Computational Optimization and Applications, 24, 207–250, 2003

Decomposition Algorithms for Stochastic Programming on a Computational Grid

JEFF LINDEROTH jt13@lehigh.edu
Industrial and Systems Engineering Department, Lehigh University, 200 West Packer Avenue, Bethlehem, PA 18015, USA

STEPHEN WRIGHT swright@cs.wisc.edu
Computer Sciences Department, University of Wisconsin, 1210 W. Dayton Street, Madison, WI 53706, USA

Received April, 2001; Revised September 10, 2002; Accepted October 31, 2002

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 two-stage stochastic linear programming problem with fixed recourse, defined as follows:

$$\min c^T x + \sum_{i=1}^{N} p_i q_i^T y_i, \tag{1a}$$

$$\text{subject to } Ax = b, \quad x \geq 0, \tag{1b}$$

$$Wy_i = h_i - T_i x, \quad y(\omega_i) \geq 0, \quad i = 1, 2, \ldots, N. \tag{1c}$$

The N scenarios are represented by $\omega_1, \omega_2, \ldots, \omega_N$, with probabilities p_i and data objects (q_i, T_i, h_i) for each $i = 1, 2, \ldots, N$. The unknowns are first-stage variables $x \in \mathsf{R}^{n_1}$ and second-stage variables $y_i \in \mathsf{R}^{n_2}$, $i = 1, 2, \ldots, N$. This formulation is sometimes known as the "deterministic equivalent" because it lists the unknowns for all scenarios explicitly and poses the problem as a (structured) linear program.

An alternative formulation is obtained by defining the ith second-stage problem as a linear program (LP) parametrized by the first-stage variables x, that is,

$$\mathcal{Q}_i(x) \stackrel{\text{def}}{=} \min_{y_i} q_i^T y_i \quad \text{subject to } Wy_i = h_i - T_i x, \quad y_i \geq 0, \tag{2}$$

so that $\mathcal{Q}_i(\cdot)$ is a piecewise linear convex function. The objective in (1a) is then

$$\mathcal{Q}(x) \stackrel{\text{def}}{=} c^T x + \sum_{i=1}^{N} p_i \mathcal{Q}_i(x), \tag{3}$$

and we can restate (1) as

$$\min_x \mathcal{Q}(x), \quad \text{subject to } Ax = b,\ x \geq 0. \tag{4}$$

We can derive subgradient information for $\mathcal{Q}_i(x)$ by considering dual solutions of (2). If π_i is the Lagrange multiplier vector for the equality constraint in (2), it is easy to show that

$$-T_i^T \pi_i \in \partial \mathcal{Q}_i(x), \tag{5}$$

where $\partial \mathcal{Q}_i$ denotes the subgradient of $\mathcal{Q}_i$. By Rockafellar [19, Theorem 23.8], using polyhedrality of each $\mathcal{Q}_i$, we have from (3) that

$$\partial \mathcal{Q}(x) = c + \sum_{i=1}^{N} p_i \partial \mathcal{Q}_i(x), \tag{6}$$

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

Let $\mathcal{S}$ denote the solution set for (4). Since (4) is a convex program, $\mathcal{S}$ is closed and convex. If $\mathcal{S}$ is nonempty, the projection operator $P(\cdot)$ onto $\mathcal{S}$ is well defined.

Subgradient information can be used by algorithms in different ways. Cutting-plane methods use this information to construct convex estimates of $\mathcal{Q}(x)$, and obtain each iterate by minimizing this estimate, as in the L-shaped methods described in Section 2. This approach can be stabilized by the use of a quadratic regularization term [15, 20, 21] 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 of $\mathcal{Q}$. The latter approach has been explored by Bahn et al. [1] and applied to a large stochastic programming problem by Frangière et al. [9].

Parallel implementation of these approaches is obvious in principle. Because evaluation of $\mathcal{Q}_i(x)$ and elements of its subdifferential can be carried out independently for each $i = 1, 2, \ldots, N$, we can partition the scenarios $i = 1, 2, \ldots, N$ into clusters of scenarios and define a computational task to be the solution of all the second-stage LPs (2) in a number of clusters. Each such task could be assigned to an available worker processor. Bunching techniques (see [5, Section 5.4]) can be used to exploit the similarity between different scenarios within a chunk. To avoid inefficiency, each task should contain enough scenarios that its computational requirements 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, and supercomputer nodes. Specifically, we use the environment provided by the Condor

system [16] running the MW runtime library [12, 13], a software layer that significantly simplifies the process of implementing parallel algorithms.

For the dimensions of problems and the size of the computational grids considered in this paper, evaluation of the functions $\mathcal{Q}_i(x)$ and their subgradients at a single x sometimes 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 a parallel environment in which failure or suspension of worker processors while performing computation is not infrequent. 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 described by Birge et al. [3, 4], Birge and Qi [6], Ruszczyński [21], and Frangière et al. [9]. In [3], 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 first-stage iterate. Birge and Qi [6] describe an interior-point method for two-stage problems, in which the linear algebra operations are implemented in parallel by exploiting the structure of the two-stage problem. However, this approach involves significant data movement and does not scale particularly well. In [9], the second-stage problems (2) are solved concurrently and inexactly by using an interior-point code. The master process maintains an upper bound on the optimal objective, and this information is used along with the approximate subgradients construct an approximate truncated epigraph of the function. The analytic center of this epigraph is computed periodically to obtain a new iterate. The numerical results in [9] 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.

The approach that is perhaps closest to the ones we describe in this paper is that of Ruszczyński [21]. When applied to two-stage problems (1), this algorithm consists of processes that solve each second-stage problem (2) at the latest available value of x to generate cuts; and a master process that solves a cutting-plane problem with the latest available cuts and a quadratic regularization term to generate new iterates x. The master process and second-stage processes can execute in parallel and share information asynchronously. This approach is essentially an asynchronous parallel version of the serial bundle-trust-region approaches described by Ruszczyński [20], Kiwiel [15], and Hiriart-Urruty and Lemaréchal [14, ch. XV]. Algorithm ATR described in Section 4 likewise is an asynchronous parallel version of the bundle-trust-region method TR of Section 3, although the asynchronicity in the algorithm ATR described in Section 4 is more structured than that considered in [21]. In addition, ℓ_∞ trust regions take the place of quadratic regularization terms in both Algorithms TR and ATR. We discuss the relationships between all these methods further in later sections.

The remainder of this paper is structured as follows. Section 2 describes an L-shaped method and an asynchronous variant. Algorithm TR, a bundle-trust-region method with

ℓ_∞ trust regions is described and analyzed in Section 3, while its asynchronous cousin Algorithm ATR is described and analyzed in Section 4. Section 5 discusses computational grids and implementations of the algorithms on these platforms. Finally, computational results are given in Section 6.

2. L-shaped methods

We describe briefly a well known variant of the L-shaped method for solving (4), together with an asynchronous variant.

2.1. *The multicut L-shaped method*

The L-shaped method of Van Slyke and Wets [25] for solving (4) proceeds by finding subgradients of partial sums of the terms that make up $\mathcal{Q}$ (3), together with linear inequalities that define the domain of $\mathcal{Q}$. We sketch the approach here, and refer to Birge and Louveaux [5, ch. 5] for a more complete description.

Suppose that the second-stage scenarios $1, 2, \ldots, N$ are partitioned into C clusters denoted by $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_C$. Let $\mathcal{Q}_{[j]}$ represent the partial sum from (3) corresponding to the cluster $\mathcal{N}_j$; that is,

$$\mathcal{Q}_{[j]}(x) = \sum_{i \in \mathcal{N}_j} p_i \mathcal{Q}_i(x). \tag{7}$$

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

$$m^k_{[j]}(x) = \inf \left\{ \theta_j \mid \theta_j e \geq F^k_{[j]} x + f^k_{[j]} \right\}, \tag{8}$$

where $e = (1, 1, \ldots, 1)^T$ and $F^k_{[j]}$ is a matrix whose rows are subgradients of $\mathcal{Q}_{[j]}$ at previous iterates of the algorithm. The constraints in (8) are called *optimality cuts*. A subgradient g_j of $\mathcal{Q}_{[j]}$ is obtained from the dual solutions π_i of (2) for each $i \in \mathcal{N}_j$ as follows:

$$g_j = -\sum_{i \in \mathcal{N}_j} p_i T_i^T \pi_i; \tag{9}$$

see (5) and (6). An optimality cut is not added to the model $m^k_{[j]}$ if the model function takes on the same value as $\mathcal{Q}_{[j]}$ at iteration k. Cuts may also be deleted in the manner described below. The algorithm also maintains a collection of *feasibility cuts* of the form

$$D^k x \geq d^k, \tag{10}$$

which have the effect of excluding values of x for which some of the second-stage linear programs (2) are infeasible. By Farkas's theorem (see [17, p. 31]), if the constraints in (2)

are infeasible, there exists π_i with the following properties:

$$W^T\pi_i \leq 0, \quad [h_i - T_i x]^T \pi_i > 0.$$

(In fact, such a π_i can be obtained from the dual simplex method for the feasibility problem for (2).) To exclude this x from further consideration, we simply add the inequality $[h_i - T_i x]^T \pi_i \leq 0$ to the constraint set (10).

