterations have been performed at iteration k, and $\Delta_{k,\ell} > 0$ to be the trust-region radius at the same stage. Under Assumption 1, there are no feasibility cuts, so that the problem to be solved to obtain the minor iteration $x^{k,\ell}$ is as follows:

$$\min_x m_{k,\ell}(x) \ \text{ subject to } Ax = b, \ x \geq 0, \ \|x - x^k\|_\infty \leq \Delta_{k,\ell} \tag{20}$$

(cf. (15)). By expanding this problem in a similar fashion to (17), we obtain

$$\min_{x,\theta_1,\ldots,\theta_T} \ c^T x + \sum_{j=1}^{T} \theta_j, \quad \text{subject to} \tag{21a}$$

$$\theta_j e \geq F_{[j]}^{k,\ell} x + f_{[j]}^{k,\ell}, \quad j = 1,2,\ldots,T, \tag{21b}$$

$$Ax = b, \ \ x \geq 0, \tag{21c}$$

$$-\Delta_{k,\ell} e \leq x - x^k \leq \Delta_{k,\ell} e. \tag{21d}$$

We assume the initial model $m_{k,0}$ at major iteration k to satisfy the following two properties:

$$m_{k,0}(x^k) = \mathcal{Q}(x^k), \tag{22a}$$

$$m_{k,0} \text{ is a piecewise linear underestimate of } \mathcal{Q}. \tag{22b}$$

Denoting the solution of the subproblem (21) by $x^{k,\ell}$, we accept this point as the new iterate x^{k+1} if the decrease in the actual objective $\mathcal{Q}$ (see (5)) is at least some fraction of the decrease predicted by the model function $m_{k,\ell}$. That is, for some constant $\xi \in (0, 1/2)$, the acceptance test is

$$\mathcal{Q}(x^{k,\ell}) \leq \mathcal{Q}(x^k) - \xi \left(\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \right). \tag{23}$$

(A typical value for ξ is 10^{-4}.)

If the test (23) fails to hold, we obtain a new model function $m_{k,\ell+1}$ by adding and possibly deleting cuts from $m_{k,\ell}(x)$. This process aims to refine the model function, so that it eventually generates a new major iteration, while economizing on storage by allowing deletion of subgradients that no longer seem helpful. Addition and deletion of cuts are implemented by adding and deleting rows from $F_{[j]}^{k,\ell}$ and $f_{[j]}^{k,\ell}$, to obtain $F_{[j]}^{k,\ell+1}$ and $f_{[j]}^{k,\ell+1}$, for $j = 1,2,\ldots,T$.

Given some parameter $\eta \in [0, 1)$, we obtain $m_{k,\ell+1}$ from $m_{k,\ell}$ by means of the following procedure:

Procedure Model-Update (k, ℓ)
for each optimality cut
  `possible_delete` $\leftarrow$ `true`;
  if the cut was generated at x^k
    `possible_delete` $\leftarrow$ `false`;
  else if the cut is active at the solution of (21)
    `possible_delete` $\leftarrow$ `false`;
  else if the cut was generated at an earlier minor iteration
    $\bar{\ell} = 0, 1, \ldots, \ell - 1$ such that

$$\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) > \eta \left[\mathcal{Q}(x^k) - m_{k,\bar{\ell}}(x^{k,\bar{\ell}}) \right] \tag{24}$$

    `possible_delete` $\leftarrow$ `false`;
  end (if)
  if `possible_delete`
    possibly delete the cut;
end (for each)
add optimality cuts obtained from each of the component functions
  $\mathcal{Q}_{[j]}$ at $x^{k,\ell}$.

In our implementation, we delete the cut if `possible_delete` is true at the final conditional statement and, in addition, the cut has not been active during the last 100 solutions of (21). More details are given in Section 6.2.

Because we retain all cuts active at x^k during the course of major iteration k, the following extension of (22a) holds:

$$m_{k,\ell}(x^k) = \mathcal{Q}(x^k), \quad \ell = 0, 1, 2, \ldots. \tag{25}$$

Since we add only subgradient information, the following generalization of (22b) also holds uniformly:

$$m_{k,\ell} \text{ is a piecewise linear underestimate of } \mathcal{Q}, \text{ for } \ell = 0, 1, 2, \ldots. \tag{26}$$

We may also decrease the trust-region radius $\Delta_{k,\ell}$ between minor iterations (that is, choose $\Delta_{k,\ell+1} < \Delta_{k,\ell}$) when the test (23) fails to hold. We do so if the match between model and objective appears to be particularly poor. If $\mathcal{Q}(x^{k,\ell})$ exceeds $\mathcal{Q}(x^k)$ by more than an estimate of the quantity

$$\max_{\|x - x^k\|_\infty \le 1} \mathcal{Q}(x^k) - \mathcal{Q}(x), \tag{27}$$

we conclude that the "upside" variation of the function $\mathcal{Q}$ deviates too much from its "downside" variation, and we choose the new radius $\Delta_{k,\ell+1}$ to bring these quantities more nearly into line. Our estimate of (27) is simply

$$\frac{1}{\min(1, \Delta_{k,\ell})} \left[\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \right],$$

that is, an extrapolation of the model reduction on the current trust region to a trust region of radius 1. Our complete strategy for reducing Δ is therefore as follows. (The `counter` is initialized to zero at the start of each major iteration.)

Procedure Reduce-Δ
evaluate

$$\rho = \min(1, \Delta_{k,\ell}) \frac{\mathcal{Q}(x^{k,\ell}) - \mathcal{Q}(x^k)}{\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell})}; \tag{28}$$

if $\rho > 0$
    `counter` $\leftarrow$ `counter`$+1$;
if $\rho > 3$ **or** (`counter` ≥ 3 **and** $\rho \in (1,3]$)
    set

$$\Delta_{k,\ell+1} = \frac{1}{\min(\rho, 4)} \Delta_{k,\ell};$$

    reset `counter` $\leftarrow 0$;

This procedure is related to the technique of Kiwiel [16, p. 109] for increasing the coefficient of the quadratic penalty term in his regularized bundle method.

If the test (23) is passed, so that we have $x^{k+1} = x^{k,\ell}$, we have a great deal of flexibility in defining the new model function $m_{k+1,0}$. We require only that the properties (22) are satisfied, with $k+1$ replacing k. Hence, we are free to delete much of the optimality cut information accumulated at iteration k (and previous iterates). In practice, of course, it is wise to delete only those cuts that have been inactive for a substantial number of iterations; otherwise we run the risk that many new function and subgradient evaluations will be required to restore useful model information that was deleted prematurely.

If the step to the new major iteration x^{k+1} shows a particularly close match between the true function $\mathcal{Q}$ and the model function $m_{k,\ell}$ at the last minor iteration of iteration k, we consider increasing the trust-region radius. Specifically, if

$$\mathcal{Q}(x^{k,\ell}) \leq \mathcal{Q}(x^k) - 0.5\left(\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell})\right), \quad \|x^k - x^{k,\ell}\|_\infty = \Delta_{k,\ell}, \tag{29}$$

then we set

$$\Delta_{k+1,0} = \min(\Delta_{\mathrm{hi}}, 2\Delta_{k,\ell}), \tag{30}$$

where Δ_{hi} is a prespecified upper bound on the radius.

Before specifying the algorithm formally, we define the convergence test. Given a parameter $\epsilon_{\mathrm{tol}} > 0$, we terminate if

$$\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \leq \epsilon_{\mathrm{tol}}(1 + |\mathcal{Q}(x^k)|). \tag{31}$$

Algorithm TR
choose $\xi \in (0, 1/2)$, maximum trust region Δ_{hi}, tolerance ϵ_{tol};
choose starting point x^0;
define initial model $m_{0,0}$ with the properties (22) (for $k = 0$);
choose $\Delta_{0,0} \in (0, \Delta_{\mathrm{hi}}]$;
for $k = 0, 1, 2, \ldots$
    `finishedMinorIteration` $\leftarrow$ `false`;

$\ell \leftarrow 0$; `counter` $\leftarrow 0$;
repeat
  solve (20) to obtain $x^{k,\ell}$;
  if (31) is satisfied
    STOP with approximate solution x^k;
  evaluate function and subgradient at $x^{k,\ell}$;
  if (23) is satisfied
    set $x^{k+1} = x^{k,\ell}$;
    obtain $m_{k+1,0}$ by possibly deleting cuts from $m_{k,\ell}$, but
      retaining the properties (22) (with $k+1$ replacing k);
    choose $\Delta_{k+1,0} \in [\Delta_{k,\ell}, \Delta_{\text{hi}}]$ according to (29), (30);
    `finishedMinorIteration` $\leftarrow$ `true`;
  else
    obtain $m_{k,\ell+1}$ from $m_{k,\ell}$ via Procedure Model-Update (k,ℓ);
    obtain $\Delta_{k,\ell+1}$ via Procedure Reduce-Δ;
  $\ell \leftarrow \ell + 1$;
until `finishedMinorIteration`
end (for)

3.2. Analysis of the Trust-Region Method

We now describe the convergence properties of Algorithm TR. We show that for $\epsilon_{\text{tol}} = 0$, the algorithm either terminates at a solution or generates a sequence of major iterates that approaches the solution set $\mathcal{S}$ (Theorem 2). When $\epsilon_{\text{tol}} > 0$, the algorithm terminates finitely; that is, it avoids generating infinite sequences either of major or minor iterates (Theorem 3).

Given some starting point x^0 satisfying the constraints $Ax^0 = b$, $x^0 \geq 0$, and setting $\mathcal{Q}_0 = \mathcal{Q}(x^0)$, we define the following quantities that are useful in describing and analyzing the algorithm:

$$\mathcal{L}(\mathcal{Q}_0) = \{x \mid Ax = b, x \geq 0, \mathcal{Q}(x) \leq \mathcal{Q}_0\}, \tag{32}$$

$$\mathcal{L}(\mathcal{Q}_0; \Delta) = \{x \mid \|x - y\| \leq \Delta, \text{ for some } y \in \mathcal{L}(\mathcal{Q}_0)\}, \tag{33}$$

$$\beta = \sup\{\|g\|_1 \mid g \in \partial\mathcal{Q}(x), \text{ for some } x \in \mathcal{L}(\mathcal{Q}_0; \Delta_{\text{hi}})\}. \tag{34}$$

Using Assumption 1, we can easily show that $\beta < \infty$.

We start by showing that the optimal objective value for (20) cannot decrease from one minor iteration to the next.

Lemma 1. *Suppose that $x^{k,\ell}$ does not satisfy the acceptance test (23). Then we have*

$$m_{k,\ell}(x^{k,\ell}) \leq m_{k,\ell+1}(x^{k,\ell+1}).$$

Proof. In obtaining $m_{k,\ell+1}$ from $m_{k,\ell}$ in Model-Update, we do not allow deletion of cuts that were active at the solution $x^{k,\ell}$ of (21). Using $\bar{F}^{k,\ell}_{[j]}$ and $\bar{f}^{k,\ell}_{[j]}$ to denote the active rows in $F_{[j]}^{k,\ell}$ and $f_{[j]}^{k,\ell}$, we have that $x^{k,\ell}$ is also the solution of the following linear program (in which the inactive cuts are not present):

$$\min_{x,\theta_1,\ldots,\theta_T} \; c^T x + \sum_{j=1}^{T} \theta_j, \quad \text{subject to} \tag{35a}$$

$$\theta_j e \geq \bar{F}_{[j]}^{k,\ell} x + \bar{f}_{[j]}^{k,\ell}, \quad j = 1, 2, \ldots, T, \tag{35b}$$

$$Ax = b, \;\; x \geq 0, \tag{35c}$$

$$-\Delta_{k,\ell} e \leq x - x^k \leq \Delta_{k,\ell} e. \tag{35d}$$

The subproblem to be solved for $x^{k,\ell+1}$ differs from (35) in two ways. First, additional rows may be added to $\bar{F}_{[j]}^{k,\ell}$ and $\bar{f}_{[j]}^{k,\ell}$, consisting of function values and subgradients obtained at $x^{k,\ell}$ and also inactive cuts carried over from the previous (21). Second, the trust-region radius $\Delta_{k,\ell+1}$ may be smaller than $\Delta_{k,\ell}$. Hence, the feasible region of the problem to be solved for $x^{k,\ell+1}$ is a subset of the feasible region for (35), so the optimal objective value cannot be smaller.

