CENTER FOR
PARALLEL OPTIMIZATION

ALTERNATING DIRECTION SPLITTINGS FOR BLOCK-ANGULAR PARALLEL OPTIMIZATION

by

Renato De Leone, Robert R. Meyer & Spyridon Kontogiorgis

Computer Sciences Technical Report #1217

February 1994

Alternating Direction Splittings For Block-Angular Parallel Optimization*

Renato De Leone† Robert R. Meyer† Spyridon Kontogiorgis†

Abstract: We develop and compare three decomposition algorithms derived from the method of alternating directions. They may be viewed as block Gauss-Seidel variants of augmented Lagrangian approaches that take advantage of block-angular structure. From a parallel computation viewpoint, they are ideally suited to a data parallel environment. Numerical results for large-scale multicommodity flow problems are presented to demonstrate the effectiveness of these decomposition algorithms on the Thinking Machines CM-5 parallel supercomputer relative to the widely-used serial optimization package MINOS 5.4 .

1 Introduction

Three decomposition algorithms are derived from the method of alternating directions [21] for block-angular optimization. The three methods will be categorized below according to the type of proximal terms that they contain. They may be viewed as block Gauss-Seidel variants of augmented Lagrangian approaches that take advantage of block-angular structure, and consequently are ideally suited for parallel computation in a data parallel environment. Numerical results are presented for the solution of multicommodity flow problems with these three techniques on the TMC CM-5 parallel supercomputer. As with many other decomposition methods (Dantzig-Wolfe decomposition [7], the Schultz-Meyer shifted barrier method [28], the Parallel Variable Distribution [13] and Parallel Constraint Distribution [12] algorithms of Ferris and Mangasarian, and the Diagonal Quadratic Approximation method of Mulvey and Ruszczyński [24]) these procedures use the *fork–join* protocol for the global organization of the computation.

In this computational model, there is a *data parallel phase* in which each processing element (PE) is assigned part of the current data and solves an optimization problem based on this data. When all PEs have completed the round, they combine solution information in a *global coordination* phase, a relatively simple process in the case of the three methods considered here, and one which, as we will see below, is efficiently performed by using special communication features of the CM-5. This is in contrast to decomposition methods that involve solving a complex optimization problem in the coordination phase, a potential serial bottleneck (for further discussion see [8]).

This article is organized as follows: section 2 introduces the convex block-angular problem and section 3 introduces the method of Alternating Directions (ADI). In section 4 we derive three highly-parallel algorithms that result from specializing the ADI method to the block-angular problem. We report computational experience on the CM-5 with multicommodity network flow problems in section 5. Finally, in section 6 we present two variants of the ADI method that can be used in deriving additional decomposition algorithms.

We describe here some of the mathematical notation we will be using. We use lower-case latin and greek letters for vectors and scalars, and capital letters for matrices and sets. All scalars, vectors and matrices are

*This material is based on research supported by the Air Force Office of Scientific Research Grants AFOSR-89-0410 and F49620-94-1-0036, and by NSF Grants CCR-8907671, CDA-9024618 and CCR-9306807

†Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin Madison, 1210 West Dayton Street, Madison, WI 53706.

real and of finite size. For any vector x and matrix A we denote the transposes by x^T and A^T, respectively. All untransposed vectors are assumed to be column vectors. We let $x^T y$ denote the Euclidean inner product of x and y, and we let $\|x\|_2$ stand for $(x^T x)^{\frac{1}{2}}$. The symbol $\|\cdot\|$ will represent an arbitrary norm. A sequence of vectors $x^1, x^2, \ldots$ will be denoted by $\{x^i\}$, with similar notation for sequences of matrices and scalars. We will reserve superscript t to denote vectors and matrices generated at step $t = 0, 1, 2, \ldots$ of an iterative process. A collection of vectors x with indices in a finite set $\{1, \ldots, K\}$ will be represented by $x_{[1]}, x_{[2]}, \ldots, x_{[K]}$. In case it is clear from the context, we sometimes let x represent the concatenation of the vectors $x_{[1]}, x_{[2]}, \ldots, x_{[K]}$. For scalar α, we define $\alpha_+ := \max\{\alpha, 0\}$. For vectors x and y, $\min\{x, y\}$ and $\max\{x, y\}$ are to be taken component-wise. Similarly, for a vector x, x_+ is defined component-wise. We let LCP (q, M) denote the following linear complementarity problem: given a vector q and a matrix M, find a non-negative vector x such that $Mx + q \geq 0$ and $x^T(Mx + q) = 0$.

The proofs of the lemmas and theorems presented in this article are given in [20].

2 The convex block-angular (CBA) problem

Our object of study is a convex minimization problem with linear constraints having a specific block structure in both the objective function and the constraint matrix. The problem, denoted by (**CBA**), is

$$\begin{array}{lllllll} \min\limits_{x_{[1]}, x_{[2]}, \ldots, x_{[K]}} & f_{[1]}(x_{[1]}) & + & f_{[2]}(x_{[2]}) & + \ \ldots \ + & f_{[K]}(x_{[K]}) & \\ \text{subject to} & A_{[1]}x_{[1]} & & & & = b_{[1]} & \\ & & A_{[2]}x_{[2]} & & & = b_{[2]} & \\ & & & \ddots & & & \\ & & & & A_{[K]}x_{[K]} & = b_{[K]} & \\ & D_{[1]}x_{[1]} & + & D_{[2]}x_{[2]} & + \ \ldots \ + & D_{[K]}x_{[K]} & \leq d \\ & & & 0 \leq x_{[i]} \leq u_{[i]} & & & i = 1, \ldots, K. \end{array} \tag{1}$$

We take each component function in the objective $f_{[i]}$ to be finite-valued, convex and continuous. In many important applications each function $f_{[i]}$ is block linear-quadratic (**LQ**) in the vector $x_{[i]}$, i.e.,

$$f_{[i]}(x_{[i]}) = {c_{[i]}}^T x_{[i]} + {x_{[i]}}^T Q_{[i]} x_{[i]}$$

in which $c_{[i]}$ is a real vector and $Q_{[i]}$ is a real symmetric positive semi-definite matrix. In this case the convex objective is also continuously differentiable. The constraint matrix has a special structure: block variables interact only in the coupling constraints, defined by the matrix $\left[D_{[1]} \ldots D_{[K]}\right]$ and the shared resource vector d. The quantity $D_{[i]}x_{[i]}$ is the amount of the vector d that activity $x_{[i]}$ consumes. It is the presence of the coupling constraints that makes the problem hard: if there were no such constraints, (**CBA**) would be fully decomposable per block, and the concatenation of optimal vectors for the block subproblems would be an optimal solution for (**CBA**) and vice versa. The vector $u_{[i]}$ is a component-wise upper bound on the levels of the activity $x_{[i]}$. If some components of the activity are allowed to grow without restriction, we take the corresponding components of $u_{[i]}$ to be $+\infty$.

Two particular classes of optimization problems that give rise to (**CBA**) and (**LQ**) problems are: multi-commodity network flow, which occurs in scheduling and transportation models [2], and scenario analysis, which arises in stochastic optimization [19], for financial planning under uncertainty over a multi-period horizon.

We make the basic assumption that *the block-angular problem (**CBA**) is solvable.*

Let $\mathcal{B}_{[i]}$, $i = 1, \ldots, K$ be the polyhedral sets representing the feasible region for the block constraints

$$\mathcal{B}_{[i]} := \{x_{[i]} \mid A_{[i]}x_{[i]} = b_{[i]} \text{ and } 0 \leq x_{[i]} \leq u_{[i]}\} \tag{2}$$

By our assumption the sets $\mathcal{B}_{[i]}$, $i = 1, \ldots, K$ are nonempty. However, feasibility is not enough to guarantee the existence of minimizers for (**CBA**), except for special cases, such as finiteness of all upper bounds $u_{[i]}$. For problem (**LQ**), if we assume that the feasible set is nonempty, we can guarantee the existence of minimizers if each $f_{[i]}$ is bounded from below on $\mathcal{B}_{[i]}$, by the Frank–Wolfe theorem [15].

We define for each block $i = 1, \ldots, K$ the extended objective function

$$h_{[i]}(x_{[i]}) := \begin{cases} f_{[i]}(x_{[i]}) & \text{if } A_{[i]}x_{[i]} = b_{[i]} \text{ and } 0 \leq x_{[i]} \leq u_{[i]} \\ \\ +\infty & \text{otherwise} \end{cases} \tag{3}$$

This function is proper, closed and convex, since it can be written as $h_{[i]}(x_{[i]}) = f_{[i]}(x_{[i]}) + \psi(\cdot \mid \mathcal{B}_{[i]})$ with $\psi(\cdot \mid \mathcal{B}_{[i]})$ the indicator function of the polyhedral set $\mathcal{B}_{[i]}$.

3 The method of Alternating Directions

In this section we introduce the Alternating Directions method for the minimization of the sum of two convex functions subject to linear equality constraints. In section 4 we show how to embed (**CBA**) in this problem in three ways, with each one producing a distinct iterative scheme.

The ADI method has been studied extensively and sufficient conditions for convergence have been obtained by either using the theory of maximal monotone operators (Lions and Mercier [21], Gabay [17], Eckstein and Bertsekas [10], [11]), or the theory of Karush–Kuhn–Tucker saddle-points of Lagrangian functions (Glowinski, Fortin and Le Tallec [14], [18]). In numerical analysis the ADI method is usually called the *Douglas–Rachford* method, because it corresponds to the method in [9] by these authors. This correspondence is discussed in detail in [17], [18], [21] and [10]. See also the monograph by Marchuk [23] for a discussion of splitting methods in the solution of pde's.

We want to solve the linearly-constrained convex problem

$$\begin{array}{ll} \min\limits_{x,\, z} & G_1(x) + G_2(z) \\ \text{subject to} & Ax + b = Bz \end{array} \tag{4}$$

where G_1 and G_2 are extended-real valued, proper, closed, convex functions, and A and B are nonzero matrices.

The Lagrangian function associated with problem (4) is

$$L_0(x, z, p) := G_1(x) + G_2(z) + p^T(Ax + b - Bz) \tag{5}$$

in which p is a tentative multiplier. From standard duality theory, if the Lagrangian admits a saddle-point at (x^*, z^*, p^*), i.e., if

$$L_0(x^*, z^*, p^*) = \min_{x,\, z} \max_{p} \; L_0(x, z, p)$$

then (x^*, z^*) is a solution of problem (4) and p^* is an associated dual multiplier vector.

Since the constraints are linear, the reverse convex constraint qualification is satisfied [22], and thus each minimizer of (4) corresponds to a saddle-point of the Lagrangian. These saddle-points coincide with the saddle-points of the augmented Lagrangian

$$L_\lambda\,(x, z, p) := G_1(x) + G_2(z) + p^T(Ax + b - Bz) + \frac{\lambda}{2}\,\|Ax + b - Bz\|_2^2 \tag{6}$$

in which λ is a positive scalar. The augmented Lagrangian function is computationally more stable than the pure Lagrangian [3]. In the augmented Lagrangian method the standard iteration consists of taking a minimization step in the primal space, followed by a steepest-ascent-type update of the multipliers p

$$\begin{aligned}(x^{t+1}, z^{t+1}) &\in \underset{x,\, z}{\operatorname{argmin}} \; L_\lambda\,(x, z, p^t) \\ p^{t+1} &= p^t + \nabla_p\, L_\lambda\,(x^{t+1}, z^{t+1}, p^t)\end{aligned}$$

In the ADI method the primal minimization is done in a block Gauss-Seidel fashion, first over the x and then over the z variables

$$\begin{aligned}x^{t+1} &\in \underset{x}{\operatorname{argmin}} \; L_\lambda\,(x, z^t, p^t) \\ z^{t+1} &\in \underset{z}{\operatorname{argmin}} \; L_\lambda\,(x^{t+1}, z, p^t) \\ p^{t+1} &= p^t + \lambda\, \nabla_p\, L_\lambda\,(x^{t+1}, z^{t+1}, p^t)\end{aligned}$$

Arbitrary vectors p^0 and z^0 are chosen as starting point. The scalar penalty factor $\lambda > 0$ remains fixed throughout the iterations. The method works towards achieving optimality in both the primal and the dual space, by taking alternating steps. See also Fukushima [16] for an application of the method to the dual, rather than the primal problem (4).

The rate of convergence is linear [17]. In order to reduce the number of iterations to optimality, we need to have the penalty term λ vary per iteration. An even better approach would be to use a separate value of λ for each linear constraint, also varying per iteration.

This results in the following extension of the ADI algorithm for problem (4)

$$x^{t+1} \in \underset{x}{\operatorname{argmin}} \; G_1(x) + p^{tT} Ax + (Ax + b - Bz^t)^T \Lambda^t (Ax + b - Bz^t) \tag{7}$$

$$z^{t+1} \in \underset{z}{\operatorname{argmin}} \; G_2(z) - p^{tT} Bz + (Ax^{t+1} + b - Bz)^T \Lambda^t (Ax^{t+1} + b - Bz) \tag{8}$$

$$p^{t+1} = p^t + \Lambda^t\,(Ax^{t+1} + b - Bz^{t+1}) \tag{9}$$

where Λ^t is the diagonal matrix of penalty terms.

In [20] the following theorem is proved, by employing saddle-point arguments similar to those in [18, chapter 3], or [5, chapter 3].

Theorem 3.1 (Convergence of ADI) *Assume that the original problem (4) and the ADI subproblems (7) and (8) are solvable. Let $\{(x^t, p^t, z^t)\}$ be some sequence of iterates produced by the ADI algorithm (7)–(9). If the entries of the diagonal positive matrices $\{\Lambda^t\}$ are bounded from above and bounded away from zero, then*
(i) The sequence $\{(Ax^t, Bz^t)\}$ converges to a feasible point for problem (4).
(ii) The sequence $\{G_1(x^t) + G_2(z^t)\}$ converges to the optimal objective for problem (4).
(iii) The sequence $\{p^t\}$ converges to an optimal dual multiplier for problem (4).
(iv) Any minimizers of subproblems of the form (7) and (8) in which p^t, Ax^{t+1} and Bz^t are fixed at their limit values are optimal for problem (4).

Notice that we do not require uniqueness of the minimizers in the subproblems.

In the existing literature convergence of the ADI iterates to an optimal point is proved under relatively strong assumptions on the problem: strict convexity and differentiability of G_1 and growth of G_1 faster than linear at infinity ([14], [18]). For the finite-dimensional case, A is assumed to have full column rank ([10], [11]) or G_1 is assumed to have compact level sets ([5, chapter 3]); for the dual algorithm in [16], both primal and dual problems are assumed to be feasible and the solution set of the primal is assumed to be bounded.

For the finite-dimensional setting, we have been able to relax these assumptions, and require only existence of Karush–Kuhn–Tucker points for problems (4), (7) and (8).

Previous researchers have concentrated on the use of a single penalty parameter, which is either fixed from the start, or ultimately fixed. Establishing convergence while allowing the penalty parameter to vary, as we do here, is essential for applications, since it allows for the incorporation of heuristics that can result in speeding convergence considerably.

In case we know more about the structure of the minimization problem we can strengthen the above theorem, by characterizing the behavior of the x^t and z^t iterates, as well.

Corollary 3.2 *Let the assumptions of theorem 3.1 hold, and let $G_1 + G_2$ be continuous. If A and B have full column rank, then*
(i) the subproblem minimizers $\{x^{t+1}\}$ and $\{z^{t+1}\}$ are uniquely defined.
(ii) the sequence $\{(x^t, z^t)\}$ converges to an optimal primal point for problem (4).

A weaker result is obtained if we make no assumptions on A and B but assume instead that the sequence $\{(x^t, z^t)\}$ has accumulation points.

Corollary 3.3 *Let the assumptions of theorem 3.1 hold, and let $G_1 + G_2$ be continuous. Then, any accumulation point $(\bar{x}, \bar{z})$ of $\{(x^t, z^t)\}$ is primal optimal for problem (4).*

Some sufficient conditions for the existence of accumulation points are: compactness of the effective domains of G_1 and G_2, or compactness of the nonempty level sets of $G_1 + G_2$.