The kth iterate x^k of the multicut L-shaped method is obtained by solving the following approximation to (4):

$$\min_x m_k(x), \quad \text{subject to } D^k x \geq d^k, \ Ax = b, \ x \geq 0, \tag{11}$$

where

$$m_k(x) \stackrel{\text{def}}{=} c^T x + \sum_{j=1}^{C} m_{[j]}^k(x). \tag{12}$$

In practice, we substitute from (8) to obtain the following linear program:

$$\min_{x,\theta_1,\ldots,\theta_C} c^T x + \sum_{j=1}^{C} \theta_j, \tag{13a}$$

$$\text{subject to} \quad \theta_j e \geq F_{[j]}^k x + f_{[j]}^k, \quad j = 1, 2, \ldots, C, \tag{13b}$$

$$D^k x \geq d^k, \tag{13c}$$

$$Ax = b, \ x \geq 0. \tag{13d}$$

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

Under this assumption, feasibility cuts (10) and (13c) are not present. Our algorithms and their analysis can be generalized to handle situations in which Assumption 1 does not hold, but for the sake of simplifying our analysis, we avoid discussing this more general case here.

Under 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 $k = 0, 1, 2, \ldots$
    obtain x^{k+1} by solving (11);
    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. *Asynchronous parallel variant of the L-shaped method*

The L-shaped approach lends itself naturally to implementation in a master-worker framework. The problem (13) 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

$$\min_x m(x), \quad \text{subject to } Ax = b,\ x \geq 0. \tag{14}$$

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`. Worker routine for evaluating $\mathcal{Q}_{[j]}(x)$ defined by (7) for a given x and one or more of the clusters $j = 1, 2, \ldots, C$, in the process generating a subgradient g_j of each $Q_{[j]}(x)$. This task 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`. Master routine that places 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 all these tasks leads to evaluation of $\mathcal{Q}(x)$.
- `initialize`. Master routine that performs initial bookkeeping, culminating in a call to `evaluate` for the initial point x^0.
- `act_on_completed_task`. Master routine, 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 tasks 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 then calls `evaluate`.

In our implementation of both this algorithm and its more sophisticated cousin Algorithm ATR of Section 4, a single task consists of the evaluation of one or more clusters $\mathcal{N}_j$. We may bundle, say, 2 or 4 clusters into a single computational task, to make 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. We use T to denote the number of computational tasks, and $\mathcal{T}_r, r = 1, 2, \ldots, T$ to denote a partitioning of $\{1, 2, \ldots, C\}$, so that task r consists of evaluation of the clusters $j \in \mathcal{T}_r$.

The implementation depends on a "synchronicity" parameter σ which is the proportion of tasks 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: `partial_evaluate`(x^q, q, r)
Given x^q, index q, and task number r, evaluate $\mathcal{Q}_{[j]}(x^q)$ from (7) for all $j \in \mathcal{T}_r$
    together with partial subgradients g_j from (9);
Activate `act_on_completed_task` (x^q, q, r) on the master processor.

ALS: `evaluate` (x^q, q)
for $r = 1, 2, \ldots, T$ (possibly concurrently)
    `partial_evaluate`(x^q, q, r);
end (for)

ALS: `initialize`
determine number of clusters C and number of tasks T,
    and the partitions $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_C$ and $\mathcal{T}_1, \mathcal{T}_2 \ldots, \mathcal{T}_T$;
choose tolerance ϵ_{tol};
choose starting point x^0;
choose threshold $\sigma \in (0, 1]$;
$\mathcal{Q}_{\min} \leftarrow \infty$;
$k \leftarrow 0$, $\texttt{speceval}_0 \leftarrow \texttt{false}$, $t_0 \leftarrow 0$;
`evaluate`$(x^0, 0)$.

ALS: `act_on_completed_task`(x^q, q, r)
$t_q \leftarrow t_q + 1$;
for each $j \in \mathcal{T}_r$
    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 (14) 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 [20] and the bundle-trust-region approaches of Kiwiel [15] and Hiriart-Urruty and Lemaréchal [14, ch. 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. We discuss these differences further in Section 3.3.

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 [20], for instance, which exploits the piecewise-linear nature of the objectives $\mathcal{Q}_i$ in (2). Because of our weaker conditions on the subgradient information, we cannot prove a finite termination result of the type presented in [20, Section 3]. However, these conditions potentially allow our algorithms to be extended to a more general class of convex nondifferentiable functions.

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 (13) as follows:

$$-\Delta e \leq x - x^k \leq \Delta e, \tag{15}$$

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 (15), 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 (15) 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:

$$\min_x m_{k,\ell}(x) \quad \text{subject to } Ax = b,\ x \geq 0,\ \|x - x^k\|_\infty \leq \Delta_{k,\ell} \tag{16}$$

(cf. (11)). By expanding this problem in a similar fashion to (13), we obtain

$$\min_{x,\theta_1,\ldots,\theta_C} c^T x + \sum_{j=1}^{C} \theta_j, \tag{17a}$$

$$\text{subject to} \quad \theta_j e \geq F_{[j]}^{k,\ell} x + f_{[j]}^{k,\ell}, \quad j = 1, 2, \ldots, C, \tag{17b}$$

$$Ax = b,\ x \geq 0, \tag{17c}$$

$$-\Delta_{k,\ell} e \leq x - x^k \leq \Delta_{k,\ell} e. \tag{17d}$$

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

$$m_{k,0}(x^k) = \mathcal{Q}(x^k), \tag{18a}$$

$$m_{k,0} \text{ is a convex, piecewise linear understimate of } \mathcal{Q}. \tag{18b}$$

Denoting the solution of the subproblem (17) 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 (4)) 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

$$\mathcal{Q}(x^{k,\ell}) \leq \mathcal{Q}(x^k) - \xi(\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell})). \tag{19}$$

(A typical value for ξ is 10^{-4}.)

If the test (19) 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 $j = 1, 2, \ldots, C$.

Given some parameter $\eta \in (\xi, 1)$, 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 (17) with positive Lagrange multiplier
        `possible_delete` $\leftarrow$ `false`;
    else if the cut was generated at an earlier minor iteration
        $\bar{\ell} = 0, 1, \ldots, \ell - 1$ such that

$$\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) > \eta[\mathcal{Q}(x^k) - m_{k,\bar{\ell}}(x^{k,\bar{\ell}})] \tag{20}$$

        `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 (17). More details are given in Section 6.2.

Because we retain all cuts generated at x^k during the course of major iteration k, the following extension of (18a) holds:

$$m_{k,\ell}(x^k) = \mathcal{Q}(x^k), \quad \ell = 0, 1, 2, \ldots. \tag{21}$$

Since we add only subgradient information, the following generalization of (18b) also holds uniformly:

$$m_{k,\ell} \text{ is a convex, piecewise linear underestimate of } \mathcal{Q}, \text{ for } \ell = 0, 1, 2, \ldots. \tag{22}$$

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 (19) fails to hold. We do so if the match between

model and objective appears to be particularly poor, adapting the procedure of Kiwiel [15, p. 109] for increasing the coefficient of the quadratic penalty term in his regularized bundle method. If $\mathcal{Q}(x^{k,\ell})$ exceeds $\mathcal{Q}(x^k)$ by more than an estimate of the quantity

$$\max_{\|x-x^k\|_\infty \le 1} \mathcal{Q}(x^k) - \mathcal{Q}(x), \tag{23}$$

We conclude that the "upside" variation of the function $\mathcal{Q}$ deviates too much from its "downside" variation, and we reduce the trust-region radius $\Delta_{k,\ell+1}$ so as to bring these quantities more nearly into line. Our estimate of (23) is simply

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

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

$$\rho = \min(1, \Delta_{k,\ell}) \frac{\mathcal{Q}(x^{k,\ell}) - \mathcal{Q}(x^k)}{\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell})}; \tag{24}$$

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` $\leftarrow 0$;

If the test (19) 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 (18) 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

$$\mathcal{Q}(x^{k,\ell}) \le \mathcal{Q}(x^k) - 0.5(\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell})), \quad \|x^k - x^{k,\ell}\|_\infty = \Delta_{k,\ell}, \tag{25}$$

then we set

$$\Delta_{k+1,0} = \min(\Delta_{\text{hi}}, 2\Delta_{k,\ell}), \tag{26}$$

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

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

$$\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \leq \epsilon_{\text{tol}}(1 + |\mathcal{Q}(x^k)|). \tag{27}$$

Algorithm TR

choose $\xi \in (0, 1/2)$, cut deletion parameter $\eta \in (\xi, 1)$,
    maximum trust region Δ_{hi}, tolerance ϵ_{tol};
choose starting point x^0;
define initial model $m_{0,0}$ with the properties (18) (for $k = 0$);
choose $\Delta_{0,0} \in [1, \Delta_{\text{hi}}]$;
for $k = 0, 1, 2, \ldots$
    `finishedMinorIteration` $\leftarrow$ `false`;
    $\ell \leftarrow 0$; `counter` $\leftarrow 0$;
    repeat
        solve (16) to obtain $x^{k,\ell}$;
        if (27) is satisfied
            STOP with approximate solution x^k;
        evaluate function and subgradient at $x^{k,\ell}$;
        if (19) 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 (18) (with $k + 1$ replacing k);
            choose $\Delta_{k+1,0} \in [\Delta_{k,\ell}, \Delta_{\text{hi}}]$ according to (25), (26);
            `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 x^k that approaches the solution set $\mathcal{S}$ (Theorem 2).