Next we have a result about the amount of reduction in the model function $m_{k,\ell}$.

Lemma 2. *For all $k = 0, 1, 2, \ldots$ and $\ell = 0, 1, 2, \ldots$, we have that*

$$\begin{aligned} m_{k,\ell}(x^k) - m_{k,\ell}(x^{k,\ell}) &= \mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \\ &\geq \min\left(\Delta_{k,\ell}, \|x^k - P(x^k)\|_\infty\right) \frac{\mathcal{Q}(x^k) - \mathcal{Q}^*}{\|x^k - P(x^k)\|_\infty} && \text{(36a)} \\ &\geq \hat{\epsilon} \min\left(\Delta_{k,\ell}, \|x^k - P(x^k)\|_\infty\right), && \text{(36b)} \end{aligned}$$

where $\hat{\epsilon} > 0$ is defined in (10).

Proof. The first equality follows immediately from (25), while the second inequality (36b) follows immediately from (36a) and (10). We now prove (36a).

Consider the following subproblem in the scalar τ:

$$\min_{\tau \in [0,1]} m_{k,\ell}\left(x^k + \tau[P(x^k) - x^k]\right) \;\; \text{subject to} \;\; \left\|\tau[P(x^k) - x^k]\right\|_\infty \leq \Delta_{k,\ell}. \tag{37}$$

Denoting the solution of this problem by $\tau_{k,\ell}$, we have by comparison with (20) that

$$m_{k,\ell}(x^{k,\ell}) \leq m_{k,\ell}\left(x^k + \tau_{k,\ell}[P(x^k) - x^k]\right). \tag{38}$$

If $\tau = 1$ is feasible in (37), we have from (38) and (26) that

$$\begin{aligned} & m_{k,\ell}(x^{k,\ell}) \leq m_{k,\ell}\left(x^k + \tau_{k,\ell}[P(x^k) - x^k]\right) \\ & \leq m_{k,\ell}\left(x^k + [P(x^k) - x^k]\right) = m_{k,\ell}(P(x^k)) \leq \mathcal{Q}(P(x^k)) = \mathcal{Q}^*. \end{aligned}$$

Therefore, when $\tau = 1$ is feasible for (37), we have from (25) that

$$m_{k,\ell}(x^k) - m_{k,\ell}(x^{k,\ell}) \geq \mathcal{Q}(x^k) - \mathcal{Q}^*,$$

so that (36a) holds in this case.

When $\tau = 1$ is infeasible for (37), consider setting $\tau = \Delta_{k,\ell}/\|x^k - P(x^k)\|_\infty$ (which is certainly feasible for (37)). We have from (38), the definition of $\tau_{k,\ell}$, the fact (26) that $m_{k,\ell}$ underestimates $\mathcal{Q}$, and convexity of $\mathcal{Q}$ that

$$\begin{aligned} m_{k,\ell}(x^{k,\ell}) &\le m_{k,\ell}\left(x^k + \Delta_{k,\ell}\frac{P(x^k) - x^k}{\|P(x^k) - x^k\|_\infty}\right) \\ &\le \mathcal{Q}\left(x^k + \Delta_{k,\ell}\frac{P(x^k) - x^k}{\|P(x^k) - x^k\|_\infty}\right) \\ &\le \mathcal{Q}(x^k) + \frac{\Delta_{k,\ell}}{\|P(x^k) - x^k\|_\infty}(\mathcal{Q}^* - \mathcal{Q}(x^k)). \end{aligned}$$

Therefore, using (25), we have

$$m_{k,\ell}(x^k) - m_{k,\ell}(x^{k,\ell}) \ge \frac{\Delta_{k,\ell}}{\|P(x^k) - x^k\|_\infty}[\mathcal{Q}(x^k) - \mathcal{Q}^*],$$

verifying (36a) in this case as well.

Our next result finds a lower bound on the trust-region radii $\Delta_{k,\ell}$. For purposes of this result we define a quantity E_k to measure the closest approach to the solution set for all iterates up to and including x^k, that is,

$$E_k \stackrel{\text{def}}{=} \min_{\bar{k}=0,1,\ldots,k} \|x^{\bar{k}} - P(x^{\bar{k}})\|_\infty. \tag{39}$$

Note that E_k decreases monotonically with k. We also define Δ_{init} to be the initial value of the trust region.

Lemma 3. *There is a constant $\Delta_{\text{lo}} > 0$ such that for all trust regions $\Delta_{k,\ell}$ used in the course of Algorithm TR, we have*

$$\Delta_{k,\ell} \ge \min(\Delta_{\text{lo}}, E_k/4).$$

Proof. We prove the result by showing that the value $\Delta_{\text{lo}} = (1/4)\min(1, \Delta_{\text{init}}, \hat{\epsilon}/\beta)$ has the desired property, where $\hat{\epsilon}$ is from (10) and β is from (34).

Suppose for contradiction that there are indices k and ℓ such that

$$\Delta_{k,\ell} < \frac{1}{4}\min\left(1, \frac{\hat{\epsilon}}{\beta}, \Delta_{\text{init}}, E_k\right).$$

Since the trust region can be reduced by at most a factor of 4 by Procedure Reduce-Δ, there must be an earlier trust region radius $\Delta_{\bar{k},\bar{\ell}}$ (with $\bar{k} \le k$) such that

$$\Delta_{\bar{k},\bar{\ell}} < \min\left(1, \frac{\hat{\epsilon}}{\beta}, E_k\right), \tag{40}$$

and $\rho > 1$ in (28), that is,

$$\begin{aligned} \mathcal{Q}(x^{\bar{k},\bar{\ell}}) - \mathcal{Q}(x^{\bar{k}}) &> \frac{1}{\min(1, \Delta_{\bar{k},\bar{\ell}})} \left(\mathcal{Q}(x^{\bar{k}}) - m_{\bar{k},\bar{\ell}}(x^{\bar{k},\bar{\ell}}) \right) \\ &= \frac{1}{\Delta_{\bar{k},\bar{\ell}}} \left(\mathcal{Q}(x^{\bar{k}}) - m_{\bar{k},\bar{\ell}}(x^{\bar{k},\bar{\ell}}) \right). \end{aligned} \tag{41}$$

By applying Lemma 2, and using (40), we have

$$\mathcal{Q}(x^{\bar{k}}) - m_{\bar{k},\bar{\ell}}(x^{\bar{k},\bar{\ell}}) \geq \hat{\epsilon} \min \left(\Delta_{\bar{k},\bar{\ell}}, \|x^{\bar{k}} - P(x^{\bar{k}})\|_\infty \right) = \hat{\epsilon} \Delta_{\bar{k},\bar{\ell}} \tag{42}$$

where the last equality follows from $\|x^{\bar{k}} - P(x^{\bar{k}})\|_\infty \geq E_{\bar{k}} \geq E_k$ and (40). By combining (42) with (41), we have that

$$\mathcal{Q}(x^{\bar{k},\bar{\ell}}) - \mathcal{Q}(x^{\bar{k}}) > \hat{\epsilon}. \tag{43}$$

By using standard properties of subgradients, we have

$$\begin{aligned} &\mathcal{Q}(x^{\bar{k},\bar{\ell}}) - \mathcal{Q}(x^{\bar{k}}) \leq g_{\bar{\ell}}^T (x^{\bar{k},\bar{\ell}} - x^{\bar{k}}) \\ &\leq \|g_{\bar{\ell}}\|_1 \|x^{\bar{k}} - x^{\bar{k},\bar{\ell}}\|_\infty \leq \|g_{\bar{\ell}}\|_1 \Delta_{\bar{k},\bar{\ell}}, \ \text{ for all } g_{\bar{\ell}} \in \partial \mathcal{Q}(x^{\bar{k},\bar{\ell}}). \end{aligned} \tag{44}$$

By combining this expression with (43), and using (40) again, we obtain that

$$\|g_{\bar{\ell}}\|_1 \geq \frac{\hat{\epsilon}}{\Delta_{\bar{k},\bar{\ell}}} > \beta.$$

However, since $x^{\bar{k},\bar{\ell}} \in \mathcal{L}(\mathcal{Q}_0; \Delta_{\text{hi}})$, we have from (34) that $\|g_{\bar{\ell}}\|_1 \leq \beta$, giving a contradiction.

Finite termination of the inner iterations is proved in the following two results. Recall that the parameters ξ and η are defined in (23) and (24), respectively.

Lemma 4. *Let $\epsilon_{\text{tol}} = 0$ in Algorithm TR, and let $\bar{\eta}$ be any constant satisfying $0 < \bar{\eta} < 1$, $\bar{\eta} > \xi$, $\bar{\eta} \geq \eta$. Let ℓ_1 be any index such that x^{k,ℓ_1} fails to satisfy the test (23). Then either the sequence of inner iterations eventually yields a point x^{k,ℓ_2} satisfying the acceptance test (23), or there is an index $\ell_2 > \ell_1$ such that*

$$\mathcal{Q}(x^k) - m_{k,\ell_2}(x^{k,\ell_2}) \leq \bar{\eta} \left[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1}) \right]. \tag{45}$$

Proof. Suppose for contradiction that the none of the minor iterations following ℓ_1 satisfies either (23) or the criterion (45); that is,

$$\begin{aligned} \mathcal{Q}(x^k) - m_{k,q}(x^{k,q}) &> \bar{\eta} \left[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1}) \right], \\ &\geq \eta \left[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1}) \right], \ \text{ for all } q > \ell_1. \end{aligned} \tag{46}$$

It follows from this bound, together with Lemma 1 and Procedure Model-Update, that none of the cuts generated at minor iterations $q \geq \ell_1$ is deleted.

We assume in the remainder of the proof that q and ℓ are generic minor iteration indices that satisfy

$$q > \ell \geq \ell_1.$$

Because the function and subgradients from minor iterations $x^{k,\ell}$, $l = l_1, l_1 + 1, \ldots$ are retained throughout the major iteration k, we have

$$m_{k,q}(x^{k,\ell}) = \mathcal{Q}(x^{k,\ell}). \tag{47}$$

By definition of the subgradient, we have

$$m_{k,q}(x) - m_{k,q}(x^{k,\ell}) \geq g^T(x - x^{k,\ell}), \;\; \text{for all } g \in \partial m_{k,q}(x^{k,\ell}). \tag{48}$$

Therefore, from (26) and (47), it follows that

$$\mathcal{Q}(x) - \mathcal{Q}(x^{k,\ell}) \geq g^T(x - x^{k,\ell}), \;\; \text{for all } g \in \partial m_{k,q}(x^{k,\ell}),$$

so that

$$\partial m_{k,q}(x^{k,\ell}) \subset \partial \mathcal{Q}(x^{k,\ell}). \tag{49}$$

Since $\mathcal{Q}(x^k) < \mathcal{Q}(x^0) = \mathcal{Q}_0$, we have from (32) that $x^k \in \mathcal{L}(\mathcal{Q}_0)$. Therefore, from the definition (33) and the fact that $\|x^{k,\ell} - x^k\| \leq \Delta_{k,\ell} \leq \Delta_{\mathrm{hi}}$, we have that $x^{k,\ell} \in \mathcal{L}(\mathcal{Q}_0; \Delta_{\mathrm{hi}})$. It follows from (34) and (49) that

$$\|g\|_1 \leq \beta, \;\; \text{for all } g \in \partial m_{k,q}(x^{k,\ell}). \tag{50}$$

Since $x^{k,\ell}$ is rejected by the test (23), we have from (47) and Lemma 1 that the following inequalities hold:

$$\begin{aligned} m_{k,q}(x^{k,\ell}) = \mathcal{Q}(x^{k,\ell}) &\geq \mathcal{Q}(x^k) - \xi\left[\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell})\right] \\ &\geq \mathcal{Q}(x^k) - \xi\left[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1})\right]. \end{aligned}$$

By rearranging this expression, we obtain

$$\mathcal{Q}(x^k) - m_{k,q}(x^{k,\ell}) \leq \xi\left[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1})\right]. \tag{51}$$

Consider now all points x satisfying

$$\|x - x^{k,\ell}\|_\infty \leq \frac{\bar{\eta} - \xi}{\beta}\left[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1})\right] \stackrel{\text{def}}{=} \zeta > 0. \tag{52}$$

Using this bound together with (48) and (50), we obtain

$$\begin{aligned} &m_{k,q}(x^{k,\ell}) - m_{k,q}(x) \leq g^T(x^{k,\ell} - x) \\ &\leq \beta\|x^{k,\ell} - x\|_\infty \leq (\bar{\eta} - \xi)\left[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1})\right]. \end{aligned}$$

By combining this bound with (51), we find that the following bound is satisfied for all x in the neighborhood (52):

$$\begin{aligned} \mathcal{Q}(x^k) - m_{k,q}(x) &= \left[\mathcal{Q}(x^k) - m_{k,q}(x^{k,\ell})\right] + \left[m_{k,q}(x^{k,\ell}) - m_{k,q}(x)\right] \\ &\leq \bar{\eta}\left[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1})\right]. \end{aligned}$$

It follows from this bound, in conjunction with (46), that $x^{k,q}$ (the solution of the trust-region problem with model function $m_{k,q}$) cannot lie in the neighborhood (52). Therefore, we have

$$\|x^{k,q} - x^{k,\ell}\|_\infty > \zeta. \tag{53}$$

But since $\|x^{k,\ell} - x^k\|_\infty \le \Delta_k \le \Delta_{\rm hi}$ for all $\ell \ge \ell_1$, it is impossible for an infinite sequence $\{x^{k,\ell}\}_{\ell \ge \ell_1}$ to satisfy (53). We conclude that (45) must hold for some $\ell_2 \ge \ell_1$, as claimed.