4 Application to the Block-Angular Problem

We now specialize the ADI algorithm to the block-angular minimization problem (**CBA**). We map the block-angular problem onto problem (4), by specifying functions G_1 and G_2 so that the objective, variables and constraints of (**CBA**) are partitioned across G_1 and G_2. We are interested in splittings that exhibit high degree of block-separability, which can then be exploited in designing algorithms suitable for parallel computation.

The three splittings that we now present are such that the first subproblem (7) for each ADI iteration decomposes into independent block-subproblems, and the second ADI subproblem (8) has a very simple closed-form solution that can be computed in parallel. The iterates are always feasible with respect to the block constraints.

4.1 Activity and Resource Proximization (ARP)

4.1.1 Derivation

We augment problem (**CBA**) by introducing for each block a vector of extra variables $\check{d}_{[i]}$ that serves as an upper bound to the corresponding vector $D_{[i]}x_{[i]}$. Since $D_{[i]}x_{[i]}$ is the amount of the shared resource d consumed by activity $x_{[i]}$, the variable $\check{d}_{[i]}$ is an allocation of the shared resource to block i. Thus, if $\sum_{i=1}^{K} \check{d}_{[i]} \leq d$, then the activities $x_{[i]}$ are feasible with respect to the coupling constraints. In this splitting we have proximal terms for both the activities $x_{[i]}$ and the resource allocation vectors $\check{d}_{[i]}$.

Let

$$G_1\left(x_{[1]},\ldots,x_{[K]},\tilde{d}_{[1]},\ldots,\tilde{d}_{[K]}\right) := \begin{cases} \sum_{i=1}^{K} h_{[i]}(x_{[i]}) & \text{if } D_{[i]}x_{[i]} \leq \tilde{d}_{[i]},\ \forall i=1,\ldots,K \\ +\infty & \text{otherwise} \end{cases} \tag{10}$$

in which the extended block-objective function $h_{[i]}(x_{[i]})$ is as in (3). Let also

$$G_2\left(y_{[1]},\ldots,y_{[K]},d_{[1]},\ldots,d_{[K]}\right) := \begin{cases} 0 & \text{if } \sum_{i=1}^{K} d_{[i]} = d \\ +\infty & \text{otherwise} \end{cases} \tag{11}$$

Both functions G_1 and G_2 are closed, convex and also proper, because of our assumption that (**CBA**) is solvable. Problem (**CBA**) is equivalent to

$$\begin{aligned} \min_{x_{[i]},\tilde{d}_{[i]},y_{[i]},d_{[i]}} \quad & G_1\left(x_{[1]},\ldots,x_{[K]},\tilde{d}_{[1]},\ldots,\tilde{d}_{[K]}\right) + G_2\left(y_{[1]},\ldots,y_{[K]},d_{[1]},\ldots,d_{[K]}\right) \\ \text{subject to} \quad & \left.\begin{array}{c} x_{[i]} = y_{[i]} \\ \tilde{d}_{[i]} = d_{[i]} \end{array}\right\} \ i=1,\ldots,K \end{aligned} \tag{12}$$

which is in the form of problem (4), with the correspondences

$$\begin{array}{rclcrclcrcl} x & \leftarrow & [x_{[1]} & \ldots & x_{[K]}, & \tilde{d}_{[1]} & \ldots & \tilde{d}_{[K]}], & A & \leftarrow & I, \\ z & \leftarrow & [y_{[1]} & \ldots & y_{[K]}, & d_{[1]} & \ldots & d_{[K]}], & B & \leftarrow & I, \quad b \leftarrow 0 \end{array} \tag{13}$$

We associate a multiplier vector $q_{[i]}$ with each constraint $x_{[i]} = y_{[i]}$, and a multiplier vector $p_{[i]}$ with each constraint $\tilde{d}_{[i]} = d_{[i]}$. Each block $i = 1,\ldots,K$ maintains its own diagonal positive penalty matrix $\Lambda^t_{x[i]}$ for the proximization of the activities $x_{[i]}$. All blocks maintain the same penalty matrix Λ^t_d for the proximization of $\tilde{d}_{[i]}$, the allocation of the shared resource.

We now present the extended ADI algorithm for this splitting. We will use a shorthand notation, and write x^{t+1} for the concatenation of the vectors $x^{t+1}_{[1]},\ldots,x^{t+1}_{[K]}$, and similarly for y^{t+1} etc. We will also write $f(x)$ for $\sum_{i=1}^{K} f_{[i]}(x_{[i]})$. We also define $\Lambda^t_x := \text{diag}\left(\Lambda^t_{x[1]},\ldots,\Lambda^t_{x[K]}\right)$ and let the diagonal matrix Λ^t_K consist of K copies of Λ^t_d placed along the diagonal.

The algorithm starts from any q^0, p^0, y^0, d^0 and any positive diagonal penalty matrices $\Lambda^0_{x[i]}$ and Λ^0_d. At iteration t we solve the two subproblems

$$\begin{aligned} (x^{t+1},\tilde{d}^{t+1}) = \ \underset{x,\,d}{\text{argmin}} \quad & f(x) + q^{tT}x + p^{tT}d + \frac{1}{2}(x-y^t)^T\Lambda^t_x(x-y^t) + \frac{1}{2}(d-d^t)^T\Lambda^t_d(d-d^t) \\ \text{s. t.} \quad & \left.\begin{array}{c} A_{[i]}x_{[i]} = b_{[i]} \\ D_{[i]}x_{[i]} \leq d_{[i]} \\ 0 \leq x_{[i]} \leq u_{[i]} \end{array}\right\} \ i=1,\ldots,K \end{aligned} \tag{14}$$

$$(y^{t+1}, d^{t+1}) = \begin{array}{ll} \underset{y,\, d}{\text{argmin}} & -q^{t^T} y - p^{t^T} d + \frac{1}{2}(y - x^{t+1})^T \Lambda_x^t (y - x^{t+1}) + \frac{1}{2}(d - \tilde{d}^{t+1})^T \Lambda_K^t (d - \tilde{d}^{t+1}) \\ \text{s. t.} & \sum_{i=1}^{K} d_{[i]} = d \end{array} \tag{15}$$

Then we update the multipliers

$$q^{t+1} = q^t + \Lambda_x^t \, (x^{t+1} - y^{t+1}) \tag{16}$$

$$p^{t+1} = p^t + \Lambda_K^t \, (\tilde{d}^{t+1} - d^{t+1}) \tag{17}$$

and possibly the penalty matrices Λ_x^t and Λ_d^t.

We now show how to simplify and parallelize the iterative step. In the first subproblem the objective and the constraints are fully decomposable per block. Thus we can solve the following K block-subproblems in parallel and then concatenate their solutions.

$$\begin{array}{ll} \underset{x_{[i]},\, d_{[i]}}{\min} & f_{[i]}(x_{[i]}) + q_{[i]}^{t^T} x_{[i]} + p_{[i]}^{t^T} d_{[i]} + \frac{1}{2}(x_{[i]} - y_{[i]}^t)^T \Lambda_{x[i]}^t (x_{[i]} - y_{[i]}^t) + \frac{1}{2}(\tilde{d}_{[i]} - d_{[i]}^t)^T \Lambda_d^t (\tilde{d}_{[i]} - d_{[i]}^t) \\ \text{s. t.} & A_{[i]} x_{[i]} = b_{[i]} \\ & D_{[i]} x_{[i]} \leq d_{[i]} \\ & 0 \leq x_{[i]} \leq u_{[i]} \end{array} \tag{18}$$

We now rename the vectors $\tilde{d}^{t+1}$ as $d^{t+\frac{1}{2}}$. The reason for this is that we want the superscript $t+1$ to designate the vectors that need to be passed from iteration t to the next. While the algorithm, as presented, appears to require six vectors per iteration, i.e. x^t, d^t, y^t, $\tilde{d}^t$, q^t and p^t (in the notation of (14)–(17)), we will show that only three of these are actually needed: x^t, d^t and p^t. Using the notation $d^{t+\frac{1}{2}}$ for the d-vector that minimizes subproblem (18) emphasizes the fact that it is an intermediate quantity that need not be passed from an iteration to the next.

Let now $(x^{t+1}, d^{t+\frac{1}{2}})$ be the solution of the block subproblems (18). A solution in closed form for subproblem (15) can be obtained by solving the Karush–Kuhn–Tucker conditions for it. This yields per block

$$y_{[i]}^{t+1} = x_{[i]}^{t+1} + (\Lambda_{x[i]}^t)^{-1} q_{[i]}^t \tag{19}$$

and

$$d_{[i]}^{t+1} = d_{[i]}^{t+\frac{1}{2}} + (\Lambda_d^t)^{-1} \left(p_{[i]}^t - \frac{1}{K} \sum_{i=1}^{K} p_{[i]}^t \right) + \frac{1}{K} \left(d - \sum_{i=1}^{K} d_{[i]}^{t+\frac{1}{2}} \right) \tag{20}$$

We observe that the second subproblem has a closed-form solution that is computationally very simple. Thus, the computationally intensive part is the solution of subproblems (18), which can be carried out in parallel.

The iterates in this scheme have a number of interesting properties.

Lemma 4.1 (Invariants of the (ARP) splitting) *Let the ADI scheme for the (**ARP**) splitting be started from any q^0, p^0, y^0, d^0 and any positive diagonal penalty matrices $\Lambda_{x[i]}^0$ and Λ_d^0. Then, the iterates are such that:*
(i) $q^{t+1} = 0$, $t \geq 0$.
(ii) $y^{t+1} = x^{t+1}$, $t \geq 1$
(iii) $\sum_{i=1}^{K} d_{[i]}^{t+1} = d$, $t \geq 0$
(iv) $p_{[1]}^{t+1} = p_{[2]}^{t+1} = \ldots = p_{[K]}^{t+1}$, $t \geq 0$

We will now derive some explicit formulas for the update of the p and d vectors that are equivalent to those in (20) and (17). One of these will use dual information from the first subproblem.

Lemma 4.2 *Define* $v^t_{[i]} := p^t_{[i]} + \Lambda^t_d \left(d^{t+\frac{1}{2}}_{[i]} - d^t_{[i]}\right)$. *Then,* $v^t_{[i]}$ *is an optimal (non-negative) dual vector associated with the* $D_{[i]}x_{[i]} \leq d_{[i]}$ *constraints in the subproblem (18).*

Lemma 4.3 (Updates for d^t) *Let the ADI scheme for the (**ARP**) splitting be started from any* q^0, p^0, y^0, d^0 *and any positive diagonal penalty matrices* $\Lambda^0_{x[i]}$ *and* Λ^0_d. *Then, the following updates are equivalent to that in (20)*

$$d^{t+1}_{[i]} = d^{t+\frac{1}{2}}_{[i]} + (\Lambda^t_d)^{-1}\left(p^t_{[i]} - p^{t+1}_{[i]}\right), \quad t \geq 0 \tag{21}$$

$$d^{t+1}_{[i]} = d^t_{[i]} + (\Lambda^t_d)^{-1}\left(v^t_{[i]} - p^{t+1}_{[i]}\right), \quad t \geq 0 \tag{22}$$

$$d^{t+1}_{[i]} = d^{t+\frac{1}{2}}_{[i]} + \frac{1}{K}\left(d - \sum_{i=1}^{K} d^{t+\frac{1}{2}}_{[i]}\right), \quad t \geq 1 \tag{23}$$

Lemma 4.4 (Multipliers p are averages) *Let the ADI algorithm for the (**ARP**) splitting be initialized from any* q^0, p^0, y^0, d^0 *and any positive diagonal penalty matrices* $\Lambda^0_{x[i]}$ *and* Λ^0_d. *Then, the following* p *updates are equivalent to that in (17).*

$$p^{t+1}_{[i]} = p^t_{[i]} - \frac{1}{K}\Lambda^t_d\left(d - \sum_{i=1}^{K} d^{t+\frac{1}{2}}_{[i]}\right), \quad t \geq 1 \tag{24}$$

$$p^{t+1}_{[i]} = \frac{1}{K}\sum_{i=1}^{K} v^t_{[i]}, \quad t \geq 1 \tag{25}$$

The last equation also shows that the multipliers $p^t_{[i]}$, $i = 1, \ldots, K$ are non-negative, for $t \geq 2$, as averages of the non-negative duals $v^t_{[i]}$.

The properties of the iterates, described in the previous lemmas, allow us to construct an iteration-by-iteration description of the algorithm, as follows: We choose initial arbitrary proximal terms $y^0_{[i]}$ and $d^0_{[i]}$ and multipliers $q^0_{[i]}$ and $p^0_{[i]}$. The first iteration produces vectors $y^1_{[i]}$, $d^1_{[i]}$, $q^1_{[i]}$ and $p^1_{[i]}$. These are such that $\sum_{i=1}^{K} d^1_{[i]} = d$, the multipliers $q^1_{[i]}$ are zero, and the $p^1_{[i]}$ are equal across the blocks. Moreover, these properties will now hold for the q^t, d^t and p^t vectors produced by all subsequent iterations. The second iteration will produce non-negative multipliers $p^2_{[i]}$. This will also be true for all subsequent iterations. The third iteration and all subsequent ones will have in the first subproblem as proximal term for each activity $x_{[i]}$ the optimal level of this activity in the subproblem of the previous iteration.

An appropriate choice of the initial vectors allows us to have these properties established one iteration earlier. We call such a choice a canonical initialization.

Definition 4.5 *An initialization* d^0, q^0, p^0 *for the ADI scheme for (**ARP**) is* CANONICAL *if*

$$q^0 = 0, \;\; p^0_{[1]} = p^0_{[2]} = \ldots = p^0_{[K]} \geq 0, \;\; \sum_{i=1}^{K} d^0_{[i]} = d$$

4.1.2 The algorithm simplified

We now make use of the results of the previous section and of canonical initialization in order to present a simpler version of the iterative step: we pass only three vectors from an iteration to the next, instead of the original six. In the following description we denote by p^t the common value of the p multiplier across all blocks.

(0) Pick a non-negative vector p^0 and proximal vectors $x^0_{[i]}$ and $d^0_{[i]}$ $i = 1, \ldots, K$, such that $\sum_{i=1}^{K} d^0_{[i]} = d$. Also pick diagonal positive matrices $\Lambda^0_{x[i]}$ and Λ^0_d. Set $t = 0$.

(1) Compute (in parallel) for each block $i = 1, \ldots, K$, $(x^{t+1}_{[i]}, d^{t+\frac{1}{2}}_{[i]})$, the solution to

$$\begin{aligned} \min_{x_{[i]}, d_{[i]}} \quad & f_{[i]}(x_{[i]}) + {p^t_{[i]}}^T d_{[i]} + \frac{1}{2}(x_{[i]} - x^t_{[i]})^T \Lambda^t_{x[i]}(x_{[i]} - x^t_{[i]}) + \frac{1}{2}(d_{[i]} - d^t_{[i]})^T \Lambda^t_d (d_{[i]} - d^t_{[i]}) \\ \text{s. t.} \quad & A_{[i]} x_{[i]} = b_{[i]} \\ & D_{[i]} x_{[i]} \leq d_{[i]} \\ & 0 \leq x_{[i]} \leq u_{[i]} \end{aligned} \tag{26}$$

(2) Adjust allocations to achieve feasibility

$$d^{t+1}_{[i]} = d^{t+\frac{1}{2}}_{[i]} + \frac{1}{K}\left(d - \sum_{i=1}^{K} d^{t+\frac{1}{2}}_{[i]}\right) \tag{27}$$

(3) Update the multipliers p

$$p^{t+1} = p^t - \frac{1}{K}\Lambda^t_d \left(d - \sum_{i=1}^{K} d^{t+\frac{1}{2}}_{[i]}\right) \tag{28}$$

(4) Update the penalty factors $\Lambda^t_{x[i]}$ and Λ^t_d.
(5) If termination criteria are met, stop. Else set $t = t + 1$ and go to (1).

Notice that the d and p updates involve just the summation of vectors over all blocks i; after all these values are distributed, the optimization problems in step (1) above are solved independently.

4.1.3 Convergence of the algorithm

The following two theorems address the convergence of this ADI method, for primal and dual variables, respectively.