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:

$$\mathcal{L}(\mathcal{Q}_0) = \{x \mid Ax = b, x \geq 0, \mathcal{Q}(x) \leq \mathcal{Q}_0\}, \tag{28}$$

$$\mathcal{L}(\mathcal{Q}_0; \Delta) = \{x \mid Ax = b, x \geq 0, \|x - y\|_\infty \leq \Delta, \text{ for some } y \in \mathcal{L}(\mathcal{Q}_0)\}, \tag{29}$$

$$\beta = \sup\{\|g\|_1 \mid g \in \partial\mathcal{Q}(x), \text{ for some } x \in \mathcal{L}(\mathcal{Q}_0; \Delta_{\text{hi}})\}. \tag{30}$$

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

$$\mathcal{Q}(x) - \mathcal{Q}^* \leq g^T(x - P(x)), \quad \text{for all } x \in \mathcal{L}(\mathcal{Q}_0; \Delta_{\text{hi}}), \text{ all } g \in \partial\mathcal{Q}(x),$$

so that

$$\mathcal{Q}(x) - \mathcal{Q}^* \leq \|g\|_1 \|x - P(x)\|_\infty \leq \beta\|x - P(x)\|_\infty. \tag{31}$$

We start by showing that the optimal objective value for (16) cannot decrease from one minor iteration to the next.

Lemma 1. *Suppose that $x^{k,\ell}$ does not satisfy the acceptance test* (19). *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 (17). Using $\bar{F}^{k,\ell}_{[j]}$ and $\bar{f}^{k,\ell}_{[j]}$ to denote the active rows in $F^{k,\ell}_{[j]}$ and $f^{k,\ell}_{[j]}$, we have that $x^{k,\ell}$ is also the solution of the following linear program (in which the inactive cuts are not present):

$$\min_{x,\theta_1,\ldots,\theta_C} c^T x + \sum_{j=1}^{C} \theta_j, \tag{32a}$$

$$\text{subject to} \quad \theta_j e \geq \bar{F}^{k,\ell}_{[j]} x + \bar{f}^{k,\ell}_{[j]}, \quad j = 1, 2, \ldots, C, \tag{32b}$$

$$Ax = b, \; x \geq 0, \tag{32c}$$

$$-\Delta_{k,\ell} e \leq x - x^k \leq \Delta_{k,\ell} e. \tag{32d}$$

The subproblem to be solved for $x^{k,\ell+1}$ differs from (32) in two ways. First, additional rows may be added to $\bar{F}^{k,\ell}_{[j]}$ and $\bar{f}^{k,\ell}_{[j]}$, consisting of function values and subgradients obtained at $x^{k,\ell}$ and also inactive cuts carried over from the previous (17). 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 (32), 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*

$$\begin{aligned} m_{k,\ell}(x^k) - m_{k,\ell}(x^{k,\ell}) &= \mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \\ &\geq \min\left(\frac{\Delta_{k,\ell}}{\|x^k - P(x^k)\|_\infty}, 1\right)[\mathcal{Q}(x^k) - \mathcal{Q}^*]. \end{aligned} \tag{33}$$

Proof: The first equality follows immediately from (21). To prove (33), consider the following subproblem in the scalar τ:

$$\min_{\tau\in[0,1]} m_{k,\ell}(x^k + \tau[P(x^k) - x^k]) \quad \text{subject to } \|\tau[P(x^k) - x^k]\|_\infty \le \Delta_{k,\ell}. \tag{34}$$

Denoting the solution of this problem by $\tau_{k,\ell}$, we have by comparison with (16) that

$$m_{k,\ell}(x^{k,\ell}) \le m_{k,\ell}(x^k + \tau_{k,\ell}[P(x^k) - x^k]). \tag{35}$$

If $\tau = 1$ is feasible in (34), we have from (35) and (22) that

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

Hence, we have from (21) that

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

so that (33) holds in this case.

When $\tau = 1$ is infeasible for (34), consider setting $\tau = \Delta_{k,\ell}/\|x^k - P(x^k)\|_\infty$ (which is certainly feasible for (34)). We have from (35), the definition of $\tau_{k,\ell}$, (22), 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 (21), 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 (33) 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,

$$E_k \stackrel{\text{def}}{=} \min_{\bar{k}=0,1,\ldots,k} \|x^{\bar{k}} - P(x^{\bar{k}})\|_\infty. \tag{36}$$

Note that E_k decreases monotonically with k. We also define F_k as follows

$$F_k \stackrel{\text{def}}{=} \min_{\bar{k}=0,1,\ldots,k, x^{\bar{k}} \notin \mathcal{S}} \frac{\mathcal{Q}(x^{\bar{k}}) - \mathcal{Q}^*}{\|x^{\bar{k}} - P(x^{\bar{k}})\|_\infty}, \tag{37}$$

with the convention that $F_k = 0$ if $x^{\bar{k}} \in \mathcal{S}$ for any $\bar{k} \leq k$. By monotonicity of $\{\mathcal{Q}(x^k)\}$, we have $F_k > 0$ whenever $x^k \notin \mathcal{S}$. Note also from (31) and the fact that $x^k \in \mathcal{L}(\mathcal{Q}_0; \Delta_{\text{hi}})$ that

$$F_k \leq \beta, \quad k = 0, 1, 2, \ldots \tag{38}$$

Lemma 3. *For all trust regions $\Delta_{k,\ell}$ used in the course of Algorithm TR, we have*

$$\Delta_{k,\ell} \geq (1/4) \min(E_k, F_k/\beta), \tag{39}$$

for β defined in (30).

Proof: Suppose for contradiction that there are indices k and ℓ such that $\Delta_{k,\ell} < (1/4)$ $\min(E_k, F_k/\beta)$. 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 $\bar{k} \leq k$) such that

$$\Delta_{\bar{k},\bar{\ell}} < \min(E_k, F_k/\beta), \tag{40}$$

and $\rho > 1$ in (24), that is,

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

where we used (38) in (40) to deduce that $\Delta_{\bar{k},\bar{\ell}} < 1$. By applying Lemma 2, and using (40) again, we have

$$\begin{aligned} \mathcal{Q}(x^{\bar{k}}) - m_{\bar{k},\bar{\ell}}(x^{\bar{k},\bar{\ell}}) &\geq \min\left(\frac{\Delta_{\bar{k},\bar{\ell}}}{\|x^{\bar{k}} - P(x^{\bar{k}})\|_\infty}, 1 \right) [\mathcal{Q}(x^{\bar{k}}) - \mathcal{Q}^*] \\ &= \frac{\Delta_{\bar{k},\bar{\ell}}}{\|x^{\bar{k}} - P(x^{\bar{k}})\|_\infty} [\mathcal{Q}(x^{\bar{k}}) - \mathcal{Q}^*] \end{aligned} \tag{42}$$

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

$$\mathcal{Q}(x^{\bar{k},\bar{\ell}}) - \mathcal{Q}(x^{\bar{k}}) \geq \frac{\mathcal{Q}(x^{\bar{k}}) - \mathcal{Q}^*}{\|x^{\bar{k}} - P(x^{\bar{k}})\|_\infty} \geq F_{\bar{k}} \geq F_k. \tag{43}$$

By using standard properties of subgradients, we have

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

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

$$\|g_{\bar{\ell}}\|_1 \geq \frac{1}{\Delta_{\bar{k},\bar{\ell}}}[\mathcal{Q}(x^{\bar{k},\bar{\ell}}) - \mathcal{Q}(x^{\bar{k}})] \geq \frac{1}{\Delta_{\bar{k},\bar{\ell}}} F_{\bar{k}} > \beta.$$

However, since $x^{\bar{k},\bar{\ell}} \in \mathcal{L}(\mathcal{Q}_0; \Delta_{\text{hi}})$, we have from (30) 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 (19) and (20), respectively.