We now show that the minor iteration sequence terminates at a point $x^{k,\ell}$ satisfying the acceptance test, provided that x^k is not a solution.

Theorem 1. *Suppose that* $\epsilon_{\rm tol} = 0$.

(i) If $x^k \notin \mathcal{S}$, *there is an* $\ell \ge 0$ *such that* $x^{k,\ell}$ *satisfies (23).*
(ii) If $x^k \in \mathcal{S}$, *then either Algorithm TR terminates (and verifies that* $x^k \in \mathcal{S}$*), or* $\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \downarrow 0$.

Proof. Suppose for the moment that the inner iteration sequence is infinite, that is, the test (23) always fails. By applying Lemma 4 recursively, with any constant $\bar\eta$ satisfying the properties stated in Lemma 4, we can identify a sequence of indices $0 < \ell_1 < \ell_2 < \dots$ such that

$$\begin{aligned} \mathcal{Q}(x^k) - m_{k,\ell_j}(x^{k,\ell_j}) &\le \bar\eta \left[\mathcal{Q}(x^k) - m_{k,\ell_{j-1}}(x^{k,\ell_{j-1}})\right] \\ &\le \bar\eta^2 \left[\mathcal{Q}(x^k) - m_{k,\ell_{j-2}}(x^{k,\ell_{j-2}})\right] \\ &\vdots \\ &\le \bar\eta^j \left[\mathcal{Q}(x^k) - m_{k,0}(x^{k,0})\right]. \end{aligned} \tag{54}$$

When $x^k \notin \mathcal{S}$, we have from Lemma 3 that

$$\Delta_{k,\ell} \ge \min(\Delta_{\rm lo}, E_k/4) \stackrel{\rm def}{=} \bar\Delta_{\rm lo} > 0, \quad \text{for all } \ell = 0, 1, 2, \dots,$$

so the right-hand side of (36a) is strictly positive. Hence for j sufficiently large, we have that

$$\mathcal{Q}(x^k) - m_{k,\ell_j}(x^{k,\ell_j}) \le 0.5 \min\left(\bar\Delta_{\rm lo}, \|x^k - P(x^k)\|_\infty\right) \frac{\mathcal{Q}(x^k) - \mathcal{Q}^*}{\|x^k - P(x^k)\|_\infty}.$$

But this inequality contradicts (36), proving (i).

For the case of $x^k \in \mathcal{S}$, there are two possibilities. If the inner iteration sequence terminates finitely at some $x^{k,\ell}$, we have $\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) = 0$ and indeed that

$$m_{k,\ell}(x) \ge \mathcal{Q}(x^k) = \mathcal{Q}^*, \quad \text{for all } x \text{ with } \|x - x^k\|_\infty \le \Delta_{k,\ell}.$$

Because of (26), we have that $\mathcal{Q}(x) \ge \mathcal{Q}(x^k)$ for all x in a neighborhood of x^k, implying that $0 \in \partial\mathcal{Q}(x^k)$. Therefore, termination under these circumstances yields a guarantee that $x^k \in \mathcal{S}$. When the algorithm does not terminate, it follows from (54) that $\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \to 0$. By applying Lemma 1, we verify our claim (ii) of monotonic convergence.

We now prove convergence of Algorithm TR to $\mathcal{S}$.

Theorem 2. *Suppose that $\epsilon_{\rm tol} = 0$. The sequence of major iterations $\{x^k\}$ is either finite, terminating at some $x^k \in \mathcal{S}$, or is infinite, with the property that $\|x^k - P(x^k)\|_\infty \to 0$.*

Proof. If the claim does not hold, there are two possibilities. The first is that the sequence of major iterations terminates finitely at some $x^k \notin \mathcal{S}$. However, Theorem 1 ensures, however, that the minor iteration sequence will terminate at some new major iteration x^{k+1} under these circumstances, so we can rule out this possibility. The second possibility is that the sequence $\{x^k\}$ is infinite but that there is some $\epsilon > 0$ and an infinite subsequence of indices $\{k_j\}_{j=1,2,\ldots}$ such that

$$\|x^{k_j} - P(x^{k_j})\|_\infty \geq \epsilon, \quad j = 0, 1, 2, \ldots .$$

Since the sequence $\{\mathcal{Q}(x^{k_j})\}_{j=1,2,\ldots}$ is infinite, decreasing, and bounded below, it converges to some value $\bar{\mathcal{Q}} > \mathcal{Q}^*$. Moreover, since the entire sequence $\{\mathcal{Q}(x^k)\}$ is monotone decreasing, it follows that $\mathcal{Q}(x^k) > \bar{\mathcal{Q}}$ and therefore

$$\mathcal{Q}(x^k) - \mathcal{Q}^* > \bar{\mathcal{Q}} - \mathcal{Q}^* > 0, \quad k = 0, 1, 2, \ldots .$$

Hence, by boundedness of the subgradients (see (34)), we can identify a constant $\bar{\epsilon} > 0$ such that

$$\|x^k - P(x^k)\|_\infty \geq \bar{\epsilon}, \quad k = 0, 1, 2, \ldots .$$

It follows from (39) that

$$E_k \geq \bar{\epsilon}, \quad k = 0, 1, 2, \ldots . \tag{55}$$

For each major iteration index k, let $\ell(k)$ be the minor iteration index that passes the acceptance test (23). By combining (23) with Lemma 2, we have that

$$\mathcal{Q}(x^k) - \mathcal{Q}(x^{k+1}) \geq \xi \hat{\epsilon} \min\left(\Delta_{k,\ell(k)}, \|x^k - P(x^k)\|_\infty\right) \geq \xi \hat{\epsilon} \min\left(\Delta_{k,\ell(k)}, \bar{\epsilon}\right).$$

Since $\mathcal{Q}(x^k) - \mathcal{Q}(x^{k+1}) \to 0$, we deduce that

$$\lim_{k \to \infty} \Delta_{k,\ell(k)} = 0. \tag{56}$$

By Lemma 3 and (55), we have

$$\Delta_{k,\ell(k)} \geq \min(\Delta_{\rm lo}, \bar{\epsilon}/4) > 0, \quad k = 0, 1, 2, \ldots,$$

which contradicts (56). We conclude that the second possibility (an infinite sequence $\{x^k\}$ not converging to $\mathcal{S}$) cannot occur either, so the proof is complete.

Finally, we show that the algorithm terminates when $\epsilon_{\rm tol} > 0$.

Theorem 3. *When $\epsilon_{\rm tol} > 0$, Algorithm TR terminates finitely.*

Proof. We show first that the algorithm cannot "get stuck" at a particular x^k, generating an infinite sequence of minor iterations at x^k without eventually satisfying either (31) or the acceptance test (23). We see from the reasoning in the proof of Theorem 1 together with the monotonicity property of Lemma 1 that an infinite sequence of minor iterations must satisfy that

$$\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \downarrow 0. \tag{57}$$

Since the right-hand side of (31) is bounded below by ϵ_{tol}, the test (31) must be satisfied for some ℓ. Therefore, the minor iteration sequence cannot be infinite.

Now consider the other possibility of an infinite sequence of major iterations $\{x^k\}_{k=1,2,\ldots}$. Since we have

$$\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) > \epsilon_{\mathrm{tol}}$$

for all k and ℓ, and since the acceptance test (23) is satisfied at all k, we have

$$\mathcal{Q}(x^k) - \mathcal{Q}(x^{k+1}) \geq \xi \epsilon_{\mathrm{tol}} > 0, \quad \text{for all } k = 0, 1, 2 \ldots.$$

But this relation is inconsistent with the fact that $\{\mathcal{Q}(x^k)\}$ is bounded below (by $\mathcal{Q}^*$), so this possibility can also be ruled out, and the proof is complete.

3.3. Discussion

The algorithm can be modified in various ways without changing its properties greatly. For instance, we could replace the step norm bound in (20) by a scaled bound of the form

$$\|S(x - x^k)\|_\infty \leq \Delta_k,$$

where S is a diagonal positive definite matrix. After this modification, (21) remains a linear program. We could also use a 1-norm trust region, at the cost of introducing an additional variable vector s of the same dimension as x. Specifically, we enforce the constraint $\|x - x^k\|_1 \leq \Delta_k$ by enforcing the following linear constraints:

$$x - x^k \leq s, \quad x^k - x \leq s, \quad e^T s \leq \Delta_k.$$

Once again, we obtain a linear programming subproblem, albeit one that involves more variables than (21)

If a 2-norm trust region is used, we can show by comparing the optimality conditions for the respective problems that the solution of the subproblem

$$\min_x m_{k,\ell}(x) \text{ subject to } Ax = b, \; x \geq 0, \; \|x - x^k\|_2 \leq \Delta_k$$

is identical to the solution of

$$\min_x m_{k,\ell}(x) + \lambda \|x - x^k\|^2 \text{ subject to } Ax = b, \; x \geq 0, \tag{58}$$

for some $\lambda \geq 0$. We can transform (58) to a quadratic program in the same fashion as the transformation of (20) to (21). The bundle-trust-region approaches described in Kiwiel [16], Hirart-Urruty and Lemaréchal [14, Chapter XV], and Ruszczyński [21,22] also lead to problems of the form (58). These approaches manipulate the parameter λ rather than adjusting the trust-region radius, more in the spirit of the Levenberg-Marquardt method for least-squares problems than of a true trust-region method. Hence, their analysis differs somewhat from that of the preceding section. Moreover, although quadratic programming solvers that exploit the special structure of the quadratic term in (58) have been designed and implemented (see [21]), we believe that the linear programming subproblem (21) is more appealing from a practical point of view. Improvements in the efficiency and ease of use of linear programming software have continued to occur at a rapid pace, and availability of high-quality software has made it much easier to implement an efficient algorithm based on (21) than would have been the case if the subproblems had the form (58).

4. An Asynchronous Bundle-Trust-Region Method

In this section we present an asynchronous, parallel version of the trust-region algorithm of the preceding section and analyze its convergence properties.

4.1. Algorithm ATR

We now define a variant of the method of Section 3 that allows the partial sums $\mathcal{Q}_{[j]}, j = 1, 2, \ldots, T$ (11) and their associated cuts to be evaluated simultaneously for different values of x. We generate candidate iterates by solving trust-region subproblems centered on an "incumbent" iterate, which (after a startup phase) is the point x^I that, roughly speaking, is the best among those visited by the algorithm whose function value $\mathcal{Q}(x)$ is fully known.