Theorem 4.6 (Convergence of (ARP) splitting) *Assume that (**CBA**) is solvable. Let the ADI method be started from arbitrary q^0, p^0, y^0, d^0 and arbitrary diagonal positive matrices Λ^0_x, Λ^0_d, and let the sequences $\{\Lambda^t_x\}$ and $\{\Lambda^t_d\}$ be bounded above and bounded away from zero. Then, the sequence of ADI iterates $\{x^t\}$ converges to an optimal primal solution for (**CBA**).*

By the general ADI convergence theorem 3.1 we also get that $\{p^t\}$ converges to p^*, an optimal multiplier for the $d_{[i]} = \tilde{d}_{[i]}$ constraints of the extended problem (12). In the case where (**CBA**) has a differentiable objective, if we assume that the sequence of duals for the first subproblem has an accumulation point, we can show that the algorithm also generates a sequence of duals that converges to a set of optimal duals for (**CBA**), with p^* an optimal dual for the coupling constraints.

Theorem 4.7 (Convergence of duals) *Assume that $f_{[i]}$, $i = 1, \ldots, K$ is differentiable. Let the optimal dual multipliers for subproblems (26) be paired with the constraints as follows:*

$$\begin{array}{llllll} w^t_{[i]} & \text{with} & A_{[i]}x_{[i]} = b_{[i]}, & v^t_{[i]} & \text{with} & D_{[i]}x_{[i]} \leq d_{[i]} \\ r^t_{[i]} & \text{with} & -x_{[i]} \leq 0, & s^t_{[i]} & \text{with} & x_{[i]} - u_{[i]} \leq 0 \end{array}$$

*If the sequence $\{(w^t, r^t, s^t)\}$ has an accumulation point $(\bar{w}, \bar{r}, \bar{s})$, then $(\bar{w}, p^*, \bar{r}, \bar{s})$ are optimal duals for (**CBA**), for the pairing*

$$\begin{array}{llllll} \bar{w}_{[i]} & \text{with} & A_{[i]}x_{[i]} = b_{[i]}, & p^* & \text{with} & \sum_{i=1}^{K} D_{[i]}x_{[i]} \leq d \\ \bar{r}_{[i]} & \text{with} & -x_{[i]} \leq 0, & \bar{s}_{[i]} & \text{with} & x_{[i]} - u_{[i]} \leq 0 \end{array}$$

in which p^* is the limit of $\frac{1}{K}\sum_{i=1}^{K} v^t_{[i]}$.

4.2 Resource Proximization (RP)

4.2.1 Derivation

The (**ARP**) splitting we just examined has a number of desirable features: a block decomposable first subproblem and a closed-form solution of the second subproblem that was very simple to compute. The cost of these features was the need to introduce a vector of extra variables $\tilde{d}_{[i]}$ for each block, and thus augment the size of the computationally intensive nonlinear first subproblem.

In the splitting we are about to examine we do not introduce extra variables yet we are still able to maintain the attractive features described above. We add proximal terms only for the vectors $D_{[i]}x_{[i]}$ that reflect the consumption of the shared resource. The trade-off is that we cannot guarantee convergence of the full set of primal variables, although the objective value converges to the optimal objective.

We define

$$G_1\left(x_{[1]}, \ldots, x_{[K]}\right) := \sum_{i=1}^{K} h_{[i]}(x_{[i]}) \tag{29}$$

with the extended objective function $h_{[i]}(x_{[i]})$ as in (3), and

$$G_2\left(d_{[1]}, \ldots, d_{[K]}\right) := \begin{cases} 0 & \text{if } \sum_{i=1}^{K} d_{[i]} \leq d \\ +\infty & \text{otherwise} \end{cases} \tag{30}$$

Both functions G_1 and G_2 are closed, convex and also proper, because of our assumption that (**CBA**) is solvable.

The splitting matrix $\mathcal{D}$ is block-diagonal, $\mathcal{D} := \text{diag}\left(D_{[1]}, D_{[2]}, \ldots, D_{[K]}\right)$. Problem (**CBA**) is equivalent to

$$\begin{array}{ll} \min\limits_{x_{[i]},\, d_{[i]}} & G_1\left(x_{[1]}, \ldots, x_{[K]}\right) + G_2\left(d_{[1]}, \ldots, d_{[K]}\right) \\ \text{subject to} & D_{[i]}x_{[i]} = d_{[i]}, \;\; i = 1, \ldots, K \end{array} \tag{31}$$

which is in the form of problem (4), with the correspondences

$$\begin{array}{rclrcl} x & \leftarrow & [x_{[1]} \;\ldots\; x_{[K]}], & A & \leftarrow & \mathcal{D}, \\ z & \leftarrow & [d_{[1]} \;\ldots\; d_{[K]}], & B & \leftarrow & I, \quad b \;\leftarrow\; 0 \end{array} \tag{32}$$

If certain variables in block i do not appear in the coupling constraints, then the corresponding columns of $D_{[i]}$ are zeros, and in this case the matrix $\mathcal{D}$ is not of full column rank.

We pair a tentative Lagrange multiplier vector $p_{[i]}$ with each block of constraints $D_{[i]}x_{[i]} = d_{[i]}$, $i = 1, \ldots, K$. We also use a diagonal positive penalty matrix Λ^t, common to all blocks. We let the diagonal matrix Λ^t_K consist of K copies of Λ^t placed along the diagonal. Then, for this splitting the iterative step of the extended ADI algorithm (in shorthand notation) consists of solving the two subproblems

$$x^{t+1} \in \begin{array}{ll} \underset{x}{\text{argmin}} & f(x) + p^{tT}Dx + \frac{1}{2}(Dx - d^t)^T \Lambda^t_K (Dx - d^t) \\ \text{subject to} & \left.\begin{array}{c} A_{[i]}x_{[i]} = b_{[i]} \\ 0 \le x_{[i]} \le u_{[i]} \end{array}\right\} \; i = 1, \ldots, K \end{array} \tag{33}$$

$$d^{t+1} = \begin{array}{ll} \underset{d}{\text{argmin}} & -p^{tT}d + \frac{1}{2}(Dx^{t+1} - d)^T \Lambda^t (Dx^{t+1} - d) \\ \text{subject to} & \sum_{i=1}^{K} d_{[i]} \le d \end{array} \tag{34}$$

and then updating the multipliers

$$p^{t+1} = p^t + \Lambda^t \left(Dx^{t+1} - d^{t+1}\right) \tag{35}$$

We now proceed to analyze and simplify this algorithm. The first subproblem is again decomposable per block. Thus we need to solve K subproblems in parallel and concatenate the solutions.

$$\begin{array}{ll} \min\limits_{x_{[i]}} & f_{[i]}(x_{[i]}) + p_{[i]}^{tT} D_{[i]}x_{[i]} + \frac{1}{2}(D_{[i]}x_{[i]} - d^t_{[i]})^T \Lambda^t (D_{[i]}x_{[i]} - d^t_{[i]}) \\ \text{subject to} & A_{[i]}x_{[i]} = b_{[i]} \\ & 0 \le x_{[i]} \le u_{[i]} \end{array} \tag{36}$$

The second subproblem is a least squares problem with linear constraints. A solution in closed form can be obtained from the Karush–Kuhn–Tucker conditions. The minimizer d^{t+1} must satisfy

$$d^{t+1}_{[i]} = D_{[i]}x^{t+1}_{[i]} + (\Lambda^t)^{-1}\left(p^t_{[i]} - \mu^{t+1}\right) \tag{37}$$

where μ^{t+1} is given by

$$\mu^{t+1} = \frac{1}{K}\left[\sum_{i=1}^{K} p^t_{[i]} + \Lambda^t\left(\sum_{i=1}^{K} D_{[i]}x^{t+1}_{[i]} - d\right)\right]_+ \tag{38}$$

Finally, the p multiplier update simplifies to

$$p_{[i]}^{t+1} = \mu^{t+1} \tag{39}$$

This iterative scheme possesses invariants similar to those for the (**ARP**) splitting.

Lemma 4.8 (Invariants of the (RP) splitting) *Let the ADI scheme (33)–(35) be started from any p^0, d^0 and any diagonal positive matrix Λ^0. Then, for $t \geq 0$,*
(i) $p_{[1]}^{t+1} = p_{[2]}^{t+1} = \ldots = p_{[K]}^{t+1} \geq 0$
(ii) $\sum_{i=1}^{K} d_{[i]}^{t+1} \leq d$

We can have these properties established one iteration earlier, by a canonical initialization.

Definition 4.9 *An initialization p^0, d^0 for the ADI scheme for (**RP**) is* CANONICAL *if*

$$p_{[1]}^0 = p_{[2]}^0 = \ldots = p_{[K]}^0 \geq 0, \ \sum_{i=1}^{K} d_{[i]}^0 = d$$

4.2.2 The algorithm simplified

By making use of the results of the previous section and of canonical initialization we now present a simpler version of the algorithm. We let p^t represent the common value of the current multiplier across all blocks.

(0) Pick a non-negative vector p^0, proximal vectors $d_{[i]}^0$, $i = 1, \ldots, K$ such that $\sum_{i=1}^{K} d_{[i]}^0 = d$, and a diagonal positive matrix Λ^0. Set $t = 0$.
(1) Compute (in parallel) per block $i = 1, \ldots, K$, $x_{[i]}^{t+1}$, a solution to

$$\begin{array}{ll} \min\limits_{x_{[i]}} & f_{[i]}(x_{[i]}) + p^{tT} D_{[i]} x_{[i]} + \frac{1}{2}(D_{[i]} x_{[i]} - d_{[i]}^t)^T \Lambda^t (D_{[i]} x_{[i]} - d_{[i]}^t) \\ \text{subject to} & A_{[i]} x_{[i]} = b_{[i]} \\ & 0 \leq x_{[i]} \leq u_{[i]} \end{array} \tag{40}$$

(2) Update the multipliers p

$$p^{t+1} = \left\{ p^t + \frac{1}{K} \Lambda^t \left(\sum_{i=1}^{K} D_{[i]} x_{[i]}^{t+1} - d \right) \right\}_+ \tag{41}$$

(3) Adjust allocations to achieve feasibility

$$d_{[i]}^{t+1} = D_{[i]} x_{[i]}^{t+1} + (\Lambda^t)^{-1} \left(p^t - p^{t+1} \right) \tag{42}$$

(4) Update the penalty matrix Λ^t.
(5) If termination criteria are met, stop. Else set $t = t + 1$ and go to (1).

As in the previous splitting, the updates are very simple to compute; they involve matrix-vector and vector-vector operations on local data, with the exception of the global quantity $\sum_{i=1}^{K} D_{[i]}x_{[i]}^{t+1}$ that requires communication between all blocks.

4.2.3 Convergence of the algorithm

For this method we have a weaker convergence result than the one for the (**ARP**) splitting. This is because the splitting matrix $\mathcal{D}$ may not have full column rank.

Theorem 4.10 (Convergence of (RP) splitting) *Assume that (**CBA**) is solvable. Let the ADI method be started from arbitrary p^0, d^0 and arbitrary diagonal positive matrix Λ^0, and let the sequence $\{\Lambda^t\}$ be bounded above and bounded away from zero. Then, a sequence $\{x^t, d^t\}$ produced by the ADI algorithm for the (**RP**) splitting is such that*

(i) $\left\{D_{[i]}x_{[i]}^t\right\}$, $i = 1, \ldots, K$, converges to $d_{[i]}^*$ such that $\sum_{i=1}^{K} d_{[i]}^* \leq d$.

*(ii) $\sum_{i=1}^{K} f_{[i]}(x_{[i]}^t)$ converges to the optimal value for (**CBA**).*

(iii) For all $i \in \{1, \ldots, K\}$ such that $D_{[i]}$ has full column rank, the sequence $\left\{x_{[i]}^t\right\}$ converges.

4.2.4 The connection between the (ARP) and (RP) splittings

The algorithm we developed for the (**RP**) splitting can be thought of as a limiting case for an (**ARP**) algorithm in which all proximal terms for the $x_{[i]}$ activities have been set to zero. Then the $D_{[i]}x_{[i]}$ terms for (**RP**) behave like $d_{[i]}^{t+\frac{1}{2}}$ terms for this (**ARP**).

In more detail: we derive an (**ARP**) splitting by defining a function G_1 as

$$G_1\left(x_{[1]}, \ldots, x_{[K]}, \check{d}_{[1]}, \ldots, \check{d}_{[K]}\right) := \begin{cases} \sum_{i=1}^{K} f_{[i]}(x_{[i]}) + \psi(\cdot \mid \mathcal{B}_{[i]})(x_{[i]}) & \text{if } D_{[i]}x_{[i]} = \check{d}_{[i]},\ \forall i \\ +\infty & \text{otherwise} \end{cases}$$

and by defining a function G_2 of the $d_{[i]}$ variables only

$$G_2\left(d_{[1]}, \ldots, d_{[K]}\right) := \begin{cases} 0 & \text{if } \sum_{i=1}^{K} d_{[i]} \leq d \\ +\infty & \text{otherwise} \end{cases} \tag{43}$$

Then, an equivalent formulation of the (**CBA**) problem is

$$\begin{array}{ll} \min\limits_{x_{[i]}, \check{d}_{[i]}, d_{[i]}} & G_1\left(x_{[1]}, \ldots, x_{[K]}, \check{d}_{[1]}, \ldots, \check{d}_{[K]}\right) + G_2\left(d_{[1]}, \ldots, d_{[K]}\right) \\ \text{subject to} & d_{[i]} = \check{d}_{[i]},\ \ i = 1, \ldots, K \end{array} \tag{44}$$

in which the $x_{[i]}$ variables are not included in the explicit constraints.

In terms of correspondences with problem (4), this splitting has

$$\begin{array}{rcl} x & \leftarrow & [x_{[1]} \ \cdots \ x_{[K]}, \ \tilde{d}_{[1]} \ \cdots \ \tilde{d}_{[K]}], \quad A \ \leftarrow \ [0 \ \ I], \\ z & \leftarrow & [y_{[1]} \ \cdots \ y_{[K]}, \ d_{[1]} \ \cdots \ d_{[K]}], \quad B \ \leftarrow \ I, \qquad b \ \leftarrow \ 0 \end{array} \tag{45}$$

The splitting matrix A has not full column rank now; moreover, multipliers and primal proximal terms are associated only with the $\tilde{d}$ variables.

After simplification, the extended ADI algorithm for this choice of G_1 and G_2 requires solving, at each iteration, the subproblems

$$\begin{array}{ll} \min\limits_{x_{[i]}, d_{[i]}} & f_{[i]}(x_{[i]}) + {p_{[i]}^t}^T d_{[i]} + \frac{1}{2}(d - d^t)^T \Lambda_d^t (d - d^t) \\ \text{subject to} & A_{[i]} x_{[i]} = b_{[i]} \\ & D_{[i]} x_{[i]} = d_{[i]} \\ & 0 \leq x_{[i]} \leq u_{[i]} \end{array} \tag{46}$$

to obtain the $x_{[i]}^{t+1}$ and $d_{[i]}^{t+\frac{1}{2}}$ vectors. Then we update in reverse order, by

$$p^{t+1} = \left(p^t - \frac{1}{K} \Lambda_d^t \left[d - \sum_{i=1}^{K} d_{[i]}^{t+\frac{1}{2}} \right] \right)_+ \tag{47}$$

and

$$d_{[i]}^{t+1} = d_{[i]}^{t+\frac{1}{2}} + (\Lambda_d^t)^{-1} \left(p_{[i]}^t - p_{[i]}^{t+1} \right) \tag{48}$$

Now, if we let $D_{[i]} x_{[i]}^{t+1} = d_{[i]}^{t+\frac{1}{2}}$, the subproblem in (46) above is comparable to the subproblem in (36) for the (**RP**) splitting, while the updates in (47) and (48) are comparable to those in (41) and (42).

4.3 Activity Proximization (AP)

4.3.1 Derivation

In the (**RP**) splitting of the previous section the number of proximal terms introduced per block equals the number of rows in the matrix describing the coupling constraints. In general this number is less than the number of block variables, and thus the subproblem objective is not strongly convex and may lack desirable properties such as uniqueness of minimizers. Also convergence is not guaranteed, in general, for the set of primal variables.

To address these shortcomings, we now present a splitting that introduces exactly one proximal term per block variable, yet no extra variables for the resource allocation are introduced, in contrast with the (**ARP**) splitting of section 4.1. In this scheme we add proximal terms only for the activity vectors $x_{[i]}$.