Lemma 4. *Let $\epsilon_{\text{tol}} = 0$ in Algorithm TR, and let ξ and η be the constants from* (19) *and* (20), *respectively. Let ℓ_1 be any index such that x^{k,ℓ_1} fails to satisfy the test* (19). *Then either the sequence of inner iterations eventually yields a point x^{k,ℓ_2} satisfying the acceptance test* (19), *or there is an index $\ell_2 > \ell_1$ such that*

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

Proof: Suppose for contradiction that the none of the minor iterations following ℓ_1 satisfies either (19) or the criterion (45); that is,

$$\mathcal{Q}(x^k) - m_{k,q}(x^{k,q}) > \eta[\mathcal{Q}(x^k) - m_{k,\ell_1}(x^{k,\ell_1})], \quad \text{for all } q > \ell_1. \quad (46)$$

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 $x^{k,\ell}$, $\ell = \ell_1, \ell_1 + 1, \ldots$ are retained throughout the major iteration k, we have

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

By definition of the subgradient, we have

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

Therefore, from (22) and (47), it follows that

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

so that

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

Since $\mathcal{Q}(x^k) \leq \mathcal{Q}(x^0) = \mathcal{Q}_0$, we have from (28) that $x^k \in \mathcal{L}(\mathcal{Q}_0)$. Therefore, from the definition (29) and the fact that $\|x^{k,\ell} - x^k\|_\infty \leq \Delta_{k,\ell} \leq \Delta_{\text{hi}}$, we have that $x^{k,\ell} \in \mathcal{L}(\mathcal{Q}_0; \Delta_{\text{hi}})$. It follows from (30) and (49) that

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

Since $x^{k,\ell}$ is rejected by the test (19), we have from (47) and Lemma 1 that the following inequalities hold:

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

By rearranging this expression, we obtain

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

Recalling that $\eta \in (\xi, 1)$, we consider the following neighborhood of $x^{k,\ell}$:

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

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

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

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

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

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 $\|x^{k,\ell} - x^k\|_\infty \leq \Delta_{k,\ell} \leq \Delta_{\text{hi}}$ for all $\ell \geq \ell_1$, it is impossible for an infinite sequence $\{x^{k,\ell}\}_{\ell \geq \ell_1}$ to satisfy (53). We conclude that (45) must hold for some $\ell_2 \geq \ell_1$, 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* $\epsilon_{\text{tol}} = 0$.
(i) *If* $x^k \notin \mathcal{S}$, *there is an* $\ell \geq 0$ *such that* $x^{k,\ell}$ *satisfies* (19).
(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 (19) always fails. By applying Lemma 4 recursively, we can identify a sequence of indices $0 < \ell_1 < \ell_2 < \ldots$ such that

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

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

$$\Delta_{k,\ell} \geq (1/4)\min(E_k; F_k/\beta) \stackrel{\text{def}}{=} \bar{\Delta}_{\text{lo}} > 0, \quad \text{for all } \ell = 0, 1, 2, \ldots,$$

so the right-hand side of (33) is uniformly positive (independently of ℓ). However, (54) indicates that we can make $\mathcal{Q}(x^k) - m_{k,\ell_j}(x^{k,\ell_j})$ arbitrarily small by choosing j sufficiently large, contradicting (33).

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 must have $\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) = 0$. Hence, from (22), we have

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

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 that the convergence is monotonic. □

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

Theorem 2. *Suppose that* $\epsilon_{\text{tol}} = 0$. *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 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

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

Hence, by boundedness of the subgradients (see (30)), and using the definitions (36) and (37), we can indentify a constant $\bar{\epsilon} > 0$ such that $E_k \geq \bar{\epsilon}$ and $F_k \geq \bar{\epsilon}$ for all k. Therefore, by Lemma 2, we have

$$\mathcal{Q}(x^k) - m_{k,\ell}(x^{k,\ell}) \geq \min(\Delta_{k,\ell}/\bar{\epsilon}, 1)[\bar{\mathcal{Q}} - \mathcal{Q}^*], \quad k = 0, 1, 2, \ldots. \tag{55}$$

For each major iteration index k, let $\ell(k)$ be the minor iteration index that passes the acceptance test (19). By combining (19) with (55), we have that

$$\mathcal{Q}(x^k) - \mathcal{Q}(x^{k+1}) \geq \xi \min\left(\Delta_{k,\ell(k)}/\bar{\epsilon}, 1\right)[\bar{\mathcal{Q}} - \mathcal{Q}^*].$$

Since $Q(x^k) - \mathcal{Q}(x^{k+1}) \to 0$, we deduce that $\lim_{k\to\infty} \Delta_{k,\ell(k)} = 0$. However, since E_k and F_k are bounded away from 0, we have from Lemma 3 that $\Delta_{k,\ell}$ is bounded away from 0, giving a contradiction. We conclude that the second possibility (an infinite sequence $\{x^k\}$ not converging to $\mathcal{S}$) cannot occur either, so the proof is complete. □

It is easy to show that the algorithm terminates finitely when $\epsilon_{\text{tol}} > 0$. The argument in the proof of Theorem 1 shows that either the test (27) is satisfied at some minor iteration, or the algorithm identifies a new major iteration. Since the amount of reduction at each major iteration is at least $\xi\epsilon_{\text{tol}}$ (from (19)), and since we assume that a solution set exists, the number of major iterations must be finite.

3.3. Discussion

If a 2-norm trust region is used in place of the ∞-norm trust region of (16), it is well known that the solution of the subproblem

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

is identical to the solution of

$$\min_x m_{k,\ell}(x) + \lambda\|x - x^k\|^2 \quad \text{subject to } Ax = b, x \geq 0, \tag{56}$$

for some $\lambda \geq 0$. We can transform (56) to a quadratic program in the same fashion as the transformation of (16) to (17). The "regularized" or "proximal" bundle approaches described in Kiwiel [15], Hiriart-Urruty and Lemaréchal [14, ch. XV], and Ruszczyński [20, 21] work with the formulation (56). They manipulate the parameter λ directly rather than adjusting the trust-region radius Δ, more in the spirit of the Levenberg-Marquardt method for least-squares problems than of a true trust-region method.

We chose to devise and analyze an algorithm based on the ∞-norm trust region for two reasons. First, the linear-programming trust-region subproblems (17) can be solved by high-quality linear programming software, making the algorithm much easier to implement than the specialized quadratic programming solver required for (56). Although it is well known that the 2-norm trust region often yields a better search direction than the ∞-norm trust region when the objective is smooth, it is not clear if the same property holds for the function $\mathcal{Q}$, which is piecewise linear with a great many pieces. Our second reason was that the convergence analysis of the ∞-norm algorithm differs markedly from that of the regularized methods presented in [14, 15, 20, 21], making this project interesting from the theoretical point of view as well as computationally.

Finally, we note that aggregation of cuts, which is a feature of the regularized methods mentioned above and which is useful in limiting storage requirements, can also be performed to some extent in Algorithm TR. In Procedure Model-Update, we still need to retain the cuts generated at x^k, and at the earlier minor iterations ℓ satisfying (20). However, the cuts active at the solution of the subproblem (17) can be aggregated into C cuts, one for each index $j = 1, 2, \ldots, C$. To describe the aggregation, we use the alternative form (32) of the subproblem (17), from which the inactive cuts have been removed. Denoting the Lagrange multiplier vectors for the constraints (32b) by λ_j, $j = 1, 2, \ldots, C$, we have from the optimality conditions for (32b) that $\lambda_j \geq 0$ and $e^T \lambda_j = 1$, $j = 1, 2, \ldots, C$. Moreover, if we replace the constraints (32b) by the C aggregated constraints

$$\theta_j \geq \lambda_j^T \bar{F}_{[j]}^{k,\ell} x + \lambda_j^T \bar{f}_{[j]}^{k,\ell}, \quad j = 1, 2, \ldots, C, \tag{57}$$

then the solution of (32) and its optimal objective value are unchanged. Hence, in Procedure Model-Update, we can delete the "else if" clause concerning the constraints active at (17), and insert the addition of the cuts (57) to the end of the procedure.

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 $Q_{[j]}$, $j = 1, 2, \ldots, C$ (7) 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 $Q(x^k)$ 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 held 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 (19), 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 (16), 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 (7) and their subgradients have been evaluated) before generating a new point. We use a logical variable $\texttt{speceval}_q$ 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 $\texttt{speceval}_q$ 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:

$$\texttt{trsub}(x^I, \Delta) : \min_x m(x) \quad \text{subject to } Ax = b,\ x \geq 0,\ \|x - x^I\|_\infty \leq \Delta. \tag{58}$$

We refer to x^I as the *parent incumbent* of the solution of (58).

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. As in the description of ALS, we use $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_T$ to denote a partition of $\{1, 2, \ldots, C\}$ such that the rth computational task consists of the clusters $j \in \mathcal{T}_r$ (that is, evaluation of the partial sums $\mathcal{Q}_{[j]}$, $j \in \mathcal{T}_r$ and their subgradients). We use t_k to count the number of tasks for the evaluation of $\mathcal{Q}(x^k)$ that have been completed 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 (58). 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 (58) 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 $\texttt{trsub}(x^{I_q}, \Delta_q)$ (58).