By performing evaluations of $\mathcal{Q}$ at different points concurrently, we relax the strict synchronicity requirements of Algorithm TR, which requires $\mathcal{Q}(x^k)$ to be evaluated fully before the next candidate x^{k+1} is generated. The resulting approach, which we call Algorithm ATR (for "asynchronous TR"), is more suitable for implementation on computational grids of the type we consider here. Besides the obvious increase in parallelism that goes with evaluating several points at once, there is no longer a risk of the entire computation being help up by the slow evaluation of one of the partial sums $\mathcal{Q}_{[j]}$ on a recalcitrant worker. Algorithm ATR has similar theoretical properties to Algorithm TR, since the mechanisms for accepting a point as the new incumbent, adjusting the size of the trust region, and adding and deleting cuts are all similar to the corresponding mechanisms in Algorithm TR.

Algorithm ATR maintains a "basket" $\mathcal{B}$ of at most K points for which the value of $\mathcal{Q}$ and associated subgradient information is partially known. When the evaluation of $\mathcal{Q}(x^q)$ is completed for a particular point x^q in the basket, it is installed as the new incumbent if (i) its objective value is smaller than that of the current incumbent x^I; and (ii) it passes a trust-region acceptance test like

(23), with the incumbent *at the time x^q was generated* playing the role of the previous major iteration in Algorithm TR. Whether x^q becomes the incumbent or not, it is removed from the basket.

When a vacancy arises in the basket, we may generate a new point by solving a trust-region subproblem similar to (20), centering the trust region at the current incumbent x^I. During the startup phase, while the basket is being populated, we wait until the evaluation of some other point in the basket has reached a certain level of completion (that is, until a proportion $\sigma \in (0,1]$ of the partial sums (11) and their subgradients have been evaluated) before generating a new point. We use a logical variable $\mathtt{speceval}_q$ to indicate when the evaluation of x^q passes the specified threshold and to ensure that x^q does not trigger the evaluation of more than one new iterate. (Both σ and $\mathtt{speceval}_q$ play a similar role in Algorithm ALS.) After the startup phase is complete (that is, after the basket has been filled), vacancies arise only after evaluation of an iterate x^q is completed.

We use $m(\cdot)$ (without subscripts) to denote the model function for $\mathcal{Q}(\cdot)$. When generating a new iterate, we use whatever cuts are stored at the time to define m. When solved around the incumbent x^I with trust-region radius Δ, the subproblem is as follows:

$$\mathtt{trsub}(x^I, \Delta)\texttt{:} \quad \min_x m(x) \;\text{ subject to } Ax = b,\; x \geq 0,\; \|x - x^I\|_\infty \leq \Delta. \tag{59}$$

We refer to x^I as the *parent incumbent* of the solution of (59).

In the following description, we use k to index the successive points x^k that are explored by the algorithm, I to denote the index of the incumbent, and $\mathcal{B}$ to denote the basket. We use t_k to count the number of partial sums $\mathcal{Q}_{[j]}(x^k)$, $j = 1, 2, \ldots, T$ that have been evaluated so far.

Given a starting guess x^0, we initialize the algorithm by setting the dummy point x^{-1} to x^0, setting the incumbent index I to -1, and setting the initial incumbent value $\mathcal{Q}^I = \mathcal{Q}^{-1}$ to ∞. The iterate at which the first evaluation is completed becomes the first "serious" incumbent.

We now outline some other notation used in specifying Algorithm ATR:

- $\mathcal{Q}^I$: The objective value of the incumbent x^I, except in the case of $I = -1$, in which case $\mathcal{Q}^{-1} = \infty$.
- I_q: The index of the parent incumbent of x^q, that is, the incumbent index I at the time that x^q was generated from (59). Hence, $\mathcal{Q}^{I_q} = \mathcal{Q}(x^{I_q})$ (except when $I_q = -1$; see previous item).
- Δ_q: The value of the trust-region radius Δ used when solving for x^q.
- Δ_{curr}: Current value of the trust-region radius. When it comes time to solve (59) to obtain a new iterate x^q, we set $\Delta_q \leftarrow \Delta_{\text{curr}}$.
- m^q: The optimal value of the objective function m in the subproblem $\mathtt{trsub}(x^{I_q}, \Delta_q)$ (59).

Our strategy for maintaining the model closely follows that of Algorithm TR. Whenever the incumbent changes, we have a fairly free hand in deleting the cuts that define m, just as we do after accepting a new major iterate in Algorithm TR.

If the incumbent does not change for a long sequence of iterations (corresponding to a long sequence of minor iterations in Algorithm TR), we can still delete "stale" cuts that represent information in m that has likely been superseded (as quantified by a parameter $\eta \in [0,1)$). The following version of Procedure Model-Update, which applies to Algorithm ATR, takes as an argument the index k of the latest iterate generated by the algorithm. It is called after the evaluation of $\mathcal{Q}$ at an earlier iterate x^q has just been completed, but x^q does *not* meet the conditions needed to become the new incumbent.

Procedure Model-Update (k)
for each optimality cut defining m
  `possible_delete` $\leftarrow$ `true`;
  if the cut was generated at the parent incumbent I_k of k
    `possible_delete` $\leftarrow$ `false`;
  else if the cut was active at the solution x^k of `trsub`(x^{I_k}, Δ_k)
    `possible_delete` $\leftarrow$ `false`;
  else if the cut was generated at an earlier iteration $\bar{\ell}$
      such that $I_{\bar{\ell}} = I_k \neq -1$ and

$$\mathcal{Q}^{I_k} - m^k > \eta[\mathcal{Q}^{I_k} - m^{\bar{\ell}}] \tag{60}$$

    `possible_delete` $\leftarrow$ `false`;
  end (if)
  if `possible_delete`
    possibly delete the cut;
end (for each)

Our strategy for adjusting the trust region Δ_{curr} also follows that of Algorithm TR. The differences arise from the fact that between the time an iterate x^q is generated and its function value $\mathcal{Q}(x^q)$ becomes known, other adjustments of Δ_{current} may have occurred, as the evaluation of intervening iterates is completed. The version of Procedure Reduce-Δ for Algorithm ATR is as follows.

Procedure Reduce-$\Delta(q)$
if $I_q = -1$
  return;
evaluate

$$\rho = \min(1, \Delta_q) \frac{\mathcal{Q}(x^q) - \mathcal{Q}^{I_q}}{\mathcal{Q}^{I_q} - m^q}; \tag{61}$$

if $\rho > 0$
  `counter` $\leftarrow$ `counter`$+1$;
if $\rho > 3$ **or** (`counter` ≥ 3 **and** $\rho \in (1,3]$)
  set $\Delta_q^+ \leftarrow \Delta_q / \min(\rho, 4)$;
  set $\Delta_{\text{curr}} \leftarrow \min(\Delta_{\text{curr}}, \Delta_q^+)$;
  reset `counter` $\leftarrow 0$;
return.

The protocol for increasing the trust region after a successful step is based on (29), (30). If on completion of evaluation of $\mathcal{Q}(x^q)$, the iterate x^q becomes the new incumbent, then we test the following condition:

$$\mathcal{Q}(x^q) \le \mathcal{Q}^{I_q} - 0.5(\mathcal{Q}^{I_q} - m^q) \text{ and } \|x^q - x^{I_q}\|_\infty = \Delta_q. \tag{62}$$

If this condition is satisfied, we set

$$\Delta_{\text{curr}} \leftarrow \max(\Delta_{\text{curr}}, \min(\Delta_{\text{hi}}, 2\Delta_q)). \tag{63}$$

The convergence test is also similar to the test (31) used for Algorithm TR. We terminate if, on generation of a new iterate x^k, we find that

$$\mathcal{Q}^I - m^k \le \epsilon_{\text{tol}}(1 + |\mathcal{Q}^I|). \tag{64}$$

We now specify the four key routines of the Algorithm ATR, which serve a similar function to the four main routines of Algorithm ALS. As in the earlier case, we assume for simplicity of description that each task consists of evaluation of the function and a subgradient for a single cluster (although in practice we may bundle more than one cluster into a single task). The routine `partial_evaluate` executes on worker processors, while the other three routines execute on the master processor.

ATR: `partial_evaluate`$(x^q, q, j, \mathcal{Q}_{[j]}(x^q), g_j)$
Given x^q, index q, and partition number j, evaluate $\mathcal{Q}_{[j]}(x^q)$ from (11)
together with a partial subgradient g_j from (13);
Activate `act_on_completed_task`$(x^q, q, j, \mathcal{Q}_{[j]}(x^q), g_j)$ on the master processor.

ATR: `evaluate`(x^q, q)
for $j = 1, 2, \ldots, T$ (possibly concurrently)
`partial_evaluate`$(x^q, q, j, \mathcal{Q}_{[j]}(x^q), g_j)$;
end (for)

ATR: `initialization`(x^0)
choose $\xi \in (0, 1/2)$, trust region upper bound $\Delta_{\text{hi}} > 0$;
choose synchronicity parameter $\sigma \in (0, 1]$;
choose maximum basket size $K > 0$;
choose $\Delta_{\text{curr}} \in (0, \Delta_{\text{hi}}]$, `counter` $\leftarrow 0$; $\mathcal{B} \leftarrow \emptyset$;
$I \leftarrow -1$; $x^{-1} \leftarrow x^0$; $\mathcal{Q}^{-1} \leftarrow \infty$; $I_0 \leftarrow -1$;
$k \leftarrow 0$; `speceval`$_0 \leftarrow$ `false`; $t_0 \leftarrow 0$;
`evaluate`$(x^0, 0)$.

ATR: $\texttt{act\_on\_completed\_task}(x^q, q, j, \mathcal{Q}_{[j]}(x^q), g_j))$
$t_q \leftarrow t_q + 1$;
add $\mathcal{Q}_{[j]}(x^q)$ and cut g_j to the model m;
$\texttt{basketFill} \leftarrow \texttt{false}$; $\texttt{basketUpdate} \leftarrow \texttt{false}$;
if $t_q = T$ (* evaluation of $\mathcal{Q}(x^q)$ is complete *)
    if $\mathcal{Q}(x^q) < \mathcal{Q}^I$ and ($I_q = -1$ or $\mathcal{Q}(x^q) \le \mathcal{Q}^{I_q} - \xi(\mathcal{Q}^{I_q} - m^q)$)
        (* make x^q the new incumbent *)
        $I \leftarrow q$; $\mathcal{Q}^I \leftarrow \mathcal{Q}(x^I)$;
        possibly increase Δ_{curr} according to (62) and (63);
        modify the model function by possibly deleting cuts not arising
            from the evaluation of $\mathcal{Q}(x^q)$;
    else
        call Model-Update(k);
        call Reduce-$\Delta(q)$ to update Δ_{curr};
    end (if)
    $\mathcal{B} \leftarrow \mathcal{B} \backslash \{q\}$;
    $\texttt{basketUpdate} \leftarrow \texttt{true}$;
else if $t_q \ge \sigma T$ **and** $|\mathcal{B}| < K$ **and** not $\texttt{speceval}_q$
    (* basket-filling phase: enough partial sums have been evaluated at x^q
        to trigger calculation of a new candidate iterate *)
    $\texttt{speceval}_q \leftarrow \texttt{true}$; $\texttt{basketFill} \leftarrow \texttt{true}$;
end (if)
if $\texttt{basketFill}$ or $\texttt{basketUpdate}$
    $k \leftarrow k + 1$; set $\Delta_k \leftarrow \Delta_{\text{curr}}$; set $I_k \leftarrow I$;
    solve $\texttt{trsub}(x^I, \Delta_k)$ to obtain x^k;
    $m^k \leftarrow m(x^k)$;
    if (64) holds
        STOP;
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{k\}$;
    $\texttt{speceval}_k \leftarrow \texttt{false}$; $t_k \leftarrow 0$;
    $\texttt{evaluate}(x^k, k)$;
end (if)

It is not generally true that the first K iterates $x^0, x^1, \ldots, x^{K-1}$ generated by the algorithm are all basket-filling iterates. Often, an evaluation of some iterate is completed before the basket has filled completely, so a "basket-update" iterate is used to generate a replacement for this point. Since each basket-update iterate does not change the size of the basket, however, the number of basket-filling iterates that are generated in the course of the algorithm is exactly K.