We define, as in the (**RP**) splitting,

$$G_1 \left(x_{[1]}, \ldots, x_{[K]} \right) := \sum_{i=1}^{K} h_{[i]}(x_{[i]}) \tag{49}$$

with the extended objective function $h_{[i]}(x_{[i]})$ as in (3). The difference with the (**RP**) splitting lies in the definition of G_2. We take here

$$G_2 \left(y_{[1]}, \ldots, y_{[K]} \right) := \begin{cases} 0 & \text{if } \sum\limits_{i=1}^{K} D_{[i]} y_{[i]} \leq d \\ +\infty & \text{otherwise} \end{cases} \tag{50}$$

Both functions G_1 and G_2 are closed, convex and also proper, because of our assumption that (**CBA**) is solvable. We can write problem (**CBA**) as

$$\begin{array}{ll} \min\limits_{x_{[i]},\, y_{[i]}} & G_1\left(x_{[1]},\ldots,x_{[K]}\right) + G_2\left(y_{[1]},\ldots,y_{[K]}\right) \\ \text{subject to} & x_{[i]} = y_{[i]},\ \ i = 1,\ldots,K \end{array} \tag{51}$$

which is in the form of problem (4), with the correspondences

$$\begin{array}{rclcrcl} x & \leftarrow & [x_{[1]} \ \ldots \ x_{[K]}], & A & \leftarrow & I, & \\ z & \leftarrow & [y_{[1]} \ \ldots \ y_{[K]}], & B & \leftarrow & I, & b \ \leftarrow \ 0 \end{array} \tag{52}$$

A tentative Lagrange multiplier vector $p_{[i]}$ is associated with each block of constraints $x_{[i]} = y_{[i]}$, $i = 1,\ldots,K$. We let each block i maintain its own diagonal positive penalty matrix $\Lambda^t_{[i]}$. We define $\Lambda^t := \text{diag}\left(\Lambda^t_{[1]},\ldots,\Lambda^t_{[K]}\right)$.

We now present the resulting ADI method using shorthand notation. At each iteration we solve the two subproblems

$$\begin{array}{rll} x^{t+1} = & \underset{x}{\text{argmin}} & f(x) + p^{tT}x + \frac{1}{2}(x - y^t)^T\Lambda^t(x - y^t) \\ & \text{subject to} & \left.\begin{array}{c} A_{[i]}x_{[i]} = b_{[i]} \\ 0 \le x_{[i]} \le u_{[i]} \end{array}\right\} \ i = 1,\ldots,K \end{array} \tag{53}$$

$$\begin{array}{rll} y^{t+1} = & \underset{y}{\text{argmin}} & -p^{tT}y + \frac{1}{2}(y - x^{t+1})^T\Lambda^t(y - x^{t+1}) \\ & \text{subject to} & \sum\limits_{i=1}^{K} D_{[i]}y_{[i]} \le d \end{array} \tag{54}$$

and then update the multipliers

$$p^{t+1} = p^t + \Lambda^t\,(x^{t+1} - y^{t+1}) \tag{55}$$

This algorithm can be simplified and parallelized in ways similar to those for the other two splittings. In the first subproblem (53) objective and constraints decompose per block, so we can solve the following K block-problems in parallel and concatenate the solutions.

$$\begin{array}{ll} \min\limits_{x_{[i]}} & f_{[i]}(x_{[i]}) + p_{[i]}^{t\,T}x_{[i]} + \dfrac{1}{2}(x_{[i]} - y^t_{[i]})^T\Lambda^t_{[i]}(x_{[i]} - y^t_{[i]}) \\ \text{subject to} & A_{[i]}x_{[i]} = b_{[i]} \\ & 0 \le x_{[i]} \le u_{[i]} \end{array} \tag{56}$$

The second subproblem is once more a linearly-constrained least squares problem. The Karush–Kuhn–Tucker conditions for quadratic programming yield that the primal solution y^{t+1} is given by

$$y^{t+1}_{[i]} = x^{t+1}_{[i]} + (\Lambda^t_{[i]})^{-1}\left(p^t_{[i]} - D_{[i]}{}^T\mu^{t+1}\right) \tag{57}$$

with the vector μ^{t+1} satisfying

$$\begin{array}{rcl} \mu^{t+1} & \ge & 0 \\ q + D(\Lambda^t)^{-1}D^T\mu^{t+1} & \ge & 0 \\ \mu^{t+1T}[q + D(\Lambda^t)^{-1}D^T\mu^{t+1}] & = & 0 \end{array} \tag{58}$$

in which

$$D(\Lambda^t)^{-1}D^T =: \sum_{i=1}^{K} D_{[i]}(\Lambda^t_{[i]})^{-1}{D_{[i]}}^T, \text{ and } q := d - \sum_{i=1}^{K} D_{[i]}\left(x^{t+1}_{[i]} - (\Lambda^t_{[i]})^{-1}p^t_{[i]}\right).$$

Thus μ^{t+1} solves the symmetric linear complementarity problem LCP $\left(q, D(\Lambda^t)^{-1}D^T\right)$. This LCP is feasible, i.e. there exist vectors that satisfy the first two inequalities in (58) (by the Karush–Kuhn–Tucker conditions) and the LCP matrix is positive semi-definite. Then, by a standard theorem, ([6, chapter 3]) the LCP is solvable. In the general case, it can be solved by either a pivotal method, such as Lemke's, or an iterative one, such as matrix-splitting schemes (see [6, chapters 4 and 5] for a good coverage of such methods.) However, having to solve a non-trivial LCP per iteration is likely to increase the duration of the non-parallelizable *join* phase of the algorithm. For certain cases, such as the multicommodity flow problem, the LCP matrix $D(\Lambda^t)^{-1}D^T$ is diagonal and invertible. Then the solution μ^{t+1} can be written in closed form

$$\mu^{t+1} = \left([D(\Lambda^t)^{-1}D^T]^{-1}\left[\sum_{i=1}^{K} D_{[i]}\left(x^{t+1}_{[i]} - (\Lambda^t_{[i]})^{-1}p^t_{[i]}\right) - d\right]\right)_+ \tag{59}$$

In all cases the p multiplier update (55) yields simply

$$p^{t+1}_{[i]} = {D_{[i]}}^T\mu^{t+1} \tag{60}$$

The invariant properties of the iterates are described in the following lemma.

Lemma 4.11 (Invariants of the (AP) splitting) *Let the ADI scheme (53)–(55) be started from any p^0, y^0 and any diagonal positive matrix Λ^0. Then, for all $t \geq 0$,*
(i) $p^{t+1}_{[i]} = {D_{[i]}}^T\mu^{t+1}$, $i = 1, \ldots, K$, with $\mu^{t+1} \geq 0$.
(ii) $\sum_{i=1}^{K} D_{[i]}y^{t+1}_{[i]} \leq d$.

We can have these properties established one iteration earlier, by a canonical initialization.

Definition 4.12 *An initialization p^0, y^0 for the ADI scheme for (**AP**) is* CANONICAL *if*

$$p^0_{[1]} = p^0_{[2]} = \ldots = p^0_{[K]} = {D_{[i]}}^T\mu^0, \text{ for } \mu^0 \geq 0 \quad \text{and} \sum_{i=1}^{K} D_{[i]}y^0_{[i]} \leq d$$

4.3.2 The algorithm simplified

For this splitting, it is the μ vectors, rather than the p multipliers, that carry the essential global information for the updates. We now present the simplified algorithm in terms of these vectors. We also make use of canonical initialization.

(0) Pick a non-negative vector μ^0, proximal vectors $y_{[i]}^0$, $i = 1, \ldots, K$ and a diagonal positive matrix Λ^0.
(1) Compute (in parallel) per block $i = 1, \ldots, K$, $x_{[i]}^{t+1}$, the solution to

$$\begin{array}{ll} \min\limits_{x_{[i]}} & f_{[i]}(x_{[i]}) + \mu^{tT} D_{[i]} x_{[i]} + (x_{[i]} - y_{[i]}^t)^T (\Lambda_{[i]}^t)(x_{[i]} - y_{[i]}^t) \\ \text{subject to} & A_{[i]} x_{[i]} = b_{[i]} \\ & 0 \leq x_{[i]} \leq u_{[i]} \end{array} \tag{61}$$

(2) Solve for μ^{t+1} the linear complementarity problem

$$LCP\ \left(d - Dx^{t+1} - D(\Lambda^t)^{-1} D^T \mu^t,\ D(\Lambda^t)^{-1} D^T \right) \tag{62}$$

(3) Update the proximal y-terms

$$y_{[i]}^{t+1} = x_{[i]}^{t+1} + (\Lambda_{[i]}^t)^{-1} D_{[i]}{}^T \left(\mu^t - \mu^{t+1} \right) \tag{63}$$

(4) Update the penalty matrix $\Lambda_{[i]}^t$.
(5) If termination criteria are met, stop. Else set $t = t + 1$ and go to (1).

If the matrix $D(\Lambda^t)^{-1} D^T$ is diagonal and invertible, the vector μ^t can be simply updated by

$$\mu^{t+1} = \left(\mu^t + [D(\Lambda^t)^{-1} D^T]^{-1} \left[\sum_{i=1}^{K} D_{[i]} x_{[i]}^{t+1} - d \right] \right)_+ \tag{64}$$

Then, all the updates are of a very simple computational nature, involving matrix-vector and vector-vector operations on local data, except for the vector $\sum\limits_{i=1}^{K} D_{[i]} x_{[i]}^{t+1}$ which needs to be computed from contributions from all blocks.

In the general case, we can have a "master" processor solve the LCP, and then broadcast the solution μ^{t+1} to all processors; alternatively, to reduce communication overhead, each processor can solve the LCP locally per iteration. In both cases, the LCP matrix can be synthesized by globally adding the local matrix-matrix products $D_{[i]} (\Lambda_{[i]}^t)^{-1} D_{[i]}{}^T$.

4.3.3 Convergence of the algorithm

Since the splitting matrix for (**AP**) has full column rank, we have strong convergence of the iterates, similar to the (**ARP**) case.

Theorem 4.13 (Convergence of (AP) splitting) *Assume that (**CBA**) is solvable. Let the ADI method be started from arbitrary p^0, y^0 and arbitrary diagonal positive matrix Λ^0, and let the sequence $\{\Lambda^t\}$ be bounded above and bounded away from zero. Then, the sequence $\{x^t\}$ of ADI iterates for the (**AP**) splitting converges to an optimal primal solution for (**CBA**).*

5 Computational results

We have implemented the three ADI algorithms on the Connection Machine CM-5 [29] at the Computer Sciences Department of the University of Wisconsin–Madison, which consists of two 32-PE partitions of 33-MHz SPARC processors connected in a fat-tree topology. The machine uses the CMOST Version 7.3 Beta.2.7 operating system. We have employed version 3.1 of the CMMD message-passing library for interprocessor communication, for efficient calculation of global quantities such as sums and binary functions.

We are mainly interested in comparing the relative performance of the three splittings in terms of number of iterations and total computational time to achieve a prescribed solution accuracy. To assess performance we used MNETGEN [1] to generate three hundred random multicommodity network problems (**MC**), one hundred with linear objectives and two hundred with quadratic objective functions. For (**MC**) the constraint matrices $A_{[i]}$ and $D_{[i]}$ have a special structure. Each $A_{[i]}$ is the node-arc incidence matrix of a topological network; rows correspond to nodes and columns to arcs; each column has exactly two non-zero entries, one of which is $+1$, and the other is -1. Each matrix $D_{[i]}$ consists of columns with one $+1$ entry and of zero columns; nonzero columns correspond to arcs that appear in the linking constraints. Thus, constraint matrices for (**MC**) are very sparse. A crucial property of the $D_{[i]}$ matrices for (**MC**) is that the matrix $D(\Lambda^t)^{-1}D^T$ for the (**AP**) splitting is diagonal and invertible: then a closed form solution for the Linear Complementarity Problem (58) exists, and is given by (59).

It is important to note that, although we mainly experimented with (**MC**) problems, our parallel code is written for the general (**CBA**) problem: we do not take into account the special structure of $A_{[i]}$ and $D_{[i]}$, other than treating them as sparse matrices. This enables the code to handle much more general block–angular problems.

Table 1 displays the characteristics of the test problems. Five instances were randomly generated for each case; the table reports average characteristics. The column labeled "LP size" refers to the size of the multicommodity problem formulated as a full linear problem; "Constraints" is the total number of rows in the constraint matrix for this LP, excluding the box constraints, and "Variables" is the total number of LP variables, excluding slacks.

The number of blocks for our test problems varies between 4 and 31, and therefore each processor is assigned exactly one subproblem. (In case there are more blocks than processors, some heuristic should be used to ensure that the assignment of blocks to processors results in balanced loads.) The number of network nodes (rows) per block varies between 50 and 400, while the number of arcs (variables) varies between 111 and 844. The number of coupling constraints varies between 65 and 543. The networks have 60% to 75% of their arcs capacitated; 55% to 70% of the arcs participate in the coupling constraints.

In order to reduce the total solution time and the number of iterations, we incorporated several heuristics in the code. In particular, we found that the choice of the starting point x^0 can be quite important. In all three splittings x^0 was chosen as the solution of (**MC**) using only the linear objective terms and discarding the coupling constraints.

We also experimented with several schemes for updating the penalty matrices Λ. Best results for this class of problems were obtained with an updating scheme similar to the one used in the standard Augmented Lagrangian algorithm: a penalty parameter is increased when the violation in the corresponding coupling constraint is not sufficiently reduced ([4, chapter 2]). In our code all penalty parameters remain within positive bounds, and are never decreased. In case a single fixed penalty parameter is used, it is reported in [16] that the number of iterations can significantly increase if the penalty is chosen too small or too large. Similar computational results are reported in [14] and [10].

The solution of the subproblems was obtained using MINOS 5.4 ([25],[26]). Since the feasible region with respect to the block constraints remains the same over all iterations, "warm start" techniques were used, and the optimal solution at iteration t was employed as the starting point for iteration $t+1$.

Multicommodity network problems						
Problem id	Blocks	Subproblem size		Coupling	Equivalent LP size	
		Nodes	Arcs	constraints	Constraints	Variables
1	4	50	111	65	265	444
2	8	50	120	82	482	960
3	16	50	120	88	888	1920
4	31	50	121	83	1633	3751
5	4	100	227	139	539	908
6	8	100	230	142	942	1840
7	16	100	241	148	1748	3856
8	31	100	244	151	3251	7564
9	4	200	450	301	1101	1800
10	8	200	450	288	1888	3600
11	16	200	450	316	3516	7200
12	31	200	449	306	6506	13919
13	4	300	683	432	1632	2732
14	8	300	687	435	2835	5496
15	16	300	713	451	5251	11408
16	31	300	682	436	9736	21142
17	4	400	844	525	2125	3376
18	8	400	839	543	3743	6712
19	16	400	818	520	6920	13088
20	31	400	832	538	12938	25792

TABLE 1: *Test problem characteristics*

The global coordination phase requires adding or averaging vectors that are local to each processor. The aggregated values can be easily computed on the CM-5 with a global call to the `CMMD_vector_reduce` function.

We terminate the algorithm when the component–wise relative difference between consecutive proximal terms was less that 10^{-5}. The feasibility tolerance for MINOS was set to 10^{-8}. This implies that the primal solution vectors violate the block constraints by no more than 10^{-8}. Moreover, at termination the coupling constraints were satisfied within a component–wise relative tolerance of 10^{-5}.

The relative performance of the three splittings is presented in a graphical way in Figures 1 to 4. In all figures each group of lines corresponds to problems with the same number of nodes, with the number of commodities varying from 4 to 31. The curves present average solution times and iteration counts over the five instances per problem case. Timing excludes only the initial reading of block data. In all figures the results for the (**AP**) spitting are shown in solid lines, the results for (**RP**) in thick dashed lines, while thin dashed lines correspond to results for the (**ARP**) splitting. The timing results clearly demonstrate the general superiority of the (**AP**) splitting.