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

$$\mathcal{Q}^{I_k} - m^k > \eta[\mathcal{Q}^{I_k} - m^{\bar{\ell}}] \tag{59}$$

        `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_{curr} 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

$$\rho = \min(1, \Delta_q)\frac{\mathcal{Q}(x^q) - \mathcal{Q}^{I_q}}{\mathcal{Q}^{I_q} - m^q}; \tag{60}$$

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 (25), (26). If on completion of evaluation of $\mathcal{Q}(x^q)$, the iterate x^q becomes the new incumbent, then we test the following condition:

$$\mathcal{Q}(x^q) \leq \mathcal{Q}^{I_q} - 0.5(\mathcal{Q}^{I_q} - m^q) \quad \text{and} \quad \|x^q - x^{I_q}\|_\infty = \Delta_q. \tag{61}$$

If this condition is satisfied, we set

$$\Delta_{\text{curr}} \leftarrow \max(\Delta_{\text{curr}}, \min(\Delta_{\text{hi}}, 2\Delta_q)). \tag{62}$$

The convergence test is also similar to the test (27) used for Algorithm TR. We terminate if, on generation of a new iterate x^k, we find that

$$\mathcal{Q}^I - m^k \leq \epsilon_{\text{tol}}(1 + |\mathcal{Q}^I|). \tag{63}$$

We now specify the four key routines of the Algorithm ATR, which serve a similar function to the four main routines of Algorithm ALS. The routine `partial_evaluate` defines a single task that executes on worker processors, while the other three routines execute on the master processor.

ATR: `partial_evaluate`(x^q, q, r)
Given x^q, index q, and task index r, evaluate $\mathcal{Q}_{[j]}(x^q)$ from (7) for each $j \in \mathcal{T}_r$,
    together with partial subgradients g_j from (9);
Activate `act_on_completed_task`(x^q, q, r) on the master processor.

ATR: `evaluate` (x^q, q)
for $r = 1, 2, \ldots, T$ (possibly concurrently)
    `partial_evaluate` (x^q, q, r);
end (for)

ATR: `initialization`(x^0)
determine number of clusters C and number of tasks T,
    and the partitions $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_C$ and $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_T$;
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: `act_on_completed_task`(x^q, q, r)
$t_q \leftarrow t_q + 1$;
for each $j \in \mathcal{T}_r$
    add $\mathcal{Q}_{[j]}(x^q)$ and cut g_j to the model m;
`basketFill` $\leftarrow$ `false`; `basketUpdate` $\leftarrow$ `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 $\mathcal{Q}(x^q) \leq \mathcal{Q}^{I_q} - \xi(\mathcal{Q}^{I_q} - m^q)$)
        (* make x^q the new incumbent *)
        $I \leftarrow q$; $\mathcal{Q}^I \leftarrow \mathcal{Q}(x^I)$;
        possibly increase Δ_{curr} according to (61) and (62);
        modify the model function by possibly deleting cuts not
        generated at x^q;
    else
        call Model-Update(k);
        call Reduce-$\Delta(q)$ to update Δ_{curr};
    end (if)
    $\mathcal{B} \leftarrow \mathcal{B} \backslash \{q\}$;
    `basketUpdate` $\leftarrow$ `true`;
else if $t_q \geq \sigma T$ **and** $|\mathcal{B}| < K$ **and** not `speceval`$_q$
    (* basket-filling phase: enough partial sums have been evaluated at x^q
        to trigger calculation of a new candidate iterate *)
    `speceval`$_q \leftarrow$ `true`; `basketFill` $\leftarrow$ `true`;
end (if)
if `basketFill` or `basketUpdate`
    $k \leftarrow k + 1$; set $\Delta_k \leftarrow \Delta_{\text{curr}}$; set $I_k \leftarrow I$;
    solve `trsub`(x^I, Δ_k) to obtain x^k;
    $m^k \leftarrow m(x^k)$;
    if (63) holds
        STOP;
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{k\}$;
    `speceval`$_k \leftarrow$ `false`; $t_k \leftarrow 0$;
    `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:

Every task is completed after a finite time. (64)

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^l generated by ATR lie in $\mathcal{L}(\mathcal{Q}_{\max})$, whereas all points x^k considered by the algorithm lie in $\mathcal{L}(\mathcal{Q}_{\max}; \Delta_{\text{hi}})$, where $\mathcal{L}(\cdot)$ and $\mathcal{L}(\cdot;\cdot)$ are defined by* (28) *and* (29)*, respectively, and $\mathcal{Q}_{\max}$ is defined by*

$$\mathcal{Q}_{\max} \stackrel{\text{def}}{=} \sup\{\mathcal{Q}(x) \mid Ax = b, x \geq 0, \|x - x^0\|_\infty \leq \Delta_{\text{hi}}\}.$$

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 $\|x^k - x^0\|_\infty \leq \Delta_k \leq \Delta_{\text{hi}}$, with $\mathcal{Q}^{l_k} = \mathcal{Q}^{-1} = \infty$.

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 $\mathcal{Q}^l = \mathcal{Q}(x^k)$, and by definition we have $\mathcal{Q}^l \leq \mathcal{Q}_{\max}$. Since all later incumbents must have objectives values smaller than this first $\mathcal{Q}^l$, 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 $\Delta_k \leq \Delta_{\text{hi}}$ either of x^0 or of one of the later incumbents x^l. 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. □

Extending (30), we redefine the bound β on the subgradients over the set $\mathcal{L}(\mathcal{Q}_{\max}; \Delta_{\text{hi}})$ as follows:

$$\beta = \sup\{\|g\|_1 \mid g \in \partial\mathcal{Q}(x), \quad \text{for some } x \in \mathcal{L}(\mathcal{Q}_{\max}; \Delta_{\text{hi}})\}. \tag{65}$$

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

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

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

Proof: We select any $k = k_1, k_1 + 1, \ldots, k_2 - 1$ and prove that $m^k \leq m^{k+1}$. 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 `trsub`(x^I, Δ_k) were still present in the model when we solved `trsub`(x^I, Δ_{k+1}) to obtain x^{k+1}. Moreover, since no new incumbent was accepted, Δ_{curr} cannot have been increased, and we have $\Delta_{k+1} \leq \Delta_k$. We now use the same argument as in the proof of Lemma 1 to deduce that $m^k \leq m^{k+1}$. □

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*

$$\mathcal{Q}^{I_k} - m^k \geq \min(\Delta_k, \|x^{I_k} - P(x^{I_k})\|_\infty) \frac{\mathcal{Q}^{I_k} - \mathcal{Q}^*}{\|x^{I_k} - P(x^{I_k})\|_\infty}. \tag{66}$$

The following analogue of Lemma 3 requires a slight redefinition of the quantity E_k from (36). We now define it to be the closest approach by an *incumbent* to the solution set, up to and including iteration k; that is,

$$E_k \stackrel{\text{def}}{=} \min_{\bar{k}=0,1,\ldots,k; I_{\bar{k}} \neq -1} \|x^{I_{\bar{k}}} - P(x^{I_{\bar{k}}})\|_\infty \tag{67}$$

We redefine F_k similarly as follows:

$$F_k \stackrel{\text{def}}{=} \min_{\bar{k}=0,1,\ldots,k; x^{I_{\bar{k}}} \notin \mathcal{S}, I_{\bar{k}} \neq -1} \frac{\mathcal{Q}(x^{I_{\bar{k}}}) - \mathcal{Q}^*}{\|x^{I_{\bar{k}}} - P(x^{I_{\bar{k}}})\|_\infty}. \tag{68}$$

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

Lemma 8. *For all trust regions Δ_k used in the course of Algorithm ATR, we have*

$$\Delta_k \geq (1/4) \min(E_k, F_k/\beta),$$

where β, E_k, and F_k are as defined in (65), (67), *and* (68), *respectively.*

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 $\epsilon_{\text{tol}} = 0$ in Algorithm ATR, and let $\bar{\eta}$ be any constant satisfying $0 < \bar{\eta} < 1$, $\bar{\eta} > \xi$, $\bar{\eta} \geq \eta$. 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*

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

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

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

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 a nonoptimal point cannot remain as the incumbent indefinitely. The following result is analogous to Theorem 1, and its proof relies on the earlier results in exactly the same way.

Theorem 3. *Suppose that $\epsilon_{\text{tol}} = 0$.*

(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 $\mathcal{Q}^I - m^k \downarrow 0$ 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 4. *Suppose that $\epsilon_{\text{tol}} = 0$. 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 [16], 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 [11] 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 [12, 13]) 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 this paper is the other side of MW, the application programming

interface. This interface takes the form of a set of three C++ abstract classes that must be reimplemented by the application programmer in a way that describes the particular application. These classes, named MWDriver, MWTask, and MWWorker, contain a total of ten methods which we describe briefly here, indicating how they are implemented for the particular case of the ATR and ALS algorithms.