4.2. Analysis of Algorithm ATR

We now analyze Algorithm ATR, showing that its convergence properties are similar to those of Algorithm TR. Throughout, we make the following assumption:

$$\text{Every task is completed after a finite time.} \tag{65}$$

The analysis follows closely that of Algorithm TR presented in Section 3.2. We state the analogues of all the lemmas and theorems from the earlier section, incorporating the changes and redefinitions needed to handle Algorithm ATR. Most of the details of the proofs are omitted, however, since they are similar to those of the earlier results.

We start by defining the level set within which the points and incumbents generated by ATR lie.

Lemma 5. *All incumbents x^I generated by ATR lie in $\mathcal{L}(\mathcal{Q}_{\max})$, whereas all points x^k considered by the algorithm lie in $\mathcal{L}(\mathcal{Q}_{\max}; \Delta_{\rm hi})$, where $\mathcal{L}(\cdot)$ and $\mathcal{L}(\cdot;\cdot)$ are defined by (32) and (33), respectively, and $\mathcal{Q}_{\max}$ is defined by*

$$\mathcal{Q}_{\max} \stackrel{\rm def}{=} \sup\{\mathcal{Q}(x) \,|\, \|x - x^0\| \le \Delta_{\rm hi}\}.$$

Proof. Consider first what happens in ATR before the first function evaluation is complete. Up to this point, all the iterates x^k in the basket are generated in the basket-filling part and therefore satisfy $\|x^k - x^0\| \le \Delta_k \le \Delta_{\rm hi}$, with $\mathcal{Q}^{I_k} = \mathcal{Q}^{-1} = \infty$.

When the first evaluation is completed (by x^k, say), it trivially passes the test to be accepted as the new incumbent. Hence, the first noninfinite incumbent value becomes $\mathcal{Q}^I = \mathcal{Q}(x^k)$, and by definition we have $\mathcal{Q}^I \le \mathcal{Q}_{\max}$. Since all later incumbents must have objective values smaller than this first $\mathcal{Q}^I$, they all must lie in the level set $\mathcal{L}(\mathcal{Q}_{\max})$, proving our first statement.

All points x^k generated within `act_on_completed_task` lie within a distance $\Delta_k \le \Delta_{\rm hi}$ either of x^0 or of one of the later incumbents x^I. Since all the incumbents, including x^0, lie in $\mathcal{L}(\mathcal{Q}_{\max})$, we conclude that the second claim in the theorem is also true.

Analogously with β (34), we define a bound on the subgradients over the set $\mathcal{L}(\mathcal{Q}_{\max}; \Delta_{\rm hi})$ as follows:

$$\bar\beta = \sup\{\|g\|_1 \,|\, g \in \partial\mathcal{Q}(x), \text{ for some } x \in \mathcal{L}(\mathcal{Q}_{\max}; \Delta_{\rm hi})\}. \tag{66}$$

The next result is analogous to Lemma 1. It shows that for any sequence of iterates x^k for which the parent incumbent x^I_k is the same, the optimal objective value in $\texttt{trsub}(x^{I_k}, \Delta_k)$ is monotonically increasing.

Lemma 6. *Consider any contiguous subsequence of iterates x^k, $k = k_1, k_1 + 1, \dots, k_2$ for which the parent incumbent is identical; that is, $I_{k_1} = I_{k_1+1} = \cdots = I_{k_2}$. Then we have*

$$m^{k_1} \le m^{k_1+1} \le \cdots \le m^{k_2}.$$

Proof. We select any $k = k_1, k_1+1, \dots, k_2-1$ and prove that $m^k \le m^{k+1}$. Since x^k and x^{k+1} have the same parent incumbent (x^I, say), no new incumbent has been accepted between the generation of these two iterates, so the wholesale cut deletion that may occur with the adoption of a new incumbent cannot have occurred. There may, however, have been a call to `Model-Update`(k). The first "else if" clause in `Model-Update` would have ensured that cuts active at the solution of $\mathtt{trsub}(x^I, \Delta_k)$ were still present in the model when we solved $\mathtt{trsub}(x^I, \Delta_{k+1})$ to obtain x^{k+1}. Moreover, since no new incumbent was accepted, Δ_{curr} cannot have been increased, and we have $\Delta_{k+1} \le \Delta_k$. We now use the same argument as in the proof of Lemma 1 to deduce that $m^k \le m^{k+1}$.

The following result is analogous to Lemma 2. We omit the proof, which modulo the change in notation is identical to the earlier result.

Lemma 7. *For all $k = 0, 1, 2, \ldots$ such that $I_k \neq -1$, we have that*

$$\mathcal{Q}^{I_k} - m^k \ge \min\left(\Delta_k, \|x^{I_k} - P(x^{I_k})\|_\infty\right) \frac{\mathcal{Q}^{I_k} - \mathcal{Q}^*}{\|x^{I_k} - P(x^{I_k})\|_\infty} \tag{67a}$$

$$\ge \hat{\epsilon} \min\left(\Delta_k, \|x^{I_k} - P(x^{I_k})\|_\infty\right), \tag{67b}$$

where $\hat{\epsilon} > 0$ is defined in (10).

The following analogue of Lemma 3 requires a slight redefinition of the quantity E_k from (39). We now define it to be the closest approach by an *incumbent* to the solution set, up to and including iteration k; that is,

$$E_k \stackrel{\mathrm{def}}{=} \min_{\bar{k}=0,1,\ldots,k;\, I_{\bar{k}} \neq -1} \|x^{I_{\bar{k}}} - P(x^{I_{\bar{k}}})\|_\infty. \tag{68}$$

We also omit the proof of the following result, which, allowing for the change of notation, is almost identical to that of Lemma 3.

Lemma 8. *There is a constant $\Delta_{\mathrm{lo}} > 0$ such that for all trust regions Δ_k used in the course of Algorithm ATR, we have*

$$\Delta_k \ge \min(\Delta_{\mathrm{lo}}, E_k/4).$$

The value of Δ_{lo} that works in this case is $\Delta_{\mathrm{lo}} = (1/4)\min(1, \hat{\epsilon}/\bar{\beta}, \Delta_{\mathrm{hi}})$, where $\bar{\beta}$ comes from (66).

There is also an analogue of Lemma 4 that shows that if the incumbent remains the same for a number of consecutive iterations, the gap between incumbent objective value and model function decreases significantly as the iterations proceed.

Lemma 9. *Let $\epsilon_{\mathrm{tol}} = 0$ in Algorithm ATR, and let $\bar{\eta}$ be any constant satisfying $0 < \bar{\eta} < 1$, $\bar{\eta} > \xi$, $\bar{\eta} \ge \eta$. Choosing any index k_1 with $I_{k_1} \neq -1$, we have either that the incumbent $I_{k_1} = I$ is eventually replaced by a new incumbent or that there is an iteration $k_2 > k_1$ such that*

$$\mathcal{Q}^I - m^{k_2} \le \bar{\eta}\left[\mathcal{Q}^I - m^{k_1}\right]. \tag{69}$$

The proof of this result follows closely that of its antecedent Lemma 4. The key is in the construction of the Model-Update procedure. As long as

$$\mathcal{Q}^I - m^k > \eta[\mathcal{Q}^I - m^{k_1}], \quad \text{for } k \ge k_1, \text{ where } I = I_{k_1} = I_k, \tag{70}$$

none of the cuts generated during the evaluation of $\mathcal{Q}(x^q)$ for any $q = k_1, k_1 + 1, \ldots, k$ can be deleted. The proof technique of Lemma 4 can then be used to show that the successive iterates $x^{k_1}, x^{k_1+1}, \ldots$ cannot be too closely spaced if the condition (70) is to hold and if all of them fail to satisfy the test to become a new incumbent. Since they all belong to a box of finite size centered on x^I, there can be only finitely many of these iterates. Hence, either a new incumbent is adopted at some iteration $k \geq k_1$ or condition (69) is eventually satisfied.

We now show that the algorithm cannot "get stuck" at a nonoptimal incumbent. The following result is analogous to Theorem 1, and its proof relies on the earlier results in exactly the same way.

Theorem 4. *Suppose that* $\epsilon_{\mathrm{tol}} = 0$.

(i) If $x^I \notin \mathcal{S}$, *then this incumbent is replaced by a new incumbent after a finite time.*
(ii) If $x^I \in \mathcal{S}$, *then either Algorithm ATR terminates (and verifies that* $x^I \in \mathcal{S}$*), or* $Q^I - m^k \downarrow 0$ *as* $k \to \infty$.

We conclude with the result that shows convergence of the sequence of incumbents to $\mathcal{S}$. Once again, the logic of proof follows that of the synchronous analogue Theorem 2.

Theorem 5. *Suppose that* $\epsilon_{\mathrm{tol}} = 0$. *The sequence of incumbents* $\{x^{I_k}\}_{k=0,1,2,\ldots}$ *is either finite, terminating at some* $x^I \in \mathcal{S}$ *or is infinite with the property that* $\|x^{I_k} - P(x^{I_k})\|_\infty \to 0$.

5. Implementation on Computational Grids

We now describe some salient properties of the computational environment in which we implemented the algorithms, namely, a computational grid running the Condor system and the MW runtime support library.

5.1. Properties of Grids

The term "grid computing" (synonymously "metacomputing") is generally used to describe parallel computations on a geographically distributed, heterogeneous computing platform. Within this framework there are several variants of the concept. The one of interest here is a parallel platform made up of shared workstations, nodes of PC clusters, and supercomputers. Although such platforms are potentially powerful and inexpensive, they are difficult to harness for productive use, for the following reasons:

- Poor communications properties. Latencies between the processors may be high, variable, and unpredictable.
- Unreliability. Resources may disappear without notice. A workstation performing part of our computation may be reclaimed by its owner and our job terminated.

- Dynamic availability. The pool of available processors grows and shrinks during the computation, according to the claims of other users and scheduling considerations at some of the nodes.
- Heterogeneity. Resources may vary in their operational characteristics (memory, swap space, processor speed, operating system).

In all these respects, our target platform differs from conventional multiprocessor platforms (such as IBM SP or SGI Origin machines) and from Linux clusters.

5.2. Condor

Our particular interest is in grid computing platforms based on the Condor system [17], which manages distributively owned collections ("pools") of processors of different types, including workstations, nodes from PC clusters, and nodes from conventional multiprocessor platforms. When a user submits a job, the Condor system discovers a suitable processor for the job in the pool, transfers the executable and starts the job on that processor. It traps system calls (such as input/output operations), referring them back to the submitting workstation, and checkpoints the state of the job periodically. It also migrates the job to a different processor in the pool if the current host becomes unavailable for any reason (for example, if the workstation is reclaimed by its owner). Condor managed processes can communicate through a Condor-enabled version of PVM [10] or by using Condor's I/O trapping to write into and read from a series of shared files.

5.3. Implementation in MW

MW (see Goux, Linderoth, and Yoder [13] and Goux et al. [12]) is a runtime support library that facilitates implementation of parallel master-worker applications on computational grids. To implement MW on a particular computational grid, a grid programmer must reimplement a small number of functions to perform basic operations for communications between processors and management of computational resources. These functions are encapsulated in the MWRMComm class. Of more relevance to the current paper is the other side of MW, the application programming interface presented to the application programmer. This interface takes the form of a set of three C++ abstract classes that must be reimplemented in a way that describes the particular application. These classes, named MWDriver, MWTask, and MWWorker, contain a total of ten methods for which the user must supply implementations. We describe these methods briefly, indicating how they are implemented for the particular case of the ATR and ALS algorithms.

MWDriver. This class is made up of methods that execute on the submitting workstation, which acts as the master processor. It contains the following four C++ pure virtual functions. (Naturally, other methods can be defined as needed to implement parts of the algorithm.)