Figure 1 reports total CPU time in seconds and total number of iterations for the three splittings for the

linear (**MC**). Results for the quadratic (**MC**) are shown in Figures 2 and 3. Here the objective functions are

$$\sum_{i=1}^{K} {c_{[i]}}^T x_{[i]} + r \left\| x_{[i]} - \bar{x}_{[i]} \right\|_2^2$$

in which $\bar{x}_{[i]}$ is the solution of (**MC**) using only the linear objective terms and discarding the coupling constraints, and the quadratic factor is $r = 0.05$ (Figure 2) and $r = 0.5$ (Figure 3). Finally the average CPU time per iteration for both quadratic cases is reported in Figure 4. With reference to these figures, we make the following observations:

- The (**AP**) and (**RP**) algorithms perform significantly better than (**ARP**), although the latter takes on the average no more iterations to optimality than either of the other two. However, the subproblem that needs to be solved at each iteration for (**ARP**) is larger than the one for the other two splittings, both in the number of variables and the number of constraints. This significantly influences the solution time, and is responsible for the high growth rate of CPU time for (**ARP**) as the problem size increases. The (**AP**) algorithm is the best among the three: for the largest problem we considered, having 12938 rows and 25792 variables in the equivalent LP formulation, it takes the (**AP**) algorithm 106 seconds to solve the linear problem, 115 to solve the quadratic problem with factor $r = 0.05$ and 147 seconds for the $r = 0.5$ quadratic problem. The respective times for (**RP**) are 142, 165 and 204 seconds.

- For all three splittings, the number of required iterations decreases, as the quadratic factor r increases. Thus it may be possible to solve linear problems more efficiently by embedding them in a series of strongly convex quadratic ones, obtained by adding a proximal term to the linear objective.

- As r increases, the cost per iteration also increases. This can be attributed to the behavior of the MINOS solver, which requires more inner iterations and function evaluations as the subproblem objective becomes a steeper quadratic. Other solvers may not exhibit such a characteristic.

- As expected, for a fixed number of sub-network nodes, the time per iteration increases, as the number of blocks increases; this is exhibited quite dramatically in the case of the problems with 200 nodes per block (problems $9-11$). It appears that the simple coordination of these splittings, via aggregation of subproblem vectors, becomes less effective as more sub-networks (i.e. blocks) are added.

- For a fixed number of blocks, the number of iterations to optimality is almost independent of the subproblem size.

Finally, we employed MINOS 5.4 on a 25-MHz DEC 5000 workstation to solve most of both the linear and quadratic (**MC**) problems. In Figures 5 and 6 we compare the CPU solution time in seconds for this serial MINOS versus the (**AP**) splitting on the CM-5. For linearly constrained problems, MINOS 5.4 employs the revised simplex method in the case of linear objective, and the reduced-gradient method otherwise. We set all MINOS tolerances to their default values; in particular, the default for both the feasibility and the optimality tolerance is 10^{-6}. We used a "cold start" in all cases, and treated all matrices as sparse.

These figures demonstrate that, for the larger problems, the parallel (**AP**) method is better than the serial MINOS 5.4 by one to two orders of magnitude. For instance, for problem 16, with 9736 rows and 21142 variables in the equivalent LP formulation, the (**AP**) algorithm requires 95 seconds to solve the linear problem, 81 seconds for the quadratic problem with $r = 0.05$ and 128 seconds for the $r = 0.5$ quadratic problem. The respective times for MINOS 5.4 on the DEC 5000 are 9867, 15625 and 6343 seconds.

This gap in performance can be attributed to the fact that solution times for MINOS 5.4 demonstrate, as expected, an almost quadratic growth as a function of the size of the problem. The decomposition methods, on the other hand, decompose the problem into subproblems whose size may remain fixed as the overall problem size grows, and therefore solution time increases slowly as a function of problem size.

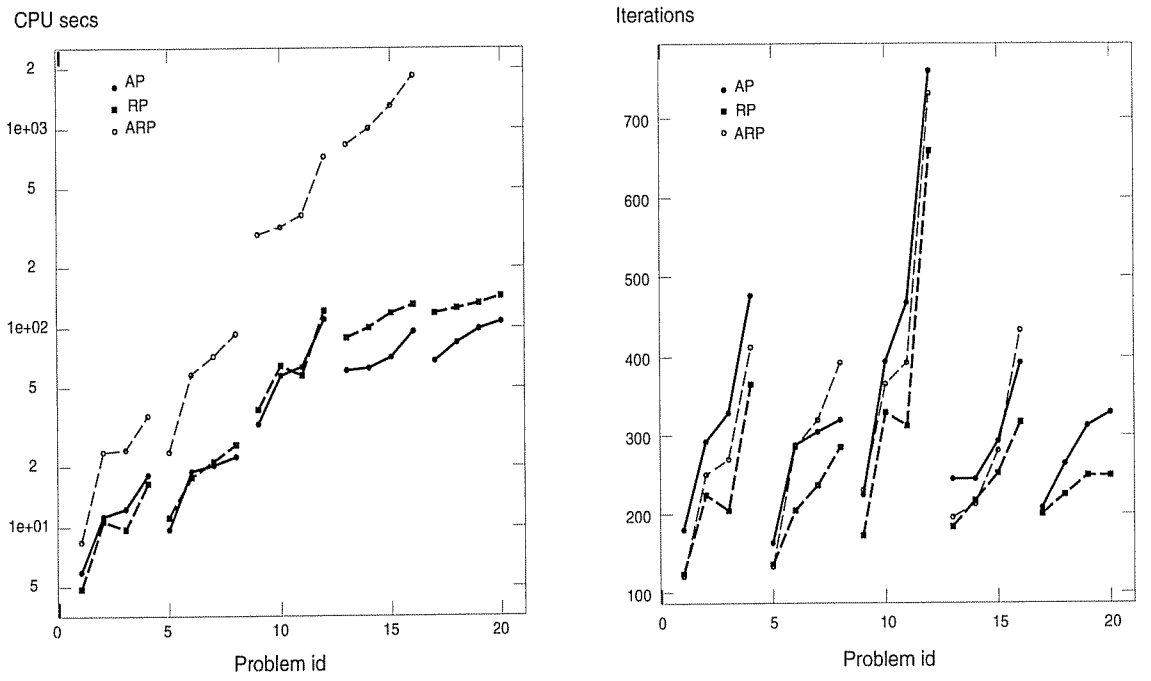

FIGURE 1: *Performance of the three splittings for the* ***linear*** *multicommodity problems: CPU seconds and number of iterations. (The* ***(AP)*** *splitting corresponds to solid lines.)*

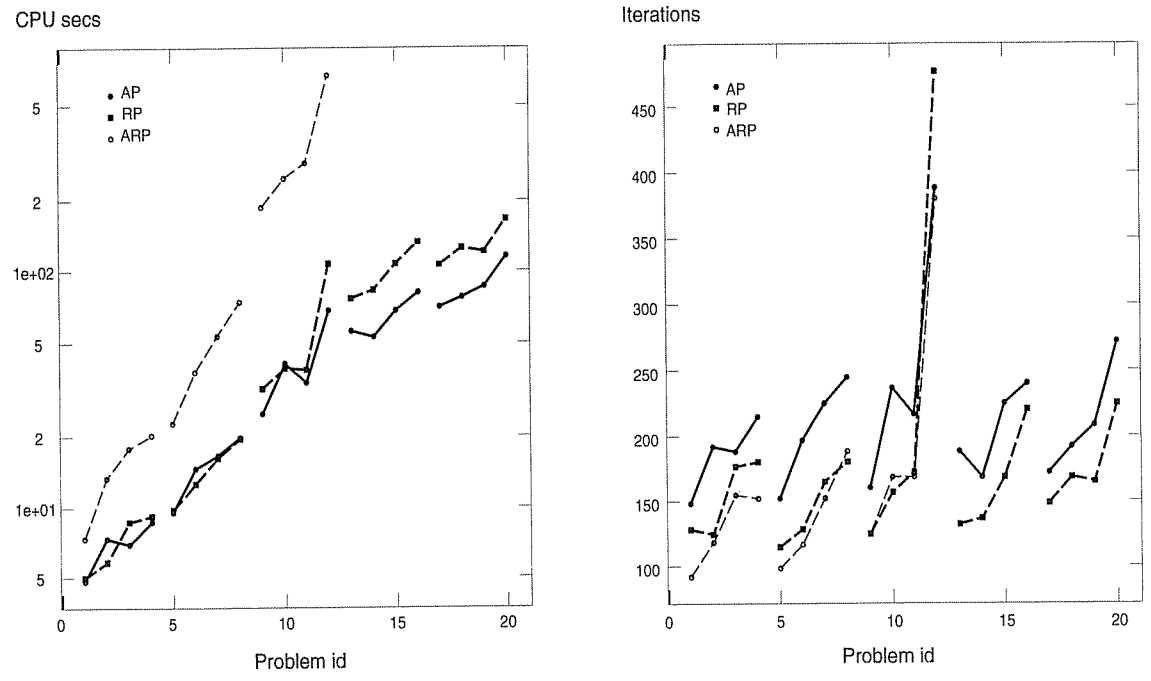

FIGURE 2: *Performance of the three splittings for the* ***quadratic*** *multicommodity problems for quadratic factor* $r = 0.05$*: CPU seconds and number of iterations. (The* ***(AP)*** *splitting corresponds to solid lines.)*

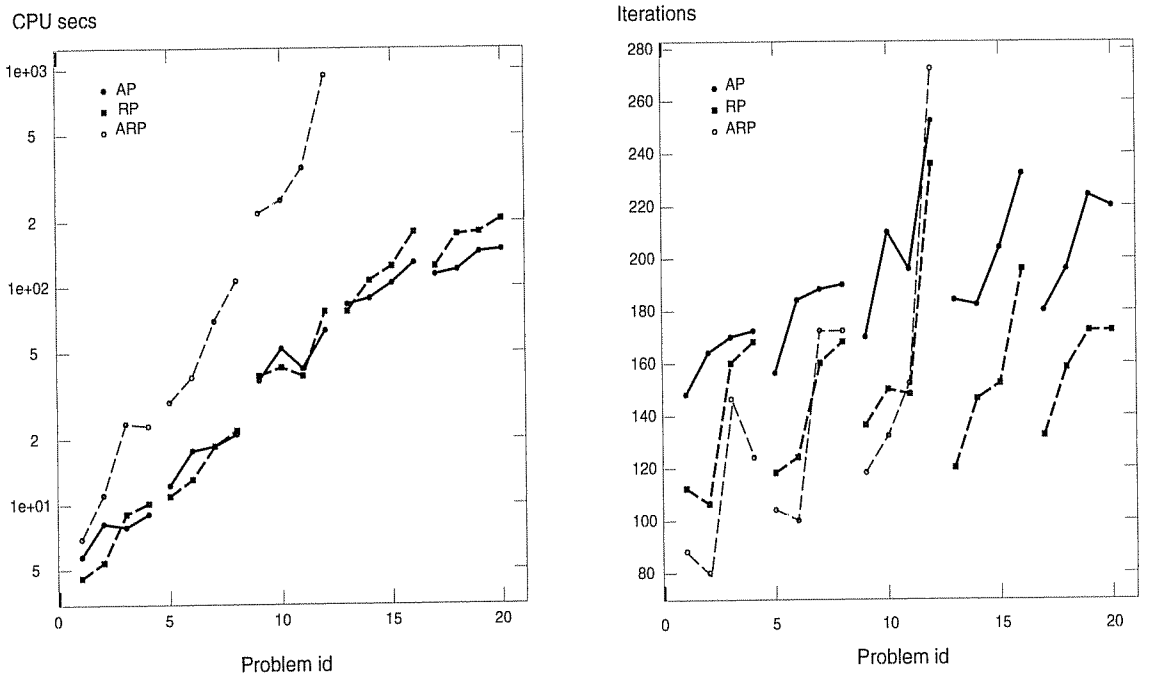

FIGURE 3: *Performance of the three splittings for the* ***quadratic*** *multicommodity problems for quadratic factor* $r = 0.5$*: CPU seconds and number of iterations. (The* ***(AP)*** *splitting corresponds to solid lines.)*

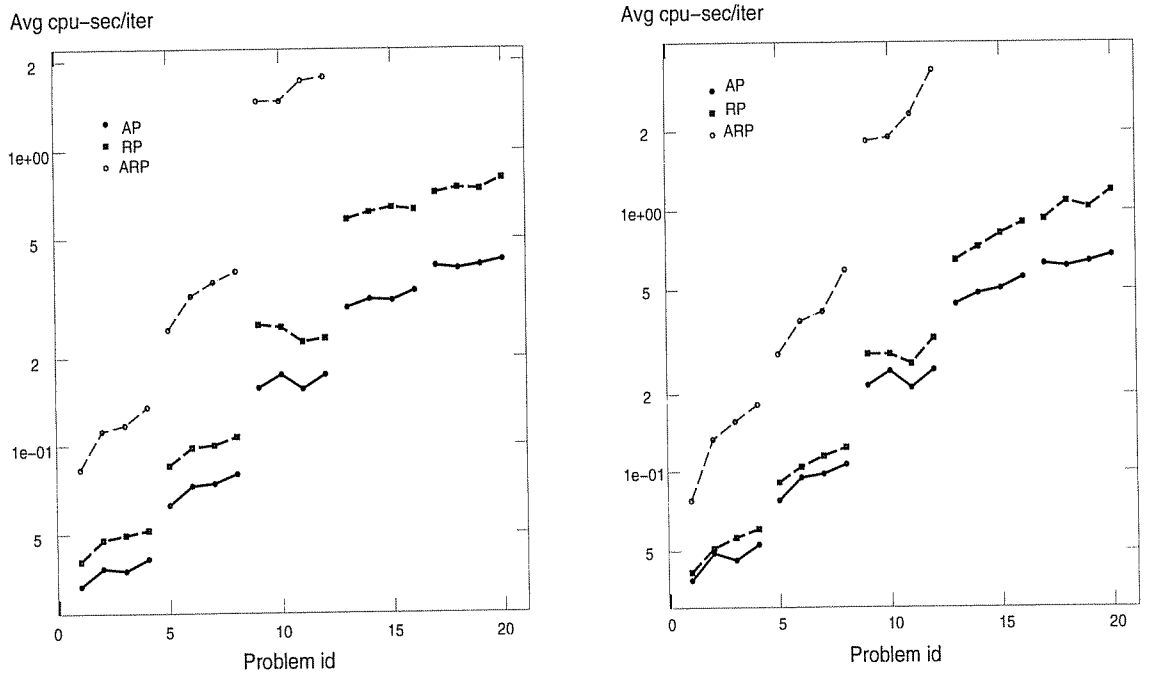

FIGURE 4: *Average CPU time spent per iteration per splitting for the* ***quadratic*** *multicommodity problems, for quadratic factor* $r = 0.05$ *(left) and* $r = 0.5$ *(right). (The* ***(AP)*** *splitting corresponds to solid lines.)*

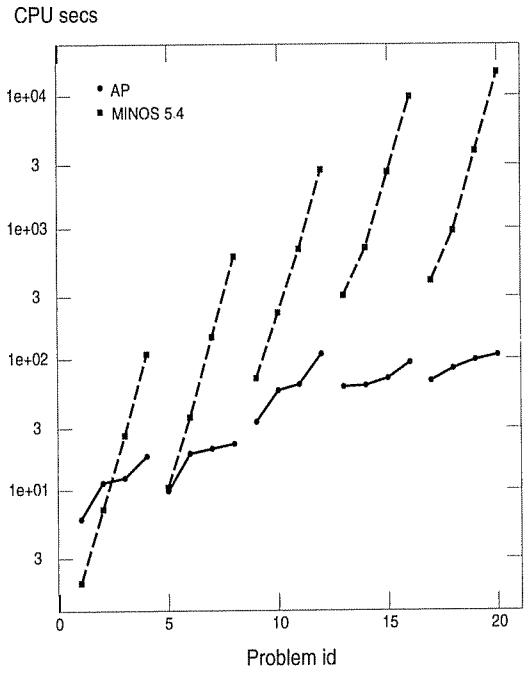

FIGURE 5: *Comparison of CPU seconds for the (**AP**) splitting on the CM-5 versus MINOS 5.4 on a DEC 5000 workstation, for the **linear** multicommodity problems. (The (**AP**) splitting corresponds to solid lines.)*