5.3.1. 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 date files and 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. For 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, this data consists of the information from the input files for the stochastic programming problem. When the worker subsequently receives a task requiring it to solve a number of second-stage scenarios, it uses these files 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`: This routine is called after the termination of each task, to process the results of the task and to take any necessary actions. Algorithms ALS and ATR contain further details.

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.

5.3.2. 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 into simple data structures that can be communicated between master and workers using the primitives appropriate to the particular computational grid platform. 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 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.

5.3.3. 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 MW-Driver function `pack_worker_init_data( )`.
- `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`. On completion of the task, it passes the results back to the worker by using the appropriate function from the MWTask class, and returns to its wait loop. In our applications, `execute_task( )` formulates the second-stage linear programs in its clusters, uses linear programming software to solve them, and then calculates 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 in the different runs. We then tabulate and discuss the results. Finally, we compare with results obtained with a state-of-the-art code implementing the regularized decomposition algorithm.

6.1. Implementations and experiments

As noted earlier, we used the Condor-PVM implementation of MW for most of the runs reported here. Most of the computational time is taken up with solving linear programs, both by the master process (in solving the master problem to determine the next iterate) and in the tasks (which solve clusters of second-stage linear programs). We used the CPLEX simplex solver on the master processor [8] and the SOPLEX public-domain simplex code of 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, as noted below. 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 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 PC.

We used input files in SMPS format (see [2, 10]), and Monte Carlo sampling to obtain approximate problems with a specified number N of second-stage scenarios. In each experiment, we supplied a starting point to the code, obtained from the solution of a different sampled instance of the same problem. In mosts cases, the function value of the starting point was quite close to the optimal objective value.

We report on experience with two test problems. The first is the SSN problem, arising from a network design application of Sen et al. [23]. SSN is based on a graph with 89 arcs, each representing a telecommunications link between two cities. The first-stage variables represent the extra capacity to be added to each are to meet uncertain demands, which consist of requests for service between pairs of nodes in the graph. There is a bound on the total capacity to be added. For each set of demands, a route through the network of sufficient capacity to meet the demands must be found, otherwise a penalty term is added to the objective. The second-stage problems are network flow problems for calculating the routing for a given set of demands. Each such problem is nontrivial; there are 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 node pairs is not known exactly. Rather, there are three to seven possible scenarios for each demand, all independent of each other, giving a total of about 10^{70} possible scenarios.

The second test problem is a cargo flight scheduling application described by Mulvey and Ruszczyński [18], known as the "storm" problem. In this application, the first-stage problem contains 121 variables, while the second-stage problem contains 1259 variables. The total number of scenarios is about 10^{81}.

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,

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

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. Specifically, we delete the cut only if more than 100 master problems have been solved since the point at which it was generated. Our cut management strategy tends to lead to subproblems (16) and (58) with fairly large numbers of cuts but, in our experience, the storage required for these cuts and the time required to solve the subproblems remain reasonable.

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 of clusters C and the number of computational tasks T is varied as shown in the tables. Note that the number of second-stage LPs per cluster is therefore N/C while the number per computational task 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 T per evaluation of $\mathcal{Q}(x)$ and the basket size K. For instance, the synchronous TR and LS algorithms can never use more than T worker processors, since they evaluate $\mathcal{Q}$ at just one x at a time. In the case of TR and ATR, we request mid(25,200, $\lfloor (K+1)T/2 \rfloor$) workers. For ALS, we request mid(25, 200, 2T) 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/T identical 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. A word of caution is in order. In a dynamic computing environment such as a computational grid, the variance in the computing pool makes it difficult to draw firm conclusions about the relative effectiveness of parameter settings. Indeed, it is not the goal of this work to determine one best set of parameters for all cases. Instead, the goal is to design an algorithm is flexible enough to run efficiently in the dynamic computing environment of the computational grid. For these trials, we use a sampled approximation to the problem SSN [23], with $N = 10{,}000$ scenarios. The deterministic equivalent 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 sampled approximation with $N = 20{,}000$ scenarios.

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 (14) for Algorithm ALS, (16) for Algorithm TR, and (58) for Algorithm ATR.
- $|\mathcal{B}|$. Maximum size of the basket, also denoted above by K.
- *Number of tasks.* Denoted above by T, the number of computational tasks into which the evaluation of each $\mathcal{Q}(x)$ is divided.
- *Number of clusters.* Denoted above by C, identical to the number of partial subgradients produced at each evaluation point.
- *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.

6.3.1. Synchronicity and ALS. The first experiment was designed to gauge the effect of varying the synchronicity parameter σ in the ALS algorithm. Table 1 shows the results of a

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

Run	Points evaluated	σ	No. of tasks (T)	No. of clusters (C)	Max. processors allowed	Av. processors	Parallel efficiency	Max. no. of cuts in model	Master problem solve time (min)	Wall clock time (min)
ALS	98	.5	25	200	200	38	.35	19505	8.8	26.2
ALS	93	.7	25	200	200	33	.34	18545	7.9	24.1
ALS	99	.85	25	200	200	34	.25	19776	8.7	33.2
ALS	98	.5	50	200	200	33	.37	19501	8.6	23.6
ALS	97	.7	50	200	200	31	.36	19339	8.6	28.4
ALS	98	.85	50	200	200	32	.33	19573	8.7	29.7
ALS	97	.5	100	200	200	26	.45	19297	8.6	24.8
ALS	106	.7	100	200	200	16	.52	20420	9.6	35.6
ALS	99	.85	100	200	200	29	.41	19771	8.7	22.8
ALS	97	.5	200	200	200	28	.44	19292	8.5	26.2
ALS	99	.7	200	200	200	36	.39	19736	8.9	24.8
ALS	99	.85	200	200	200	40	.32	19767	9.0	27.0

series of trials of Algorithm ALS with three different values of σ (.5, .7, and .85) and four different choices for the number of tasks T (25, 50, 100, and 200). The number of clusters C was fixed at 200, so that up to 200 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 10, but was rarely more than 5 except in the version with $T = 200$. For $\sigma = .85$, there were usually just 2 (and never more than 3) points being evaluated simultaneously.

We conclude from this table that the performance of ALS is not particularly sensitive to the choice of σ, although there seems to be a slight preference for the smaller values $\sigma = .5$ and $\sigma = .7$. (The relatively long runtime for the case of $T = 100, \sigma = .7$ was due to a smaller number of workers being available from the Condor pool during that run.) Also, for the number of processors available, there is not a strong dependence of wallclock time on T. If more processors were available, we would expect to see an advantage for larger T, since the larger number of tasks would be able to keep the processors more fully occupied.

A note on typical task sizes: For $T = 200$, most tasks required between 0.5 and 3 seconds to execute on a worker machine, while for $T = 25$, between 3.3 and 32 seconds were required per task. We used only Linux machines for these runs, which were performed in September 2002, when the Wisconsin Condor pool contained many fast machines of this kind. Few pre-emptions were experienced, but the pool was quite diverse in speed; the ratio of benchmark performance between the fastest and slowest worker used in any given run ranged up to 10.

One advantage of the ALS algorithm that we noted was that the asymptotic convergence 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 to a solution of high accuracy.

6.3.2. TR and ATR performance: Slower Condor pool. The second experiment measured the effect of the basket size $|\mathcal{B}|$ and the number of clusters C and tasks T on computational performance of ATR, in a lower-quality Condor pool. Recall that an increase in basket size reduces the synchronization requirements of the algorithm, and may therefore be effective when some workers in the pool are notably slower than others, or when workers are liable to be suspended during a run. The number of clusters C is likely to affect the number of iterations required; a higher C yields richer subgradient information and therefore a reduced iteration count, while possibly increasing the amount of time required to solve each master problem. The number of workers that can be used effectively can be increased by increasing the number of tasks T. However, a too-high value of T can result in tasks that are too small, and require too much work from the master in acting on the results that return from the tasks.

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

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

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

Run	Points evaluated	$\|\mathcal{B}\|$ (K)	No. of tasks (T)	No. of clusters (C)	Max. processors allowed	Av. processors	Parallel efficiency	Max. no. of cuts in model	Master problem 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

The parameter σ was fixed at .7 in all these runs, as we noted little sensitivity to the value of this parameter within a fairly broad range.

The results reported in Tables 2 and 3 were obtained in March-April 2001 on the Wisconsin Condor pool. We used a combination of PCs, some of which were running Linux and some Solaris. At the time, the pool contained a many relatively slow machines,

Table 3. SSN, with $N = 10{,}000$ scenarios, second trial on slower Condor pool, Algorithms TR and ATR.

Run	Points evaluated	$\|\mathcal{B}\|$ (K)	No. of tasks (T)	No. of clusters (C)	Max. processors allowed	Av. processors	Parallel efficiency	Max. no. of cuts in model	Master problem 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

and many were user-owned machines that were liable to be reclaimed by their users at random times, pre-empting our jobs. (In contrast, the runs reported in Tables 1, 4–6 were obtained in September 2002, when we drew only on Linux PCs, most of which were members of Linux clusters rather than individual user workstations, and were therefore less liable to preemption.)