- `get_userinfo`: Processes command-line arguments and does basic setup. In our applications this function reads a command file to set various parameters, including convergence tolerances, number of scenarios, number of partial sums to be evaluated in each task, maximum number of worker processors to be requested, initial trust region radius, and so on. It calls the routines that read and store the problem data files, and it reads the initial point, if one is supplied. It also performs the operations specified in the `initialization` routine of Algorithms ALS and ATR, except for the final `evaluate` operation, which is handled by the next function.
- `setup_initial_tasks`: Defines the initial pool of tasks. In the case of Algorithms ALS and ATR, this function corresponds to a call to `evaluate` at x^0.
- `pack_worker_init_data`: Packs the initial data to be sent to each worker processor when it joins the pool. In our case, the information contained in the input files for the stochastic programming problem is sent to each worker. When the worker subsequently receives a task requiring it to solve a number of second-stage scenarios, it can use the original input data to generate the particular data for its assigned set of scenarios. By loading each new worker with the problem data, we avoid having to subsequently pass a complete set of data for every scenario in every task.
- `act_on_completed_task`: Is called every time a task finishes, to process the results of the task and to take any actions arising from these results. See Algorithms ALS and ATR for our definition of this function in our applications.

The MWDriver base class performs many other operations associated with handling worker processes that join and leave the computation, assigning tasks to appropriate workers, rescheduling tasks when their host workers disappear without warning, and keeping track of performance data for the run. All this complexity is hidden from the application programmer.

MWTask. The MWTask is the abstraction of a single task. It holds both the data describing that task and the results obtained by executing the task. The user must implement four functions for packing and unpacking this data and results between master and workers into simple data structures that can be communicated between master and workers using the appropriate primitives for the particular computational grid platform on which MW is implemented. In most of the results reported in Section 6, the message-passing facilities of Condor-PVM were used to perform the communication. By simply changing compiler directives, the same algorithmic code can also be implemented on an alternative communication protocol that uses shared files to pass messages between master and workers. The large run reported in the next section used this version of the code.

In our applications, each task evaluates the partial sum $\mathcal{Q}_{[j]}(x)$ and a subgradient for a given number of clusters. The task is described by a range of scenario indices for each cluster in the task and by a value of the first-stage variables x.

The results consist of the function and subgradient for each of the clusters in the task.

MWWorker. The MWWorker class is the core of the executable that runs on each worker. The user must implement two pure virtual functions:

- `unpack_init_data`: Unpacks the initial information passed to the worker by the MWDriver function `pack_worker_init_data()` when the worker joins the pool. (See the discussion of `pack_worker_init_data` in the MWDriver class.)
- `execute_task`: Executes a single task.

After initializing itself, using the information passed to it by the master, the worker process sits in a loop, waiting for tasks to be sent to it. When it detects a new task, it calls `execute_task` to compute the results. It passes the results back to the worker by using the appropriate function from the MWTask class, and then returns to its wait loop. The wait loop terminates when the master sends a termination message. In our applications, the `execute_task()` function formulates the second-stage linear programs in its clusters by using the information in the task definition and the data passed to the worker on initialization. It then calls the linear programming solvers SOPLEX or CPLEX to solve these linear programs, and uses the dual solutions to calculate the subgradient for each cluster.

6. Computational Results

We now report on computational experiments obtained with implementations of the ALS, TR, and ATR algorithms using MW on the Condor system. After describing some further details of the implementations and the experiments, we discuss our choices for the various algorithmic parameters and how these were varied between runs. We then tabulate and discuss the results.

6.1. Implementations and Experiments

As noted earlier, we used the Condor-PVM implementation of MW for most of the the runs reported here. Most of the computational time is taken up with solving linear programming problems, both by the master process (in the `act_on_completed_task` function) and in the tasks, which solve clusters of second-stage linear programs. We used the CPLEX simplex solver on the master processor and the SOPLEX public-domain simplex code (see Wunderling [26]) on the workers. SOPLEX is somewhat slower in general, but since most of the machines in the Condor pool do not have CPLEX licenses, there was little alternative but to use a public-domain code.

We ran most of our experiments on the Condor pool at the University of Wisconsin, sometimes using Condor's flocking mechanism to augment this pool with processors from other sites. The other sites included the University of New Mexico, Columbia University, and the Linux cluster Chiba City at Argonne National Laboratory. The architectures included PCs running Linux, and PCs and Sun workstations running different versions of Solaris. The number of workers available for our use varied dramatically between and during each set of trials, because of the differing priorities of the two accounts we used, the variation of our priority during each run, the number and priorities of other users of the Condor pool at the time, and the varying number of machines available to the pool. The latter number tends to be larger during the night, when owners of the individual workstations are less likely to be using them. The master process was run on a Linux machine in some experiments and an Intel Solaris machine in other cases.

The input files for the problems reported here were in SMPS format (see Birge et al. [3] and Gassmann and Schweitzer [9]). We considered two-stage stochastic linear programs in which the number of scenarios is finite but extremely large. We used Monte Carlo sampling to obtain approximate problems with a specified number N of second-stage scenarios. Brief descriptions of the test problems can be found at [15]. In each experiment, we supplied a starting point to the code, obtained from the solution of a different sampled instance of the same problem. The function value of the starting point was therefore quite close to the optimal objective value.

6.2. Critical Parameters

As part of the initialization procedure (implemented by the `get_userinfo` function in the MWDriver class), the code reads an input file in which various parameters are specified. Several parameters, such as those associated with modifying the size of the trust region, have fixed values that we have discussed already in the text. Others are assigned the same values for all algorithms and all experiments, namely,

$$\epsilon_{\mathrm{tol}} = 10^{-5}, \quad \Delta_{\mathrm{hi}} = 10^3, \quad \Delta_{0,0} = \Delta_0 = 1, \quad \xi = 10^{-4}.$$

We also set $\eta = 0$ in the Model-Update functions in both TR and ATR. In TR, this choice has the effect of not allowing deletion of cuts generated during any major iterations, until a new major iterate is accepted. In ATR, the effect is to not allow deletion of cuts that are generated at points whose parent incumbent is still the incumbent. Even among cuts for which `possible_delete` is still true at the final conditional statement of the Model-Update procedures, we do not actually delete the cuts until they have been inactive at the solution of the trust-region subproblem for a specified number of consecutive iterations. For TR, we delete the cut if it has been inactive for more than 100 consecutive minor iterations, while in ATR we delete the cut if it was last active at subproblem ℓ, where $\ell < k - 100$ and k is the current iteration index. Our cut deletion strategy is therefore not at all parsimonious; it tends to lead to subproblems (20) and (59) with fairly large numbers of cuts. In most cases, however, the storage required for these cuts and the time required to solve the subproblems remain reasonable. We discuss the exceptions below.

The synchronicity parameter σ, which arises in Algorithms ALS and ATR and which specifies the proportion of clusters from a particular point that must be evaluated in order to trigger evaluation of a new candidate solution, is varied between .5 and 1.0 in our experiments. The size K of the basket $\mathcal{B}$ is varied between 1 and 14. For each problem, the number T of clusters is also varied in a manner described in the tables, as is the number of tasks into which the second-stage calculations are divided, which we denote by C. Note that the number of second-stage LPs per chunk is therefore N/C while the number per cluster is N/T.

The MW library allows us to specify an upper bound on the number of workers we request from the Condor pool, so that we can avoid claiming more workers than we can utilize effectively. We calculate a rough estimate of this number based on the number of tasks C per evaluation of $\mathcal{Q}(x)$ and the basket size K. For instance, the synchronous TR and LS algorithms can never use more than C worker processors, since they evaluate $\mathcal{Q}$ at just one x at a time. In the case of TR and ATR, we request $\mathrm{mid}(25, 200, \lfloor (K+1)C/2 \rfloor)$ workers. For ALS, we request $\mathrm{mid}(25, 200, 2C)$ workers.

We have a single code that implements all four algorithms LS, ALS, TR, and ATR, using logical branches within the code to distinguish between the L-shaped and trust-region variants. There is no distinction in the code between the two synchronous variants and their asynchronous counterparts. Instead, by setting $\sigma = 1.0$, we force synchronicity by ensuring that the algorithm considers only one value of x at a time.

Whenever a worker processor joins the computation, MW sends it a benchmark task that typifies the type of task it will receive during the run. In our case, we define the benchmark task to be the solution of N/C second-stage LPs. The time required for the processor to solve this task is logged, and we set the ordering policy so as to ensure that when more than one worker is available to process a particular task, the task is sent to the worker that logged the fastest time on the benchmark task.

6.3. Results: Varying Parameter Choices

In this section we describe a series of experiments on the same problem, using different parameter settings, and run under different conditions on the Condor pool. For these trials, we use the problem SSN, which arises from a network design application described by Sen, Doverspike, and Cosares [23]. This problem is based on a graph with 89 arcs, each representing a telecommunications link between two cities. The first-stage variables represent the (nonnegative) extra capacity to be added to each of these 89 arcs to meet an uncertain demand pattern. There is a constraint on the total added capacity. The demands consist of requests for service between pairs of nodes in the graph. For each set of requests, a route through the network of sufficient capacity to meet the requests must be found, otherwise a penalty term for each request that cannot be satisfied is added to the objective. The second-stage problems are network flow problems for calculating the routing for a given set of demand flows. Each such problem is nontrivial: 706 variables, 175 constraints, and 2284 nonzeros in the constraint matrix. The uncertainty lies in the fact that the demand for service on each of the 86 pairs is not known exactly. Rather, there are three to seven possible scenarios for these demands, all independent of each other, giving a total of about 10^{70} possible scenarios. We use Monte Carlo sampling to obtain a sampled approximation with $N = 10,000$ scenarios. The deterministic equivalent for this sampled approximation has approximately 1.75×10^6 constraints and 7.06×10^6 variables. In all the runs, we used as starting point the computed solution for a different sampled approximation—one with $20,000$ scenarios and a different random seed. The starting point had a function value of approximately 9.868860, whereas the optimal objective was approximately 9.832544.

In the tables below we list the following information.

- **points evaluated**. The number of distinct values of the first-stage variables x generated by solving the master subproblem—the problem (18) for Algorithm ALS, (20) for Algorithm TR, and (59) for Algorithm ATR.
- $|\mathcal{B}|$. Maximum size of the basket, also denoted above by K.
- **number of tasks (chunks)**. Denoted above by C.
- **number of clusters**. Denoted above by T, the number of partial sums (11) into which the second-stage problems are divided.
- **max processors**. The number of workers requested.
- **average processors**. The average of the number of active (nonsuspended) worker processors available for use by our problem during the run. Because of the dynamic nature of the Condor system, the actual number of available processors fluctuates continually during the run.
- **parallel efficiency**. The proportion of time for which worker processors were kept busy solving second-stage problems while they were owned by this run.
- **maximum number of cuts in the model**. The maximum number of (partial) subgradients that are used to define the model function during the course of the algorithm.
- **masterproblem solve time**. The total time spent solving the master subproblem to generate new candidate iterates during the course of the algorithm.
- **wall clock**. The total time (in minutes) between submission of the job and termination.

Table 1 shows the results of a series of trials of Algorithm ALS with three different values of σ (.5, .7, and .85) and three different choices for the number of chunks C into which the second-stage solutions were divided (10, 25, and 50). The number of clusters T was fixed at 50, so that up to 50 cuts were generated at each iteration. For $\sigma = .5$, the number of values of x for which second-stage evaluations are occurring at any point in time ranged from 2 to 4 during the runs, while for $\sigma = .85$, there were never more than 2 points being evaluated simultaneously.

run	points evaluated	σ	# tasks (C)	# clusters (T)	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	masterproblem solve time (min)	wall clock time (min)
ALS	269	.5	10	50	20	15	.74	5491	26	368
ALS	275	.5	25	50	50	21	.90	5536	25	270
ALS	293	.5	50	50	100	20	.83	5639	27	329
ALS	270	.7	10	50	20	12	.79	5522	27	509
ALS	274	.7	25	50	50	25	.73	5550	25	281
ALS	282	.7	50	50	100	26	.81	5562	24	254
ALS	254	.85	10	50	20	12	.58	5496	22	575
ALS	276	.85	25	50	50	19	.57	5575	23	516
ALS	278	.85	50	50	100	35	.49	5498	25	260

Table 1. SSN, with $N = 10,000$ scenarios, Algorithm ALS.