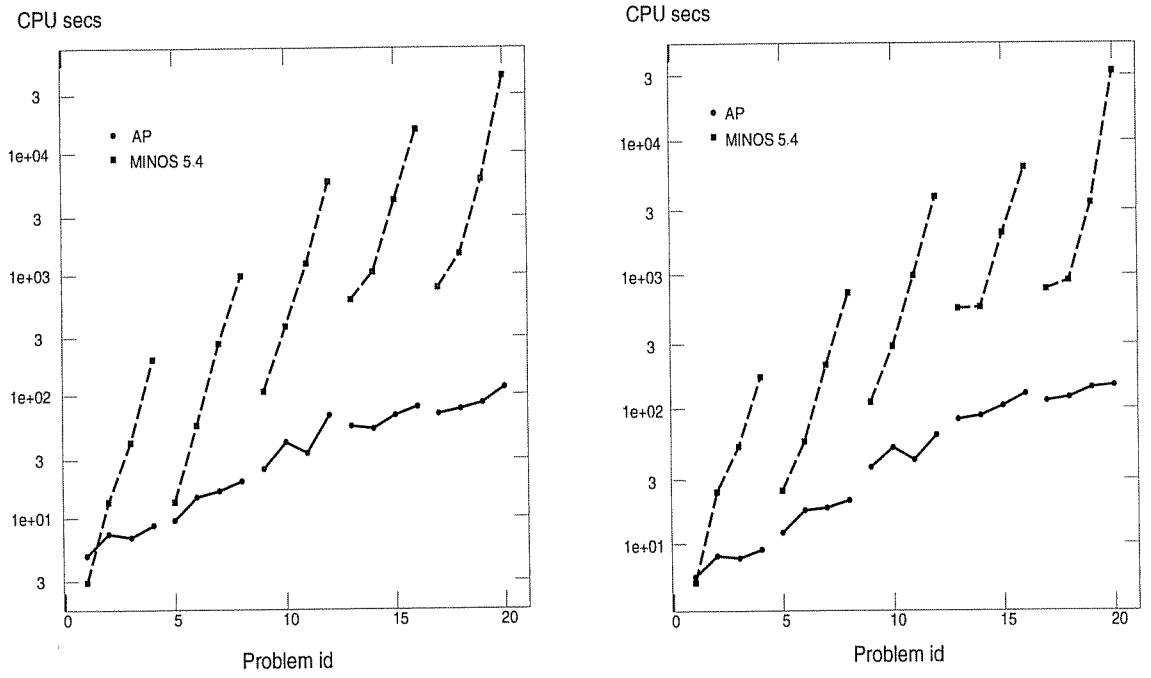

FIGURE 6: *Comparison of CPU seconds for the (**AP**) splitting on the CM-5 versus MINOS 5.4 on a DEC 5000 workstation, for the **quadratic** multicommodity problems with quadratic factors $r = 0.05$ (left) and $r = 0.5$ (right). (The (**AP**) splitting corresponds to solid lines.)*

6 Extensions of the Alternating Directions method

We now briefly mention two variants of the general ADI method, that can be used in deriving additional splitting algorithms for (**CBA**).

In the first variant the subproblems are solved inexactly and a relaxation parameter ω is added. The following theorem is from Eckstein and Bertsekas [11].

Theorem 6.1 (Over-relaxation plus inexact minimization) *Consider the finite - dimensional problem*

$$\begin{array}{ll} \min\limits_{x,\,z} & G_1(x) + G_2(z) \\ \textit{subject to} & Ax = z \end{array}$$

with A a full-column rank matrix and G_1, G_2 closed, proper, convex, extended-real-valued functions. Let the non-negative real sequences $\{\mu^t\}$, $\{\nu^t\}$ and $\{\omega^t\}$ satisfy

$$\sum_t \mu^t + \sum_t \nu^t < \infty, \quad \{\omega^t\} \subset (0,2), \quad 0 < \inf_t \omega^t \leq \sup_t \omega^t < 2$$

and let the real sequences $\{x^t\}$, $\{z^t\}$ and $\{p^t\}$ satisfy

$$\left\| x^{t+1} - \underset{x}{argmin}\, \{G_1(x) + p^{t^T} Ax + \frac{\lambda}{2} \left\| Ax - z^t \right\|_2^2\} \right\| \leq \mu^t$$

$$\left\| z^{t+1} - \underset{z}{argmin}\, \{G_2(z) - p^{t^T} z + \frac{\lambda}{2} \left\| \omega^t Ax^{t+1} + (1-\omega^t) z^t - z \right\|_2^2\} \right\| \leq \nu^t$$

$$p^{t+1} = p^t + \lambda \left[\omega^t Ax^{t+1} + (1-\omega^t) z^t - z^{t+1} \right]$$

in which λ is a real positive scalar and (p^0, z^0) is arbitrary. If the problem has a Karush–Kuhn–Tucker point, then $\{x^t\}$ converges to a primal solution and $\{p^t\}$ to a solution of the dual. If the dual has no optimal solution, then at least one of the sequences $\{x^t\}$, $\{p^t\}$ is unbounded.

In the second variant, called the *Peaceman–Rachford* method after [27], an additional multiplier is introduced and updated between the solution of the subproblems. The intention is to incorporate in the modified objective of the second subproblem the most recent information about the violation of the constraints. The resulting algorithm converges under more stringent assumptions, and is also less robust numerically, as Fortin and Glowinski point out in [14], citing experience in a variety of numerical analysis applications. The following theorem is from Eckstein [10].

Theorem 6.2 (Peaceman–Rachford method for convex optimization) *Let the convex problem described in theorem 6.1 have a Karush–Kuhn–Tucker point. Let either of the following conditions hold*
(i) G_1 is strictly convex, the subdifferential set of G_1 is a singleton, A is square and invertible,
(ii) G_2 is strictly convex, the subdifferential set of G_2 is a singleton, A has full column rank,
Let the iterative method

$$\begin{array}{rcl} x^{t+1} & \in & \underset{x}{argmin}\ G_1(x) + p^{t^T} Ax + \frac{\lambda}{2} \left\| Ax - z^t \right\|_2^2 \\ q^{t+1} & = & p^t + \lambda(Ax^{t+1} - z^t) \\ z^{t+1} & \in & \underset{z}{argmin}\ G_2(z) - q^{t+1^T} z + \frac{\lambda}{2} \left\| Ax^{t+1} - z \right\|_2^2 \\ p^{t+1} & = & q^{t+1} + \lambda \left(Ax^{t+1} - z^{t+1} \right) \end{array} \tag{65}$$

be started from arbitrary vectors p^0, z^0 and let the scalar $\lambda > 0$ be ultimately fixed. Then, $\{x^t\}$ and $\{p^t\}$ converge to solutions of the primal and dual problem, respectively.

In [20] relaxation and Peaceman–Rachford iterative schemes for the (**ARP**) and (**AP**) splittings for (**CBA**) are derived. The main theoretical result is that the update formulas for these variants and the standard ADI (Douglas–Rachford) method are a family parameterized by a single scalar. For these splittings it is shown that the Peaceman–Rachford method is the Douglas–Rachford method with over-relaxation factor 2.

Conclusions

We have investigated the theoretical aspects and the practical performance of three decomposition algorithms derived from the method of Alternating Directions. These methods can be viewed as Augmented Lagrangian methods modified to take full advantage of the special block-structure of the problem. The resulting algorithms are highly parallel and have been efficiently implemented under the *fork–join* protocol. Computational experience on linear and quadratic multicommodity network problems on the Thinking Machines CM-5 parallel supercomputer indicates that, because of the reduced size of the subproblems solved at each iteration, the (**AP**) and (**RP**) algorithms, based on proximizing either resources or activities, perform significantly better than the (**ARP**) algorithm, which proximizes both resources and activities. On large-scale problems, this parallel implementation of the (**AP**) splitting is one to two orders of magnitude faster than serial MINOS 5.4 .

References

[1] A.I. Ali and J.L. Kennington. MNETGEN program documentation. Technical Report IEOR 77003, Dept. of Industrial Engineering and Operations Research, Southern Methodist University, Dallas, TX, 1977.

[2] A.A. Assad. Multicommodity network flows: A survey. *Networks*, 8:37–91, 1978.

[3] D.P. Bertsekas. Multiplier methods: A survey. *Automatica*, 12:133–145, 1976.

[4] D.P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Academic Press, New York, 1982.

[5] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice–Hall Inc., 1989.

[6] R.W. Cottle, J.-S. Pang, and R.E. Stone. *The Linear Complementarity Problem*. Academic Press, 1992.

[7] G.B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.

[8] R. De Leone, R.R. Meyer, S. Kontogiorgis, A. Zakarian, and G. Zakeri. Coordination methods in coarse-grained decomposition. To appear in *SIAM Journal on Optimization*.

[9] J. Douglas and H.H. Rachford Jr. On the numerical solution of heat conduction problems in two- and three–space variables. *Transactions of the American Mathematical Society*, 82:421–439, 1956.

[10] J. Eckstein. *Splitting methods for monotone operators with applications to parallel optimization*. PhD thesis, Massachusetts Institute of Technology, Department of Civil Engineering, 1989.

[11] J. Eckstein and D.P. Bertsekas. On the Douglas–Rachford splitting method and the proximal point method for maximal monotone operators. *Mathematical Programming, Series A*, 55(3):293–318, 1992.

[12] M.C. Ferris and O.L. Mangasarian. Parallel constraint distribution. *SIAM Journal on Optimization*, 1(4):487–500, 1991.

[13] M.C. Ferris and O.L. Mangasarian. Parallel variable distribution. Technical Report 1175, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1993.

[14] M. Fortin and R. Glowinski. On decomposition–coordination methods using an Augmented Lagrangian. In M. Fortin and R. Glowinski, editors, *Augmented Lagrangian methods: Applications to the numerical solution of boundary-valued problems*. North–Holland, 1983.

[15] M. Frank and F. Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3:95–110, 1956.

[16] M. Fukushima. Application of the alternating direction method of multipliers to separable convex programming problems. *Computational Optimization and Applications*, 1:93–111, 1992.

[17] D. Gabay. Applications of the method of multipliers to variational inequalities. In M. Fortin and R. Glowinski, editors, *Augmented Lagrangian methods: Applications to the numerical solution of boundary-valued problems*. North–Holland, 1983.

[18] R. Glowinski and P. Le Tallec. *Augmented Lagrangian and Operator-Splitting Methods in Nonlinear Mechanics*. Society for Industrial and Applied Mathematics, 1989.

[19] P. Kall. *Stochastic Linear Programming*. Springer Verlag, 1976.

[20] S. Kontogiorgis. *Alternating Direction Methods for the Parallel Solution of Large-scale Block-structured Optimization Problems*. PhD thesis, University of Wisconsin – Madison, Department of Computer Sciences, in preparation.

[21] P.L. Lions and B. Mercier. Splitting algorithms for the sum of two nonlinear operators. *SIAM Journal on Numerical Analysis*, 16(6):964–979, 1979.

[22] O.L. Mangasarian. *Nonlinear Programming*. McGraw-Hill, 1969.

[23] G.I. Marchuk. Splitting and alternating direction methods. In P.G. Ciarlet and J.L. Lions, editors, *Handbook of Numerical Analysis*. North–Holland, Berlin, 1990.

[24] J.M. Mulvey and A. Ruszczyński. A diagonal quadratic approximation method for large scale linear programs. *Operations Research Letters*, 12:205–215, 1992.

[25] B.A. Murtagh and M.A. Saunders. MINOS 5.1 user's guide. Technical Report SOL 83.20R, Stanford University, 1987.

[26] B.A. Murtagh and M.A. Saunders. MINOS 5.4 release notes, appendix to MINOS 5.1 user's guide. Technical report, Stanford University, 1992.

[27] D.W. Peaceman and H.H. Rachford Jr. The numerical solution of parabolic and elliptic differential equations. *SIAM Journal*, 3:28–42, 1955.

[28] G.L. Schultz and R.R. Meyer. An interior point method for block angular optimization. *SIAM Journal on Optimization*, 1(4):583–602, 1991.

[29] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM–5 Technical Summary*, 1991.

Bulk Loading Into an OODB: A Performance Study

Janet L. Wiener
Jeffrey F. Naughton

Technical Report #1218a

November 1994 (Revised)

Bulk Loading into an OODB: A Performance Study*

Janet L. Wiener
Jeffrey F. Naughton
Department of Computer Sciences
University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
{wiener,naughton}@cs.wisc.edu

Abstract

Object-oriented database (OODB) users bring with them large quantities of legacy data (megabytes and even gigabytes). In addition, scientific OODB users continually generate new data. All this data must be loaded into the OODB. Every relational database system has a load utility, but most OODBs do not. The process of loading data into an OODB is complicated by inter-object references, or relationships, in the data. These relationships are expressed in the OODB as object identifiers, which are not known at the time the load data is generated; they may contain cycles; and there may be implicit system-maintained inverse relationships that must also be stored.

We introduce seven algorithms for loading data into an OODB that examine different techniques for dealing with circular and inverse relationships. We present a performance study based on both an analytic model and an implementation of all seven algorithms on top of the Shore object repository. Our study demonstrates that it is important to choose a load algorithm carefully; in some cases the best algorithm achieved an improvement of one to two orders of magnitude over the naive algorithm.

1 Introduction

As object-oriented databases (OODB) attract more and more users, the problem of loading the users' data into the OODB becomes more and more important. The current methods of loading, i.e., **insert** statements in a data manipulation language, or **new** statements in a database programming language, are more appropriate for loading tens and hundreds of objects than tens and hundreds of megabytes of objects. Yet users want to load megabytes and even gigabytes of data:

- Users bring legacy data from relational and hierarchical databases (that is better suited to an OODB).
- Users with data already in an OODB sometimes need to dump and reload that data, into either the same or another OODB. The most common need for dumping and loading arises when a particular database must be reclustered for performance reasons. If the database uses physical object identifiers (OIDs), there may be no good way to recluster the objects online, but if the objects are dumped to a data file in the order in which they should be clustered, it is simple to recluster them properly

*This work supported in part by NSF grant IRI-9157357 and by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

while reloading. Data must also be dumped and reloaded if the user is switching OODB products, or transferring a large quantity of data across a great distance, e.g., on tape.

- Scientists are starting to use OODB to store their experimental data. Scientific applications generate a large volume of data with many complex associations in the information structure [Sho93]. It is not uncommon for a single experiment to have input and output parameters that number in the hundreds and thousands, and must be loaded into the OODB for each experiment. As an example, the climate modeling project at Lawrence Livermore National Laboratory has a very complex schema and generates single data points in the range of 20 to 150 Megabytes; a single data set typically contains 1 to 20 *Gigabytes* of data [DLP+93].

Relational database systems provide a load utility to bypass the individual language statements; OODB need a similar facility. Users are currently spending too much time and effort just loading the data they want to examine. For example, Cushing reports that loading the experimental data was the most time-consuming part of analyzing a set of computational chemistry experiments [CMR92]. In addition, we know of another commercial OODB customer who currently spends 36 hours loading a single set of new data every month.

A load utility takes a description of all the data to be created, usually in text format, and loads the corresponding objects into the database. Additionally, a load utility can group certain operations, such as integrity checks, to dramatically reduce their cost for the load [Moh93a]. Although a load utility is common in relational databases, we are aware of only one OODB load utility, in Objectivity/DB [Obj92], and it is limited in that it can only load data that already contains system-specific OIDs.

Loading object-oriented data is complicated by the presence of relationships among the objects; these relationships prevent simply the use of a relational load utility for an OODB.

- In a relational database, all data stored in a tuple is either a string or a number. Tuples use foreign keys, which are part of the user data, to reference other tuples. Objects use relationships to reference each other by their OIDs. These OIDs are created and maintained by the database and are usually not visible to the user. Furthermore, these OIDs are not available at all when the load file is written, because the corresponding objects have not yet been created. Relationships must therefore be represented by some other means in the load file. We call this representation a *surrogate* identifier.

- Relationships may be circular. That is, an object A may refer to an object B which refers back to object A, either directly or via a chain of relationships. Therefore, the load utility must be able to resolve surrogate identifiers to objects that have not yet been created when the surrogate is first seen.