Table 4. SSN trial on faster Condor pool, $N = 10{,}000$ scenarios, Algorithms TR and ATR.

Run	Points evaluated	$\lvert\mathcal{B}\rvert$ (K)	No. of tasks (T)	No. of clusters (C)	Max. processors allowed	Av. processors	Parallel efficiency	Max. no. of cuts in model	Master problem solve time (min)	Wall clock time (min)
TR	69	–	50	50	50	24	.54	3450	0.5	18.4
TR	45	–	50	100	50	16	.68	4072	0.5	14.3
TR	34	–	50	200	50	12	.65	6456	0.6	12.4
TR	49	–	100	100	100	13	.68	3731	0.5	16.5
TR	33	–	100	200	100	14	.57	5825	0.6	12.5
TR	33	–	200	200	200	16	.44	5096	0.5	13.5
ATR	154	3	50	50	100	28	.90	4688	1.4	19.5
ATR	115	3	50	100	100	23	.86	6787	1.3	16.5
ATR	105	3	50	200	100	20	.87	8946	1.7	14.8
ATR	90	3	100	100	200	20	.78	5818	1.0	17.6
ATR	93	3	100	200	200	21	.86	11043	1.9	15.5
ATR	84	3	200	200	200	30	.47	11135	1.9	18.6
ATR	218	6	50	50	150	36	.73	6111	3.0	26.5
ATR	199	6	50	100	150	31	.67	7999	3.6	24.6
ATR	152	6	50	200	150	31	.57	14645	5.9	25.8
ATR	186	6	100	100	200	41	.55	9551	3.8	26.0
ATR	129	6	100	200	200	29	.55	13284	3.8	24.8
ATR	144	6	200	200	200	45	.28	14602	5.0	33.2

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

Run	Points evaluated	$\lvert\mathcal{B}\rvert$ (K)	No. of tasks (T)	No. of clusters (C)	Max. processors allowed	Av. processors	Parallel efficiency	Max. no. of cuts in model	Master problem solve time (min)	Wall clock time (min)
TR	77	–	200	200	200	107	.39	15601	4.4	66.4
ATR	247	3	200	200	200	144	.72	27338	18.7	77.3

Table 6. Storm, with $N = 250{,}000$ scenarios.

Run	Points evaluated	$\lvert\mathcal{B}\rvert$ (K)	No. of tasks (T)	No. of clusters (C)	Max. processors allowed	Av. processors	Parallel efficiency	Max. no. of cuts in model	Master problem solve time (min)	Wall clock time (min)
TR	10	–	250	250	200	66	.63	2935	0.1	24.2
ATR	18	3	250	250	200	50	.85	4931	0.1	42.5

Since the Condor pool that we tapped in Tables 2 and 3 was identical, it is possible to do a meaningful comparison between corresponding lines of the two tables. Conditions on the Condor pool varied between and during each trial, leading to large variability of runtime from one trial to the next. Even for synchronous TR, the slightly different numerical values for function and subgradient value returned by different workers in different runs results in variations in the iteration sequence and therefore 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 are returned in different orders, so the information used by the trust-region subproblem (58) in generating the new point varies from run to run, and the resulting iteration sequences 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 for basket sizes $K = 3$ and $K = 6$ are competitive with the results obtained for the synchronous TR algorithm. However, for the large basket sizes, the loss of control induced by the increase in asynchronicity leads to a significant increase in the number of iterates, giving longer wall clock times even when more processors are available.

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 $T = 50$ and $C = 100$. 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.

Based on these observations, we make the following recommendation for users of ATR. If the computing platform consists of mostly heterogenous machines that are unlikely to fail, then a small basket size (or even the synchronous variant TR) is a good choice. If the computing platform is more variable and unreliable, we would recommend a slightly larger basket size (say, $K = 3$). We return to this point in discussing Table 4 below.

We note too that the larger number of clusters $C = 100$ gives slightly better wall clock times in general than the smaller choice $C = 50$, and significantly fewer iterations. The larger choices of number of tasks $T = 25$ and $T = 50$ also appear to be better in general than the smallest choice $T = 10$.

6.3.3. TR and ATR performance: Faster Condor pool. We now report on results obtained with TR and ATR in September 2002 on a higher-quality Condor pool than that used to obtain Tables 2 and 3. We set the parameters K, C, and T to values suggested by our

experience reported in those tables, choosing larger values of T and C than were used earlier, and focusing on the smaller values of K. As mentioned above, the Condor pool for this run consisted of generally faster and more reliable machines than were available in our earlier tests. Our results are shown in Table 4.

Note by comparing Table 4 to Tables 2 and 3 that the number of iterations is definitely smaller for larger values of C. The maximum number of cuts in the model increase with C, but even for $K = 6$, the total time required to solve the master problems is not too great a fraction of the total wall clock time. The most striking improvements in Table 4 as compared with the earlier Tables 2 and 3 is in the total wall clock times, which have decreased dramatically and become much more consistent as the quality of the Condor pool has improved. By comparing Table 4 with the results for ALS in Table 1, which were obtained at the same time in the same Condor environment, we see that that regularization strategy of Algorithm ATR leads to definite improvements in wall clock time and number of iterations, at least for Algorithm TR and for ATR with $K = 3$.

We note too that while the wall clock times for ATR with $K = 3$ are similar to those of the TR, the advantages of asynchronicity largely disappear when the prospect of pre-emption of the workers vanishes.

6.4. *Larger instances*

In this section, we report results on several larger instances of SSN (with $N = 100{,}000$ scenarios) and on some very large instances of the storm problem, described above [18]. Our interest in this section in the sheer size of the problems that can be solved using the algorithms developed for the computational grid, rather than with the relative performance of the algorithms with different parameter settings. Indeed, the computational grid is better suited to solving extremely large problem instances, rather than improving solution times on instances of a more moderate, fixed size.

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. We ran this experiment in September 2002, in the same Condor environment as used to obtain Table 4. We compared TR with 200 tasks and clusters, to ATR with the same values of T and C, and a basket size of 3. In both cases, the problem was solved in not much more than one hour. Although ATR required a much larger of iterations, its better parallel efficiency on the large number of workers available resulted in only a 16% degradation in wall clock time.

For the storm problem, we consider first a sampled approximation of this problem with 250,000 scenarios, which resulted in a linear program with 1.32×10^8 constraints and 3.15×10^8 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. In fact, the initial point is accepted after 10 iterations of TR and 18 iterations of ATR as the approximation solution, to within the required relative tolerance of 10^{-5}. TR required just 24 minutes of wall clock time, while the ATR run was somewhat slower, due to the smaller number of worker processors available and the time and its larger number of iterations.

Finally, we report on a very large sampled instance of storm with $N = 10^7$ scenarios, an instance whose deterministic equivalent is a linear program with 1.26×10^{10} variables.

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

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

This run was performed in February 2001 on a combined set of Condor pools from various locations at the University of Wisconsin with machines from Georgia Tech, the University of New Mexico/Albuquerque High Performance Computing Center, the Italian National Institute of Physics (INFN), the NCSA at the University of Illinois, and the Industrial Engineering and Operations Research Department at Columbia. Table 7 shows the number and type of processors available at each of these locations. In contrast to the other experiments reported in this paper, 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.

We used the tighter convergence tolerance $\epsilon_{\text{tol}} = 10^{-6}$ 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.

Performance is profiled in Table 8. 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 dropped

Table 8. Storm, with $N = 10^7$ scenarios.

Run	Points evaluated	$\lvert\mathcal{B}\rvert$ (K)	No. of tasks (T)	No. of clusters (C)	Max. processors allowed	Av. processors	Parallel efficiency	Max. no. of cuts in model	Master problem solve time (hr)	Wall clock time (hr)
ATR	38	4	1024	1024	800	433	.668	39647	1.9	31.9

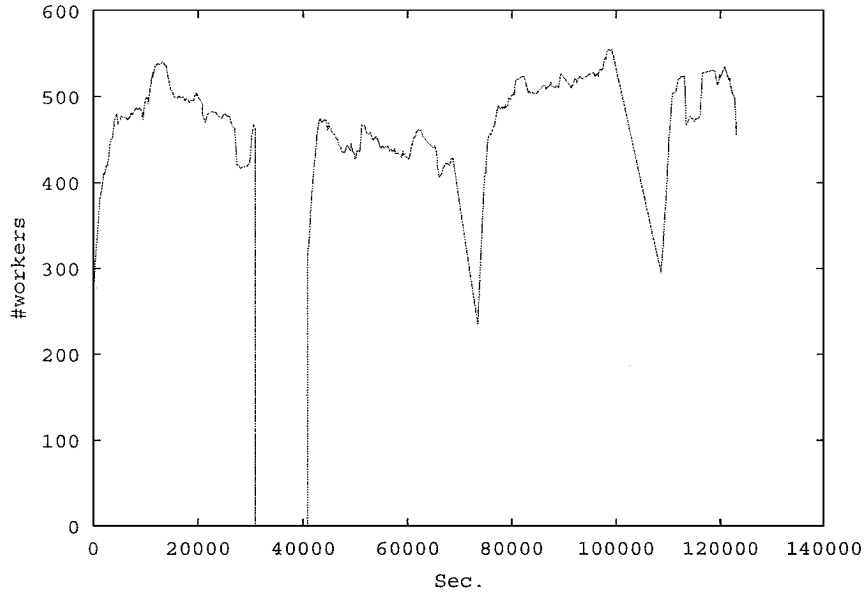

Figure 1. Number of workers used for stormG2, with $N = 10^7$ scenarios.