When these runs were performed, we were not able to obtain anything approaching the requested number $2C$ of workers from the Condor pool. As general trends, we see that the less synchronous variants (with $\sigma = .5$ and $\sigma = .7$) tend to be faster than the more synchronous variant (with $\sigma = .85$), except for the final run, during which more processors were available. Moreover, larger values of C also tend to produce faster runs. We also note that the number of iterations does not depend strongly on σ. We would not, of course, expect C to affect strongly the number of iterations, but since it affects the manner in which the second-stage evaluation work is distributed, we *would* expect it to affect the run time. Since the number of workers available to us during this run was limited, however, we did not see the full benefit of a finer-grained work distribution ($C = 50$), though the relatively low parallel efficiency of the final run ($\sigma = .85$, $C = 50$) indicates that the benefits of more processors may not have been great in any case.

A note on typical task sizes: For $C = 10$, a typical task required about 50-280 seconds on a typical worker machine available to us, while for $C = 50$, about 9-60 seconds were required. The large variation reflects the wide range in processing ability of the machines available in a pool during a typical run. These numbers also generally hold for the results in Tables 2 and 3.

By comparing the results from Table 1 with those reported in Tables 2 and 3, we verified that Algorithm ALS was not as efficient on this problem as Algorithm TR and certain variants of Algorithm ATR. One advantage, however, was that the asymptotic convergence of ALS was quite fast. Having taken many iterations to build up a model and return to a neighborhood of the solution after having strayed far from it in early iterations, the last three to four iterations home in rapidly from a relatively crude approximate solution (a relative accuracy $(\mathcal{Q}_{\min} - m(x^{k+1}))/(1+|\mathcal{Q}_{\min}|)$ of between .0006 and .0026) to a solution of high accuracy.

We now turn to Tables 2 and 3, which report on two sets of trials on the same problem as in Table 1. In these trials we varied the following parameters:

run	points evaluated	$\|\mathcal{B}\|$ (K)	# tasks (C)	# clusters (T)	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	masterproblem solve time (min)	wall clock time (min)
TR	48	-	10	100	20	19	.21	4284	3	131
TR	72	-	10	50	20	19	.26	3520	3	150
TR	39	-	25	100	25	22	.49	3126	2	59
TR	75	-	25	50	25	23	.48	3519	3	114
TR	43	-	50	100	50	42	.52	3860	3	35
TR	61	-	50	50	50	44	.53	3011	3	40
ATR	109	3	10	100	20	18	.74	7680	9	107
ATR	121	3	10	50	20	19	.66	4825	6	111
ATR	105	3	25	100	50	37	.73	7367	8	49
ATR	113	3	25	50	50	41	.60	4997	6	48
ATR	103	3	50	100	100	66	.55	7032	9	29
ATR	129	3	50	50	100	66	.59	5183	7	32
ATR	167	6	10	100	35	24	.93	7848	13	99
ATR	209	6	10	50	35	22	.89	5730	15	92
ATR	186	6	25	100	87	49	.77	8220	14	53
ATR	172	6	25	50	87	49	.80	5945	7	49
ATR	159	6	50	100	175	31	.89	7092	11	65
ATR	213	6	50	50	175	40	.88	6299	12	70
ATR	260	9	10	100	50	12	.95	14431	35	267
ATR	286	9	10	50	50	23	.90	6528	19	160
ATR	293	9	25	100	125	17	.93	9911	30	232
ATR	377	9	25	50	125	15	.96	7080	24	321
ATR	218	9	50	100	200	28	.82	10075	25	101
ATR	356	9	50	50	200	23	.93	6132	23	194
ATR	378	14	10	100	75	18	.88	15213	77	302
ATR	683	14	10	50	75	14	.98	8850	48	648
ATR	441	14	25	100	187	22	.89	14597	61	312
ATR	480	14	25	50	187	20	.94	8379	36	347
ATR	446	14	50	100	200	20	.83	13956	64	331
ATR	498	14	50	50	200	22	.94	7892	35	329

Table 2. SSN, with $N = 10{,}000$ scenarios, first trial, Algorithms TR and ATR.

- **basket size:** $K = 1$ (synchronous TR) as well as $K = 3, 6, 9, 14$;
- **number of tasks:** $C = 10, 25, 50$, as in Table 1;
- **number of clusters:** $T = 50, 100$.

The parameter σ was fixed at .7 in all these runs.

The results in Table 2 were obtained with the master processor running on an Intel Solaris machine, while Table 3 was obtained with a Linux master. In both cases, the Condor pool that we tapped for worker processors was identical. Therefore, it is possible to do a meaningful comparison between each line of Table 3 and its counterpart in Table 2. Conditions on the Condor pool varied between and during each trial. This fact, combined with the properties of the algorithm, resulted in large variability of runtime from one trial to the next, as we discuss below.

run	points evaluated	$\vert\mathcal{B}\vert$ (K)	# tasks (C)	# clusters (T)	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	masterproblem solve time (min)	wall clock time (min)
TR	47	-	10	100	20	17	.24	3849	4	192
TR	67	-	10	50	20	13	.34	3355	3	256
TR	47	-	25	100	25	18	.49	3876	4	97
TR	57	-	25	50	25	18	.40	2835	3	119
TR	42	-	50	100	50	30	.22	3732	3	122
TR	65	-	50	50	50	31	.25	3128	4	151
ATR	92	3	10	100	20	11	.89	7828	9	125
ATR	98	3	10	50	20	11	.84	4893	5	173
ATR	86	3	25	100	50	34	.38	6145	5	70
ATR	95	3	25	50	50	32	.41	4469	4	77
ATR	80	3	50	100	100	52	.23	5411	5	80
ATR	131	3	50	50	100	59	.47	4717	6	55
ATR	137	6	10	100	35	30	.57	8338	12	84
ATR	200	6	10	50	35	26	.60	5211	9	130
ATR	119	6	25	100	87	52	.55	7181	7	44
ATR	199	6	25	50	87	58	.48	5298	9	81
ATR	178	6	50	100	175	50	.47	9776	15	77
ATR	240	6	50	50	175	61	.64	5910	11	74
ATR	181	9	10	100	50	37	.56	8737	15	96
ATR	289	9	10	50	50	19	.93	7491	25	238
ATR	212	9	25	100	125	90	.66	11017	21	45
ATR	272	9	25	50	125	65	.45	6365	15	105
ATR	281	9	50	100	200	51	.72	11216	34	88
ATR	299	9	50	50	200	26	.83	7438	27	225
ATR	304	14	10	100	75	38	.89	13608	43	129
ATR	432	14	10	50	75	42	.95	7844	28	132
ATR	356	14	25	100	187	71	.78	13332	48	111
ATR	444	14	25	50	187	45	.89	7435	36	163
ATR	388	14	50	100	200	42	.79	12302	52	192
ATR	626	14	50	50	200	48	.81	7273	46	254

Table 3. SSN, with $N = 10,000$ scenarios, second trial, Algorithms TR and ATR.

The nondeterministic nature of the algorithms is evident in doing a side-by-side comparison of the two tables. Even for synchronous TR, the slightly different numerical values for function and subgradient value returned by different workers in different runs results in slight variations in the iteration sequence and therefore slight differences in the number of iterations. For the asynchronous Algorithm ATR, the nondeterminism is even more marked. During the basket-filling phase of the algorithm, computation of a new x is triggered when a certain proportion of tasks from a current value of x has been returned. On different runs, the tasks will be returned in different orders, so the information used by the trust-region subproblem (59) in generating the new point will vary from run to run, and the resulting iteration sequences will generally show substantial differences.

The synchronous TR algorithm is clearly better than the ATR variants with $K > 1$ in terms of total computation, which is roughly proportional to the number of iterations. In fact, the total amount of work increases steadily with basket size. Because of the decreased synchronicity requirements and the greater parallelism obtained for $K > 1$, the wall clock times (last columns) do not follow quite the same trend. The wall clock times for basket sizes $K = 3$ and $K = 6$ are at least competitive with the results obtained for the synchronous TR algorithm. The choice $K = 6$ gave few of the fastest runs but did yield consistent performance over all the different choices for the other parameters, and under different Condor pool conditions.

The deleterious effects of synchronicity in Algorithm TR can be seen in its poor performance on several instances, particularly during the second trial. Let us compare, for instance, the entries in the two tables for the variant of TR with $C = 50$ and $T = 100$. In the first trial, this run used 42 worker processors on average and took 35 minutes, while in the second trial it used 30 workers on average and required 122 minutes. The difference in runtime is too large to be accounted for by the number of workers. Because this is a synchronous algorithm, the time required for each iteration is determined by the time required for the slowest worker to return the results of its task. In the first trial, almost all tasks required between 6 and 35 seconds, except for a few iterations that contained tasks that took up to 62 seconds. In the second trial, the slowest worker at each iteration almost always required more than 60 seconds to complete its task. We return to this point in discussing Table 4 below.

Other general observations we can make are that 100 clusters give almost uniformly better results in terms of wall clock time than 50 clusters, although the higher number results in a larger number of cuts in the trust-region subproblems and an increased amount of time on the master processor in solving these problems. The latter factor is critical for $K = 9$ and $K = 14$, which do not compare favorably with the smaller values of K on this problem, even if many more worker processors are available. For the large basket sizes, the loss of control induced by the increase in assynchronicity leads to a significantly larger number of points that are evaluated.

In all cases, it takes some time for the model m to become a good enough approximation to $\mathcal{Q}$ that it generates a step that meets the trust-region acceptance criteria. The six TR runs in Table 3, for instance, required 18, 27, 16, 22, 16, and 26 trust-region subproblems to be solved, respectively, before they stepped away from the initial point. (Note that, as expected, the runs with $T = 100$ required fewer such iterations than those with $T = 50$.) After the first step is taken, most steps are successful; that is, the first minor iterate usually is accepted as the next major iterate. Occasionally, two to four minor iterations are required before the next major iteration is identified. Similar behavior is observed for the runs of ATR, except that successful iterations are more widely spaced. For the first run with $K = 6$ in Table 3, for instance, the 37th solution of (59) yields the first successful step; then 36 of the following 99 solutions of the subproblem yield successful steps.

In Table 4, we took the most promising parameter combinations from Tables 3 and 2 and ran three trials with each combination. The Condor pool conditions varied widely during this trial, as can be seen by the way that the average

run	points evaluated	$\|\mathcal{B}\|$ (K)	# tasks (C)	# clusters (T)	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	masterproblem solve time (min)	wall clock time (min)
TR	47	-	25	100	25	23	.49	4040	3	58
TR	44	-	25	100	25	21	.31	3220	3	97
TR	45	-	25	100	25	20	.23	3966	4	158
TR	51	-	50	100	50	37	.33	4428	3	48
TR	51	-	50	100	50	45	.14	4806	3	135
TR	46	-	50	100	50	41	.15	3847	4	135
ATR	81	3	25	100	50	43	.38	7451	6	64
ATR	81	3	25	100	50	39	.41	6461	5	64
ATR	87	3	25	100	50	36	.44	6055	8	66
ATR	106	3	50	100	100	84	.28	8222	9	53
ATR	95	3	50	100	100	65	.26	6786	7	64
ATR	94	3	50	100	100	23	.44	6593	8	105
ATR	171	6	25	100	87	70	.45	9173	19	61
ATR	135	6	25	100	87	61	.39	7354	12	75
ATR	145	6	25	100	87	38	.35	8919	16	146
ATR	177	6	50	100	175	87	.41	9263	22	54
ATR	162	6	50	100	175	93	.34	7832	18	66
ATR	159	6	50	100	175	39	.27	8215	22	199

Table 4. SSN final trial with best parameter combinations, $N = 10,000$ scenarios, Algorithms TR and ATR.

number of workers varies within each group of three runs. For the asynchronous ATR runs, the differences in wall clock times within each set of three runs usually can be explained in terms of the varying number of workers available. (A possible exception is the last line of the table, the third run of ATR with $K = 6$, $C = 50$ and $T = 100$, which took almost four times as long as the first run while having only slightly fewer than half as many processors. While the speed of machines available was roughly similar between these runs, the third run was plagued with numerous suspensions as the workers were reclaimed by their owners. Total time that workers were suspended was over 23,000 seconds on the third run and less than 2,800 seconds during the first run.) On the other hand, the variability in wall clock time between the six runs of the synchronous TR algorithm was due not to the number of available workers but rather to the synchronicity effect described above. In the run reported in the first line of the table, for instance, the slowest worker on any iteration typically took less than 65 seconds. In the run reported on the third line, the time required by the slowest worker varied significantly but was typically much longer, 150 seconds and more.