- Inverse relationships, sometimes called bidirectional relationships, are relationships that are maintained in both directions, so that an update to one direction of the relationship causes an update to the other. Many OODB support system-maintained inverse relationships [Obj92, Ont92, Ver93, OHMS92, WI93], and they are part of the ODMG standard [Cat93]. As an example, suppose that object A has an inverse relationship with object B. Then when B's OID is stored in A, A's OID should be stored in B. The most obvious way to maintain inverse relationships — and the only way if each object is created separately, as by **insert** or **new** — is to update each inverse object immediately after realizing that the update is needed, e.g, updating B immediately after creating A. There are two reasons why this method is not always appropriate: first, object B may not have been created yet; second, this approach leads to performance several orders of magnitude worse than is possible using a different approach.

We examine several techniques for dealing with circular and inverse relationships in our loading algorithms. We evaluate the performance of these algorithms with an analytic model and an implementation on top of the Shore object repository [CDF+94]. We use the analytic model to explore a wide range of load file and system configurations. The implementation not only validates our analytic model, the performance of the algorithms also highlights several key advantages and disadvantages of using logical object identifiers. Furthermore, our performance results show that one algorithm almost always outperforms all the others.

We know of no other work involving loading data into an OODB. There are several published methods for mapping complex data structures to an ASCII or binary file, and then reading it back in again, including Snodgrass's Interface Description Language [Sno89], Pkl [Nel91] for Modula3 data, and Vegdahl's method for Smalltalk images [Veg86]. However, these methods do not address the problem of loading more data than can fit into virtual memory, and also ignore the performance issues that arise when the data to be loaded fits in virtual but not physical memory.

The remainder of the paper is organized as follows. We present the loading algorithms in Section 2. In Section 3 we describe the analytic cost model. Section 4 describes the parameters of the loading algorithms that we varied in our studies and in Section 5 we discuss the performance results obtained from the analytic model. Section 6 describes our implementation and experimental results on top of Shore. We conclude and outline our future work plan in Section 7.

2 Loading Algorithms

We present seven algorithms for loading the database from data stored in a text file. In all the algorithms, we read the file and create the objects described in it. The algorithms differ in the way they handle relationships between objects and in when they create system-maintained inverse relationships.

2.1 Example database schema

```
class Experiment {
    attribute char scientist[16];
    relationship Input input
        inverse Input::expts;
    relationship Output output
        inverse Output::expt;
};
class Input {
    attribute double temperature;
    attribute integer humidity;
    relationship Set<Experiment> expts
        inverse Experiment::input;
};
class Output {
    attribute double plant_growth;
    relationship Experiment expt
        inverse Experiment::output;
};
```

Figure 1: Experiment schema definition in ODL.

We use an example schema, which describes the data for a simplified soil science experiment, to illustrate our algorithms. In this schema, each Experiment object has a many-to-one relationship with an Input object and a one-to-one relationship with an Output object. Figure 1 defines the schema in the Object Definition Language proposed by ODMG [Cat93].

2.2 Data file description

```
Input(temperature, humidity) {
    101:  27.2, 14;
    102:  14.8, 87;
    103:  21.5, 66;
}
Experiment(scientist, input, output) {
    1:  ''Lisa'', 101, 201;
    2:  ''Alex'', 103, 202;
    3:  ''Alex'', 101, 203;
}
Output(plant_growth) {
    201:  2.1;
    202:  1.75;
    203:  2.0;
}
```

Figure 2: Sample data file for the Experiment schema.

The data file is an ASCII text file describing the objects to be loaded[1]. We illustrate the data file format

[1]Loading from binary data files would be similar. We chose to use ASCII files because they are transferrable across different

in Figure 2. Although we developed it for the Moose data model [WI93], it fits a generic OO data model. Furthermore, any existing data file can be converted easily by a simple script to this format. Such conversions will be important for loading pre-existing data, such as the data many scientists have previously kept in flat files.

Within the data file, objects are grouped together by class, although the classes may appear in any order and a given class may appear more than once. Each class is described by its name and relationships. If a relationship of the class is not specified, then objects get a null value for that relationship. Next, each object in the class is described by a surrogate identifier and a list of its values. In this example, all the surrogates are integers, and they are unique in the data file. In general, however, the surrogates may be strings or numbers; if the class has a key they may even be part of the object's data [PG88]. The values for a collection relationship are listed inside curly brackets.

Whenever one object references another object, the data file entry for the referencing object contains the surrogate for the referenced object. The process of loading includes translating all the surrogates into the OIDs that the database assigns to the corresponding objects. To reference objects already in the database, surrogates may be assigned to them by using queries (either before the load or inside the load data file) to individually select the objects; in this study, we do not consider references to existing objects.

2.3 Mapping surrogates to OIDs

Surrogate	OID
101	OID1
102	OID2
103	OID3
1	OID4
2	OID5
3	OID6
201	OID7
202	OID8
203	OID9

Figure 3: Id table built by the load algorithms.

All the algorithms use an *id table* to map surrogates to the database's OIDs. As each object is created, its surrogate and OID are entered into the id table. The OID can subsequently be retrieved from the id table by using its surrogate as a key. Table 3 shows the id table built for the Experiment data file.

hardware platforms and are easy for the user to examine.

2.4 Creating relationships from surrogates

For each relationship from an object A to another object B, the data file contains the surrogate of B in the description of A. At some point during the load, the load utility must store the OID of B inside object A. We present three techniques for converting that surrogate to an OID and storing it in A.

The first technique we call *two-pass*, because the data file is read twice. On the first pass, the objects are created without data inside them and their surrogates and OIDs are entered into the id table. On the second pass, we reread the data file and store the data in the objects. Since all the objects have already been created, we are guaranteed to find all surrogates in the id table.

OID for object to update	Surrogate for OID to store	Update offset
OID4	201	24
OID5	202	24
OID6	203	24

Figure 4: Todo list built by the resolve-early algorithms.

The second technique, called *resolve-early*, employs a *todo list*. The data file is read only once, and we try to resolve all the surrogates to OIDs at that time. Surrogates that refer to objects described further down in the file, however, cannot be resolved immediately. These surrogates are placed on a todo list of updates to do later. Each todo list entry contains the OID of the object to be updated, the surrogate for the OID to store in the object, and the offset at which to store the relationship. Figure 4 contains the todo list created for the Experiment data file in by the resolve-early algorithms. The todo list is read and the updates performed after the entire data file has been read.

The third technique we call *assign-early*. Like in *resolve-early*, in *assign-early* we try to resolve all surrogates on the first and only pass through the data file. Unlike in *resolve-early*, when we encounter a surrogate for an as-yet-uncreated object, we *pre-assign* the OID. Pre-assigning the OID involves requesting an unused OID from the database without creating the corresponding object on disk. This is only possible with logical OIDs. We believe that any OODB that provides logical OIDs can also provide pre-assignment of OIDs; we know it is possible at the buffer manager level in GemStone [Mai94] and in Ontos, as well as in Shore.

2.5 Creating inverse relationships

Whenever we find a relationship from object A to object B that has an inverse, we know we need to store the inverse relationship, i.e., store the OID for A in object B. We present two methods of performing inverse

updates.

In the *immediate inverse update* algorithms, we update the inverse object as soon as we discover the relationship. We note that since surrogates may refer to objects not yet created, this technique only applies to the second pass of *two-pass* algorithms.

Surrogate for object to update	OID to store	Update offset
101	OID4	12
201	OID4	8
103	OID5	12
202	OID5	8
101	OID6	12
203	OID6	8

Figure 5: Inverse todo list built by the inverse-sort algorithms.

In the *inverse sort* algorithms, we make an entry on an *inverse todo list.* Inverse todo entries contain the surrogate for the object to update, the OID to fill in, and an offset. The inverse todo list created for the Experiment data file is shown in Figure 5.

After reading the data file, we process the inverse todo list. The order of the entries is unrelated to the physical order of the objects to update. To avoid a large number of random disk reads, we first sort the inverse todo list so that the order of the entries corresponds to their objects' creation order in the database, which roughly corresponds to their physical order.[2] For the two-pass and resolve-early algorithms, OIDs are assigned sequentially as objects are created; therefore, the OID is the sorting key. For the assign-early algorithms, we use a creation order counter, store each object's order in its id table entry, and use the creation order as the sorting key. The creation order is chosen, instead of the actual physical order, because it matches the order of the objects seen when reading the data file a second time in two-pass algorithms and it matches the order of the todo entries in resolve-early algorithms.

Sorting is done in two phases. First, for each inverse todo list entry, we look up the OID (and creation order) of the object to be updated in the id table and add it to the entry. In this phase we read the inverse todo list in chunks and create sorted runs of 64 Kb. In the second phase, we merge the sorted runs. On the final merge pass, we perform all the updates, touching each page of the database at most once. Figure 6 shows the inverse todo list from Figure 5 after sorting.

Integrity checking is very similar to processing inverse updates. Doing integrity checks during the course of the load corresponds to immediate inverse updates, and deferring integrity checking until the end of the

[2]We predicted that without sorting the inverse todo list, the performance would be similar to that of the immediate inverse update algorithms. Since immediate inverse updates had unacceptable performance, we did not implement an unsorted inverse todo list.

OID for object to update	OID to store	Update offset
OID1	OID4	12
OID1	OID6	12
OID3	OID5	12
OID7	OID4	8
OID8	OID5	8
OID9	OID6	8

Figure 6: Inverse todo list after converting to OIDs and sorting.

load corresponds to building an inverse todo list and then processing it in a separate phase. For relational integrity checking, it is known to be faster to load relations when integrity checking is deferred, because the integrity checks can be reordered to get better sequential I/O [Moh93a].

We note that both of our inverse update techniques ensure the integrity of the inverse relationship, and could be used for other integrity checks that are not part of an inverse relationship.

2.6 An optimization: clearing the todo lists

Both the todo and the inverse todo list are initially constructed in memory. As each list exceeds the size of memory allotted to it, that portion of the list is written out to disk. An optimization for processing both the todo list and the inverse todo list involves checking the entries on each list before writing them to disk, and clearing (removing) those entries from the list that update objects currently in the buffer pool, as these updates can be performed with no I/O cost. Note that an entry can be cleared from the todo list only if the surrogate to store in the object can now be resolved to an OID, that is, if the corresponding object has been created since the todo entry was written.

Minimally, the todo lists are cleared only when they become full and must be written out to disk. However, in our implementation, we clear the todo lists at intervals corresponding to a one-quarter turnover of the contents of the buffer pool and we keep an old and a new todo list. At the end of each interval, we clear both the old and the new todo list and write the old list out to disk. Therefore, we attempt to clear each todo entry twice before writing it to disk.

Figure 7 shows the inverse todo list from Figure 5 as it would look after clearing, if the buffer pool contained three pages (which is half the database). In this example, we were able to clear two entries, or one-third of the total entries, from the inverse todo list.

Surrogate for object to update	OID to store	Update offset
201	OID4	8
202	OID5	8
101	OID6	12
203	OID6	8

Figure 7: Inverse todo list after clearing, with a 3 page buffer pool.

2.7 The algorithms

We now present the seven algorithms we studied, which span all the viable combinations of resolving surrogates and handling inverse relationships.

Naive Naive is the simplest algorithm. It is a two-pass algorithm in which inverse relationships are processed with immediate inverse updates. On the first pass, it reads the data file, creates all the objects (with empty contents), and builds the id table. On the second pass, the objects are filled in with the correct data. Updates for inverse relationships are performed as they are encountered.

Smart-invsort Smart-invsort is also a two-pass algorithm. However, it uses the inverse-sort technique to process inverse relationships. The inverse todo list is constructed during the first pass over the data file, and then sorted before the second pass. During the second pass, the inverse todo updates are read concurrently with the data file, and each object is updated only once.

Late-invsort Late-invsort is an optimization of smart-invsort that requires logical OIDs. In the first pass of smart-invsort, the objects are created simply to obtain their OIDs; they are not filled in until the second pass. In the first pass of late-invsort, OIDs are pre-assigned to the objects and the database is not touched. On the second pass over the data file, the inverse todo updates are merged with the object creations.

Res-early-invsort Res-early-invsort employs the resolve-early technique for surrogates and inserse-sort for inverse relationships. It therefore manages both a todo list and an inverse todo list, and merges the entries from the two lists (after sorting the inverse todo list) during the update phase so that all updates to an object are performed at once. Note that the todo list does not need to be sorted, since the order of the entries already corresponds to the creation order of the objects.

Assign-early-invsort Assign-early-invsort combines the assign-early technique for surrogates with inverse-sort for inverse relationships. It makes one pass over the data file, then sorts the inverse todo list, and makes one pass over the database to perform the updates dictated by the inverse todo entries.

Res-clear-invclear Res-clear-invclear is similar to res-early-invsort, except that it employs the clearing optimization for both the todo and the inverse todo lists.

Assign-early-invclear Assign-early-invclear is similar to assign-early-invsort, except that it uses the clearing optimization for the inverse todo list.

3 Analytic Cost Model

The analytic model measures projected disk I/O costs. We estimated the disk I/O costs to gauge the overall performance of the algorithms because we felt that loading data is inherently I/O bound: loading primarily involves reading a data file and creating (and updating) objects in the database.

Reading the data file once and creating the database objects accounts for the minimum number of I/O's possible in a load. Except for the assign-early algorithms, each algorithm had an additional cost for resolving surrogates to OIDs, and all the algorithms had additional costs for implementing inverse relationships.

We modeled *nearest* locality of reference among the objects, which indicates that an object is most likely to have relationships with objects *near* it in the data file, and hence in the database. More specifically, x% of the relationships from a given object will be to objects within y% of the data file from it. The remaining (1-x)% will be to random objects. When x and y are 0, there is no locality of reference.

Nearest locality models different kinds of complex objects for a data file clustered by complex object. Y says how much of the data file each complex object spans. X says how many relationships are within a given complex object, versus between complex objects. If the data file were clustered by some other criterion, or randomly, there would be no locality.

We now describe the cost formulas used in the analytic model. We present the (much simpler) formulas for when the id table fits in memory. We used 8 byte OIDs (this is the size used by Shore), so each two-pass and resolve-early id table entry is 12 bytes; each assign-early id table entry is 16 bytes (including the creation order); and the clearing algorithms' id table entries have an additional 4 bytes for the page numbers needed to check if an object is in the buffer pool. The parameters used in the cost of each algorithm are listed in Table 1.

The cost for each algorithm is now as follows:

$$
\begin{aligned}
naive &= 2 * P_{file} + 3 * P_{db} + 2 * P_{immedupdates} \\
P_{immedupdates} &= N_{objs} * N_{invrel} * Pb_{notinmem} \\
Pb_{notinmem} &= 1 - [(x * \frac{S_{mem} - S_{idtable}}{S_{db} * y}) + ((1-x) * \frac{S_{mem} - S_{idtable} - (S_{db} * y)}{S_{db} * (1-y)})] \\
S_{idtable} &= S_{identry} * N_{obj}
\end{aligned}
$$

Variable	Meaning
P_{file}	pages in data file
P_{db}	pages in database
S_{db}	size of database (bytes)
S_{mem}	size of memory (bytes)
$S_{identry}$	size of an id table entry (bytes)
$S_{idtable}$	size of id table (bytes)
P_{todo}	pages in todo list
$P_{invtodo}$	pages in inverse todo list
$P_{clrtodo}$	pages in cleared todo list
$P_{clrinvtodo}$	pages in cleared inverse todo list
N_{objs}	number of objects to load
N_{rel}	number of relationships per object
N_{invrel}	average number of inverse relationships per object
x	% relnships to nearby objects
y	% database considered nearby
z	% database in buffer pool
$P_{immedupdates}$	pages read into memory by immediate inverse updates
$Pb_{notinmem}$	probability that a page is not in memory
Pb_{notclr}	probability that a todo entry is not cleared
$Pb_{invnotclr}$	probability than an inverse todo entry is not cleared

Table 1: Parameters of the cost model.

Naive's file cost is for reading the data file twice; the database cost is for creating the database and then updating (reading and writing) all the objects, one page at a time.