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 to other jobs.

As noted in Table 8, 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 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 of the worker pool was 9014 hours, or just over one year of computation.

6.5. *Comparisons with regularized decomposition code*

To verify that the algorithms and software described above do not contain gross inefficiencies, we solved some of the same test problems using the regularized decomposition (RD) code of Ruszczyński and Świetanowski [22], which is based on the algorithm described in [20]. This is a single-processor code that uses a quadratic penalty term in the master problem in place of the ℓ_∞ trust region of (16). It contains its own implementation of the LP simplex method for solving the second-stage problems, together with an efficient quadratic programming solver for the master problem that exploits its structure.

We modified the code in two ways. First, we removed the 5000-scenario limit, so that we could solve problems with a number of scenarios limited only by the capabilities of

Table 9. RD code performance on problem SSN.

N	Starting point	Iterations	Final objective	Time (min)
100	Cold start	34	8.1288875333	1.4
500	Warm start ($N = 100$)	58	9.29075190	7.4
10000	Warm start ($N = 500$)	100	9.98600225	238.3

the computer. Second, we gave the user the option of choosing a starting point, instead of accepting the initial point chosen by RD. We could therefore use the starting points for RD of similar quality to the ones used in the tests of our algorithms.

We ran RD on a 1.2 GHz Linux PC with 1GB of memory—the same machine as used for the master processor in the experiments reported in Tables 1, 4–6. Results for the SSN problem are shown in Table 9. We tried three randomly generated sampled approximations to SSN with 100, 500, and 10,000 scenarios, respectively. The latter two instances were warm-started from the computed solution of its predecessor in the table. We note that the number of iterations appears to grow slowly with N, and consequently the overall runtime is slightly superlinear in N. Without studying details of the respective strategies in RD and Algorithm TR for adjusting the regularization, generating and deleting the cuts, etc., it is difficult to draw a detailed comparison between these two approaches. By observing the iteration count for the $N = 10{,}000$ case, reported for Algorithm TR in the second column of the first 6 lines of Table 4, we see that TR's use of an ℓ_∞ trust region does not appear to degrade its effectiveness over the regularization term used by RD. The much faster wall clock times reported for Algorithm TR are of course chiefly attributable to parallel execution of the second-stage evaluations.

Results for RD on the storm problem are reported in Table 10. As pointed out in the discussion of Table 6, solutions obtained for relatively small N are near-optimal for larger N, making storm fundamentally different from SSN in this respect. For this problem, RD appears to require much longer runtimes than TR and ATR. It was not possible to solve instances of storm with RD for the value $N = 250{,}000$ used in Table 6 in a reasonable time. An attempted run of RD with $N = 100{,}000$ (see the last line of Table 10) was terminated after performing just three iterations in 14 hours. It is likely that RD's longer runtimes are due mainly to its internal simplex method implementation not being as efficient as the SOPLEX code used to solve the second-stage problems in our implementations of ALS, TR, and ATR. The respective iteration counts in Tables 6 and 10 again suggest that the regularization strategy used in our codes is as effective as that of RD.

Table 10. RD code performance on problem storm.

N	Starting point	Iterations	Final objective	Time (min)
1,000	Cold start	38	15508008.1	13.1
10,000	Warm start ($N = 1{,}000$)	41	15499432.7	255.6
100,000	Warm start ($N = 10{,}000$)	3*	–	877.0*

*Terminated prior to convergence.

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. Finally, we report on the solution of some large sampled instances of problems from the literature, including an instance of the storm problem whose deterministic equivalent has more than 10^{10} unknowns.

Because 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. In research carried out subsequent to this paper, Buaklee et al. [7] used a performance model to devise an optimal adaptive scheduling policy for the ATR code in a heterogeneous pool.

We have demonstrated that it is possible to solve very large instances of two-stage stochastic linear programming with recourse on potentially inexpensive parallel computational platforms. The tool we have developed will therefore be useful in obtaining high-quality solutions to difficult problems, by sampling a large number of scenarios from the available pool and/or by incorporating our solver in a sample-average approximation approach such as that of 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 Grants CDA-9726385, MTS-0086559, ACI-0196485, and EIA-0127857. 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 Instituto 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. We also thank the editors and referees for their comments on the first draft, which improved the final paper considerably.

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, vol. 69, pp. 45–73, 1995.
2. 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, vol. 17, pp. 1–19, 1987.

3. J.R. Birge, C.J. Donohue, D.F. Holmes, and O.G. Svintsiski, "A parallel implementation of the nested decomposition algorithm for multistage stochastic linear programs," Mathematical Programming, vol. 75, pp. 327–352, 1996.
4. J.R. Birge, C.J. Donohue, D.F. Holmes, and O.G. Svintsitski, "ND-UM Version 1.0: Computer Code for Nested Decomposition Algorithm," Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, Michigan 48109-2117, Aug. 1998.
5. J.R. Birge and F. Louveaux, Introduction to Stochastic Programming, Springer Series in Operations Research, Springer: Berlin, 1997.
6. J.R. Birge and L. Qi, "Computing block-angular karmarkar projections with applications to stochastic programming," Management Science, vol. 34, pp. 1472–1479, 1988.
7. D. Buaklee, G. Tracy, M. Vernon, and S.J. Wright, "An adaptive model to control a large scale application," in Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS 2002), June 2002.
8. CPLEX Optimization, Inc., Incline Village, NV, Using the CPLEX Callable Library, 1995.
9. E. Fragnière, J. Gondzio, and J.-P. Vial, "Building and solving large-scale stochastic programs on an affordable distributed computing system," Annals of Operations Research, vol. 99, pp. 167–187, 2000.
10. 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, Dec. 1997.
11. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, PVM: Parallel Virtual Machine, The MIT Press: Cambridge, MA, 1994.
12. J.-P. Goux, S. Kulkarni, J.T. Linderoth, and M. Yoder, "Master-Worker: An enabling framework for master-worker applications on the computational grid," Cluster Computing, vol. 4, pp. 63–70, 2001.
13. J.-P. Goux, J.T. Linderoth, and M. Yoder, "Metacomputing and the master-worker paradigm," Preprint MCS/ANL-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., Feb. 2000.
14. J.-B. Hiriart-Urruty and C. Lemaréchal, Convex Analysis and Minimization Algorithms II, Comprehensive Studies in Mathematics, Springer-Verlag: Berlin, 1993.
15. K. Kiwiel, "Proximity control in bundle methods for convex nondifferentiable minimization," Mathematical Programming, vol. 46, pp. 105–122, 1990.
16. M. Livny, J. Basney, R. Raman, and T. Tannenbaum, "Mechanisms for high throughput computing," SPEEDUP, vol. 11, 1997. Available from `http://www.cs.wisc.edu/condor/doc/htc_mech.ps`.
17. O.L. Mangasarian, Nonlinear Programming, McGraw-Hill: New York, 1969.
18. J.M. Mulvey and A. Ruszczyński, "A new scenario decomposition method for large scale stochastic optimization," Operations Research, vol. 43, pp. 477–490, 1995.
19. R.T. Rockafellar, Convex Analysis, Princeton University Press: Princeton, NJ, 1970.
20. A. Ruszczyński, "A regularized decomposition method for minimizing a sum of polyhedral functions," Mathematical Programming, vol. 35, pp. 309–333, 1986.
21. A. Ruszczyński, "Parallel decomposition of multistage stochastic programming problems," Mathematical Programming, vol. 58, pp. 201–228, 1993.
22. A. Ruszczyński and A. Świetanowski, "On the regularized decomposition method for two-stage stochastic linear problems," Working paper WP-96-014, IIASA, Laxenburg, Austria, 1996.
23. S. Sen, R.D. Doverspike, and S. Cosares, "Network planning with random demand," Telecommunications Systems, vol. 3, pp. 11–30, 1994.
24. A. Shapiro and T. Homem de Mello, "On the rate of convergence of optimal solutions of Monte Carlo approximations of stochastic programs," SIAM Journal on Optimization, vol. 11, pp. 70–86, 2000.
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, vol. 17, pp. 638–663, 1969.
26. R. Wunderling, "Paralleler und Objektorientierter Simplex-Algorithmus," Ph.D. Thesis, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1996.