6.4. Larger Instances

We also performed runs on several larger instances of SSN (with $N = 100,000$ scenarios) and on some very large instances of the stormG2 problem, a cargo

run	points evaluated	$\|\mathcal{B}\|$ (K)	# tasks (C)	# clusters (T)	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	masterproblem solve time (min)	wall clock time (min)
ATR	177	3	100	100	200	38	.52	10558	47	1357

Table 5. SSN, with $N = 100,000$ scenarios.

run	points evaluated	$\|\mathcal{B}\|$ (K)	# tasks (C)	# clusters (T)	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	masterproblem solve time (min)	wall clock time (min)
TR	17	-	125	125	250	106	.55	2310	0.5	146
ATR	25	3	125	125	250	106	.90	3292	0.5	116

Table 6. stormG2, with $N = 250000$ scenarios.

flight scheduling application described by Mulvey and Ruszczyński [19]. Our interest in this section is more in the sheer size of the problems that can be solved using the algorithms developed for the computational grid than with the relative performance of the algorithms with different parameter settings.

Table 5 shows results for a sampled instance of SSN with $N = 100,000$ scenarios, which is a linear program with approximately 1.75×10^7 constraints and 7.06×10^7 variables. This run was performed at a time when not many machines were available, and many suspensions occurred during the run. We chose $T = 100$ chunks per evaluation and found that most tasks required between 41 and 300 seconds on the workers, with a few task times of more than 500 seconds. (The benchmarks indicated that the worker speed varied over a factor of 7.) A total of 77 different workers were used during the run, though the average number of nonsuspended workers available at any time was only 39. In fact, at any given point in the computation there were an average of 7 workers assigned to this task that were suspended. Still, a result was obtained in about 22 hours.

In the stormG2 problem of Mulvey and Ruszczyński [19], the first-stage problem contained 121 variables, while each second-stage problem contained 1259 variables. We considered first a sampled approximation of this problem with 250000 scenarios, which resulted in a linear program with 1.32×10^8 constraints and 315×10^8 unknowns. Results are shown in Table 6. The algorithm was started at a solution of a sampled instance with fewer scenarios and was quite close to optimal. The objective function at the initial point was approximately 15499595.1, compared with an optimal value of 15499591.9 achieved by Algorithm TR. In fact, the TR algorithm takes only one major iteration—it accepts

run	points evaluated	$\lvert\mathcal{B}\rvert$ (K)	# tasks (C)	# clusters $(\mathcal{T})$	max. processors allowed	av. processors	parallel efficiency	max. # cuts in model	masterproblem solve time (hr)	wall clock time (hr)
ATR	28	4	1024	1024	800	433	.668	39647	1.9	31.9

Table 7. stormG2, with $N = 10^7$ scenarios.

the 16th minor iteration as the first major iterate x^1. The ATR variant does not take even one step—it terminates after determining that the initial point x^0 is optimal to within the given convergence tolerance. Although we requested 250 processors, an average of only 106 were available during the time that we performed these two test runs. The second run is able to utilize these to high efficiency, as the second-stage workload can be divided into a large number of chunks and very little time is spent in solving the trust-region subproblem.

Finally, we report on a very large sampled instance of stormG2 with $N = 10^7$ scenarios, an instance whose deterministic equivalent is a linear program with 9.85×10^8 constraints and 1.26×10^{10} variables. Performance is profiled in Table 7.

We used the tighter convergence tolerance $\epsilon_{\rm tol} = 10^{-6}$ for this run. The algorithm took successful steps at iterations 28, 34, 37, and 38, the last of these being the final iteration. The first evaluated point had a function value of 15526740, compared with a value of 15498842 at the final iteration.

For this run, we augmented the Wisconsin Computer Science Condor pool with machines from Georgia Tech, the University of New Mexico, the Italian National Institute of Physics (INFN), the NCSA at the University of Illinois, and the IEOR Department at Columbia, the Albu, and the Wisconsin engineering Department. Table 8 shows the number and type of processors available at each of these locations. In contrast to the other runs reported here, we used the "MW-files" implementation of MW, the variant that uses shared files to perform communication between master and workers rather than Condor-PVM.

The job ran for a total of almost 32 hours. The number of workers being used during the course of the run is shown in Figure 1. The job was stopped after approximately 8 hours and was restarted manually from a checkpoint about 2 hours later. It then ran for approximately 24 hours to completion. The number of workers dopped off significantly on two occasions. The drops were due to the master processor "blocking" to solve a difficult master problem and to checkpoint the state of the computation. During this time the worker processors were idle, and MW decided to release a number of the processors rather than have them sit idle.

As noted in Table 7, an average of 433 workers were present at any given point in the run. The computation used a maximum of 556 workers, and there was a ratio of 12 in the speed of the slowest and fastest machines, as determined by the benchmarks. A total of 40837 tasks were generated during the run, representing

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		

Table 8. Machines available for stormG2, with $N = 10^7$ scenarios.

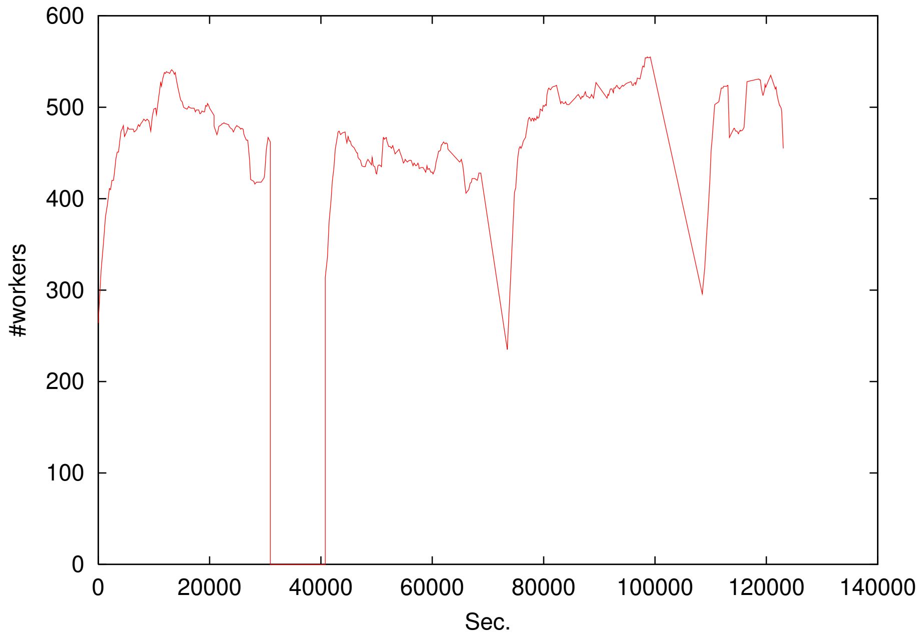

Fig. 1. Number of workers used for stormG2, with $N = 10^7$ scenarios.

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 just over one year of computation.

7. Conclusions

We have described L-shaped and trust-region algorithms for solving the two-stage stochastic linear programming problem with recourse, and derived asynchronous variants suitable for parallel implementation on distributed heterogeneous computational grids. We prove convergence results for the trust-region algorithms. Implementations based on the MW library and the Condor system are described, and we report on computational studies using different algorithmic parameters under different pool conditions. Becasue of the dynamic nature of the computational pool, it is impossible to arrive at a "best" configuration or set of algorithmic parameters for all instances. Instead, it may be important to adjust the algorithm parameters dynamically; we suggest this as a line of future research. Finally, we report on the solution of some large sampled instances of problems from the literature, including an instance of the stormG2 problem whose deterministic equivalent has more than 10^{10} unknowns. Since the use of the computational grid has the greatest benefit on problems that require large amounts of computation, the algorithms developed here are best suited to larger (multistage) problems or incorporated into a sample average approximation approach (see Shapiro and Homem-de-Mello [24].

Acknowledgments

This research was supported by the Mathematics, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. We also acknowledge the support of the National Science Foundation, under Grant CDA-9726385. We would also like to acknowledge the IHPCL at Georgia Tech, which is supported by a grant from Intel; the National Computational Science Alliance under grant number MCA00N015N for providing resources at the University of Wisconsin, the NCSA SGI/CRAY Origin2000, and the University of New Mexico/Albuquerque High Performance Computing Center AltaCluster; and the Italian Istituto Nazionale di Fisica Nucleare (INFN) and Columbia University for allowing us access to their Condor pools.

We are grateful to Alexander Shapiro and Sven Leyffer for discussions about the algorithms presented here.

References

1. O. Bahn, O. du Merle, J.-L. Goffin, and J. P. Vial. A cutting-plane method from analytic centers for stochastic programming. *Mathematical Programming, Series B*, 69:45–73, 1995.
2. J. F. Benders. Partitioning procedures for solving mixed variable programming problems. *Numerische Mathematik*, 4:238–252, 1962.
3. J. R. Birge, M. A. H. Dempster, H. I. Gassmann, E. A. Gunn, and A. J. King. A standard input format for multiperiod stochastic linear programs. *COAL Newsletter*, 17:1–19, 1987.
4. 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.
5. J. R. Birge and R. Louveaux. *Introduction to Stochastic Programming*. Springer, New York, 1997.
6. J. R. Birge and L. Qi. Computing block-angular Karmarkar projections with applications to stochastic programming. *Management Science*, 34:1472–1479, 1988.

7. J. V. Burke and M. C. Ferris. Weak sharp minima in mathematical programming. *SIAM Journal on Control and Optimization*, 31:1340–1359, 1993.
8. E. Frangière, J. Gondzio, and J.-P. Vial. Building and solving large-scale stochastic programs on an affordable distributed computing system. *Annals of Operations Research*, 2000. To appear.
9. H. I. Gassmann and E. Schweitzer. A comprehensive input format for stochastic linear programs. Working Paper WP-96-1, School of Business Administration, Dalhousie University, Halifax, Canada, December 1997.
10. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, MA, 1994.
11. J. Gondzio and J.-P Vial. Warm start and ϵ-subgradients in the cutting plane scheme for block-angular linear programs. *Computational Optimization and Applications*, 14:17–36, 1999.
12. J.-P. Goux, S. Kulkarni, J. T. Linderoth, and M. E. Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, 2000.
13. J.-P. Goux, J. T. Linderoth, and M. E. Yoder. Metacomputing and the master-worker paradigm. Preprint ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, 2000.
14. J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms II*. Comprehensive Studies in Mathematics. Springer-Verlag, 1993.
15. 1997. `http://www-personal.umich.edu/~jrbirge/dholmes/SPTSlists.html`.
16. K. C. Kiwiel. Proximity control in bundle methods for convex nondifferentiable minimization. *Mathematical Programming*, 46:105–122, 1990.
17. 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`.
18. O. L. Mangasarian. *Nonlinear Programming*. McGraw-Hill, New York, 1969.
19. J. M. Mulvey and A. Ruszczyński. A new scenario decomposition method for large scale stochastic optimization. *Operations Research*, 43:477–490, 1995.
20. R. T. Rockafellar. *Convex Analysis*. Princeton University Press, Princeton, 1970.
21. A. Ruszczyński. A regularized decomposition for minimizing a sum of polyhedral functions. *Mathematical Programming*, 35:309–333, 1986.
22. A. Ruszczyński. Parallel decomposition of multistage stochastic programming problems. *Mathematical Programming*, 58:201–228, 1993.
23. S. Sen, R. D. Doverspike, and S. Cosares. Network planning with random demand. *Telecommunications Systems*, 3:11–30, 1994.
24. Alexander Shapiro and Tito Homem-de-Mello. On the rate of convergence of optimal solutions of Monte Carlo approximations of stochastic programs. *SIAM Journal on Optimization*, 11(1):70–86, 2001.
25. 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.
26. R. Wunderling. *Paralleler und Objektorientierter Simplex-Algorithmus*. PhD thesis, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1996.