The cost for the immediate inverse updates is more complicated. The number of updates is simply $N_{obj} * N_{invrel}$. However, an I/O is only incurred when the updated object is not in the buffer pool. We calculate a probability that the object is not in the buffer pool based on the locality parameters x and y, and use that to determine the number of I/Os incurred.

$$smart - invsort = 2 * P_{file} + 3 * P_{db} + 4 * P_{invtodo}$$

$$P_{invtodo} = N_{obj} * N_{invrel}$$

Smart-invsort's inverse todo list cost involves writing the inverse todo list out to disk, reading it back in and writing out sorted runs, and then reading and merging the runs to produce the sorted list. If the sort required an extra merge pass, the cost would be $6 * P_{invtodo}$.

The size of the inverse todo list is bounded by the number of inverse relationships per object. Since all inverse relationships are entered onto the inverse todo list, the size of the inverse todo list is thus the same as its upper bound.

$$late - invsort = 2 * P_{file} + P_{db} + 4 * P_{invtodo}$$

The cost for late-invsort is the same as for smart-invsort, except that it does not need to update the database after creating it.

$$res - early - invsort = P_{file} + 3 * P_{db} + 2 * P_{todo} + 4 * P_{invtodo}$$
$$P_{todo} = N_{obj} * N_{rel} * 0.5$$

Res-early-invsort reads the data file only once. However, it incurs the cost of writing and reading both a todo list and a inverse todo list. The inverse todo list cost is the same as for smart-invsort. The size of the todo list is bounded by $N_{obj} * N_{rel}$. However, on average, only half of the references from each object will be to objects described later on in the data file. We therefore model the size of the todo list as one-half the potential number of entries.

$$assign - early - invsort = P_{file} + 3 * P_{db} + 4 * P_{invtodo}$$

Assign-early-invsort does not use a todo list, since it pre-assigns OIDs whenever an unresolved surrogate appears. The inverse todo list cost is the same as for smart-invsort.

$$res - clear - invclear = P_{file} + 3 * P_{db} + 2 * P_{clrtodo} + 4 * P_{clrinvtodo}$$
$$P_{clrtodo} = N_{obj} * N_{rel} * Pb_{notcleared}$$
$$Pb_{notcleared} = [(x * \frac{y - z}{y}) + ((1 - x) * (\frac{1 - z}{1 - y}))] * 0.5$$
$$z = \frac{S_{mem} - S_{idtable}}{S_{db}}$$
$$P_{clrinvtodo} = N_{obj} * N_{invrel} * Pb_{invnotcleared}$$
$$Pb_{invnotcleared} = (x * \frac{y - z}{y}) + ((1 - x) * (\frac{1 - z}{1 - y}))$$
$$assign - early - invclear = P_{file} + 3 * P_{db} + 4 * P_{clrinvtodo}$$

The costs for the inverse-clear algorithms are superficially the same as for their inverse-sort counterparts. The difference lies in the size of the todo and inverse todo lists. Since some of the todo list entries are removed when the todo list is cleared, the cleared todo list and cleared inverse todo list are significantly smaller than their non-cleared counterparts.

When the entire database fits in the buffer pool, the sizes of the todo list and the inverse todo list drop to zero, since all entries will be cleared. At the other extreme, when the buffer pool holds only the id table, no entries are cleared. In between, the percentage of the database in the buffer pool is used in conjunction

with the locality to determine how many entries can be cleared. Since each entry will be checked for clearing shortly after it is created, the probability of clearing the entry is much greater if the object being referenced (in the case of the todo list) or the object to be updated (in the case of the inverse todo list) is physically nearby the object that generated the todo or inverse todo entry in the database, and therefore in the buffer pool at the same time. We model writing each todo list entry out to disk at the same time as the object that generated that entry is flushed from the buffer pool. Hence, the formulas for clearing the todo and inverse todo lists are very similar.

We note that only the algorithms that try to update objects in a random order are affected by the locality of reference. For this purpose, random means any order that is not the same as the data file order. Thus, naive, res-clear-invclear and assign-early-invclear are affected by locality, and by the size of the buffer pool, while smart-invsort, late-invsort, res-early-invsort, and assign-early-invsort are not.

We also note that the I/O cost of naive is a multiple of the number of objects and the number of inverse relationships. For all the other algorithms, the cost is linear in the number of objects when the id table fits in memory. (When the id table does not fit, the cost is also a multiple of the number of objects and the number of relationships.)

4 Data file and system parameters

For most of the analytical and implementation experiments we used 200 byte objects. Each 200 byte object had 10 slots for relationships, and 10 slots for inverse relationships to it. Additionally, each object had a 40 byte string field. We varied the number of objects to control the size of the database. The 5 Mb database has 25,000 objects; the 20 Mb database has 100,000 objects. The data file for the 5 Mb database was actually 2.3 Mb. We varied the locality of reference from no locality to having 90% of references stay within the nearest 10% of the database (hence called 90-10 locality). In the implementation experiments, the locality was built into the actual references in the data file. In the analytic experiments, it was a parameter.

5 Analytic model results

For the first set of experiments with the analytic model, we varied the amount of memory available for the load. In Figures 8 and 9, we show the predicted number of I/Os to load a 5 Mb database with 90-10 locality. We varied the memory available from 0.5 Mb to 10 Mb. At 10 Mb, the entire database plus all auxiliary data structures, such as the inverse todo list, fit in memory.

Figure 8 illustrates how much worse the naive algorithm performs relative to the others until the entire database fits in memory; when the buffer pool holds only 10% of the database, naive performs a full order

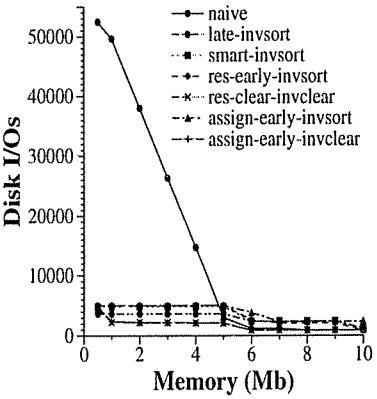

Figure 8: 5 Mb database with 90-10 locality.

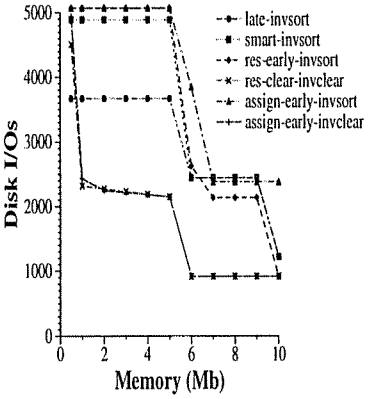

Figure 9: 5 Mb database with 90-10 locality, without naive.

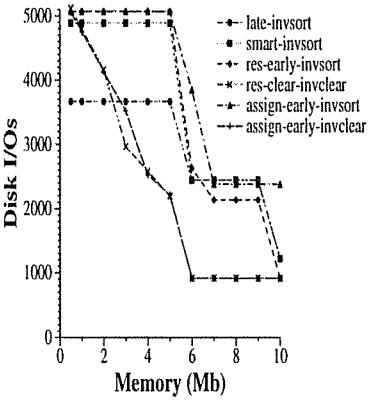

Figure 10: 5 Mb database with no locality.

of magnitude worse. Figure 9 shows the differences in performance among the remaining algorithms. At 10% of the database, or 0.5 Mb of memory, late-invsort is the best algorithm. Once 20% of the database, or 1 Mb, fits in memory, the clearing algorithms outperform the non-clearing algorithms. This is due to their writing and reading much smaller versions of the todo list and inverse todo list. When both a todo list and an inverse todo list are needed, res-clear-invclear is able to perform as well as assign-early-invclear because the updates dictated by both lists are merged in the same pass over the database. Late-invsort continues to dominate the non-clearing algorithms. Smart-invsort performs comparably to res-early-invsort. Although smart-invsort does not create a todo list, it incurs approximately the same number of I/O's because it reads the data file a second time.

When there is no locality of reference among the objects, late-invsort outperforms over the clearing algorithms until approximately half the database fits in memory, as shown in Figure 10. The relative performance of the other algorithms remains the same. However, while the non-clearing algorithms are unaffected by the locality, the clearing algorithms perform significantly worse, because fewer of the todo list and inverse todo list entries update objects that are in the buffer pool when the entry is generated. We do not show naive's performance in this graph because it is so much worse that the other algorithms appear as a single line on the graph. Relative to the other algorithms, naive now performs two orders of magnitude worse! With 1 Mb of available memory, naive requires 427,000 I/O's, while late-invsort performs merely 3,700 and res-clear-invclear only 4,800.

In some cases, such as when an OODB is dumped to a file and then reloaded, it is possible to dump both halves of an inverse relationship. That is, instead of storing only the fact that A has an inverse relationship with B in the data file, and letting the load algorithm take care of storing the relationship from B to A, it is possible to indicate both the relationship from A to B and the relationship from B to A explicitly in the data file. That way, the load algorithm does not need to perform any inverse updates. Also, in some

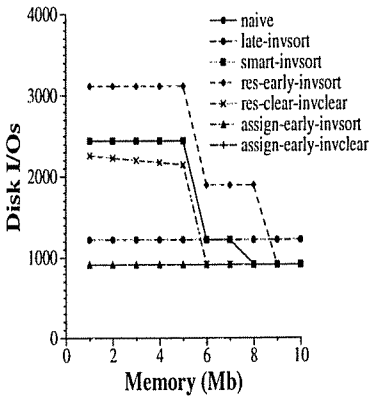

Figure 11: 5 Mb database with 20 relationships and 0 inverses.

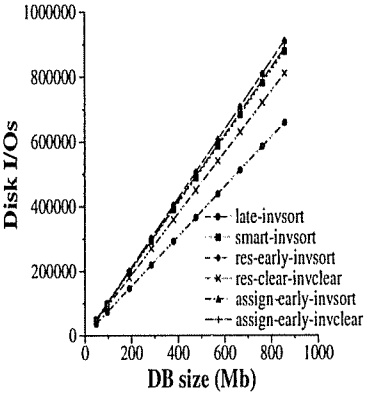

Figure 12: Scaling database size to 1 Gb, with 10% in memory.

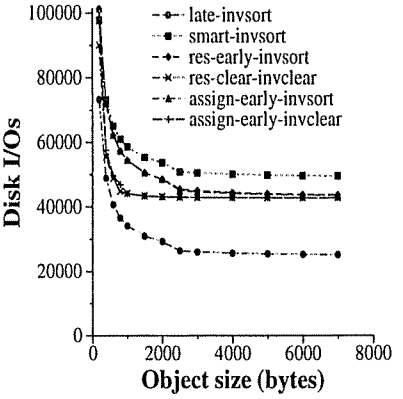

Figure 13: Scaling object size for 5 Mb database.

schemas, there are no inverse relationships. We therefore test the algorithms' performance for a data file containing twice as many relationships, to represent both halves of an inverse relationship but no implicit inverse relationships, in Figure 11. For all the algorithms, the performance was improved two-to-fourfold. The assign-early algorithms achieved the best performance possible: since they resolve all surrogates to OIDs on the first pass over the database, they did not need an second (update) pass over the database. Naive and smart-invsort appear as a single line, since they differ only in their handling of inverse updates. Res-clear-invclear performs slightly better than smart-invsort because the cost of writing and reading the cleared todo list is less than that of rereading the data file; res-early-invsort performs slightly worse for the opposite reason.

In the next experiment, shown in Figure 12, we scale the database size from 5 Mb to 1 Gb, while keeping the buffer pool size equal to 10% of the database. We chose 10% since we do not expect more than that to be available for loading massive amounts of data. All the other parameters are the same as before. We verify with this experiment that the relative performance of the algorithms does not change as we scale the database, and that with a corresponding increase in the buffer pool, the increase in I/O cost for the algorithms (except naive) is linear.

For the final experiment, shown in Figure 13, we held the database size constant and varied the object size from 200 bytes to 8 Kb, the size of a Shore page. To keep the database size constant, we decreased the number of objects as we increased the objects' size. For this test, we used a 100 Mb database with a 10 Mb buffer pool. Although the relative performance of the algorithms does not change, as the objects get larger the individual performance of each algorithm improves. There are two reasons why the corresponding decline in object size causes the improved performance: First, the id table shrinks and so more of the database fits in the buffer pool. Second, the absolute number of relationships declines, and so the size of the todo and inverse todo lists also declines.

5.1 Discussion

According to the analytic model, the relative ranking of the algorithms is late-invsort, followed closely by assign-early-invclear and res-clear-invclear, when there is a relatively small buffer pool available, and the opposite when most of the database fits in the buffer pool. Also, the clearing algorithms perform better when there is a higher locality of reference. They are then followed by assign-early-invsort and then res-early-invsort and smart-invsort, and this ranking is fairly consistent regardless of the locality in the data file or the number of objects or relationships. Naive, on the other hand, performs very poorly in the presence of inverse relationships, unless the entire database fits in memory. At that point, it doesn't really matter which algorithm is used.

The resolve-early and assign-early algorithms have the added benefit that since they only read the data file once, they can read the data file from a pipe. Therefore, if the program generating the data produces it in the data file format, the data file need never be physically stored. This can be very important when disk space is tight, because the data file tends to be the same order of magnitude as the database it describes.

All the algorithms cost significantly less when there are no inverse relationships. However, we have already noted that most commercial OODB systems (Ontos, Objectivity, Versant, ObjectStore) today support inverse relationships and sometimes it is not feasible to generate both halves of the relationship for the data file. For example, a dumped relational database would have foreign keys in one relation for one-half of the relationship, but the other relation would most likely store nothing that references the first relation. In addition, explicitly storing twice as many relationships in the data file can substantially increase the size of the data file and may not be a viable option when disk space is at a premium. Furthermore, when the load utility handles inverse relationships, it also handles all the referential integrity checks for the inverse relationships. The cost of doing first a load, and then referential integrity checks, would be much higher than doing the checks as part of the load. If the data to be stored contains no relationships at all, this study does not apply.

Although we do not present the results for loads when the id table does not fit in the buffer pool, we note that the I/O cost greatly increases: we do an insert in the id table for each object, and a lookup for each relationship and inverse relationship. When each of these inserts and lookups causes a I/O for the correct id table page, the cost skyrockets to the same magnitude as the naive algorithm, for all algorithms. For example, the predicted cost for late-invsort for a 5 Mb database is only 4,900 I/Os with 0.5 Mb of memory, which just barely holds the id table, but 524,000 I/Os with 0.1 Mb of memory. All of the algorithms exhibit similar one-hundred-fold increases in cost. Therefore, we recommend enough memory to store the id table as the minimum amount of memory that should be made available to the load. This limitation does not absolutely constrain the amount of data that can be loaded at one time, but rather the number of objects

that may be loaded: a data file containing 1 Gb of 8 Kb objects builds an id table of only 2 Mb.

6 Implementation

We ran all seven loading algorithms on a Hewlett-Packard 9000/720 with 32 Mbytes of physical memory. However, we were only able to use about 16 Mb for any test run, due to operating system and daemon memory requirements. The database was stored under the Shore storage manager [CDF+94] on a raw Seagate ST-12400N disk controlled exclusively by Shore. The data file resided on a separate disk on the local file system, and thus did not interfere with the database I/O. For these tests, we turned logging off. It is important to be able to turn off logging when loading a lot of new data [Moh93a]; we found that when we used full logging, the log outgrew the database. It is unlikely that users have enough disk space to accomodate such a log.

We used Shore as the underlying persistent object manager, even though Shore is still under development, for two reasons. First, Shore provides the notion of a "value-added server" (VAS), which allowed us to place the load utility directly in the server. We feel that this is the best place for a load utility; the client-server communication overhead is greatly reduced. The implementors of DB2 experienced significantly better performance when the load utility interacted directly with the buffer manager, instead of as a client [Moh93b]. Additionally, the load algorithms have direct access to the server buffer pools and can determine what is in the buffer pool at any given time, which was needed by the algorithms that try to clear the todo list and inverse todo list. The non-clearing algorithms, however, could be implemented at the client level.

Second, Shore provides logical OIDs, which we needed to test the late-invsort and assign-early algorithms, as well as physical OIDs. Shore uses a logical OID index to maps from logical OIDs to physical OIDs. This index is stored in the database.

We stored the todo list as a single large object, and the inverse todo list as several large objects, since they are too large to keep in main memory. The id table is implemented as a open addressing hash table, hashed on the surrogate. Our code for all the load algorithms combined was about 5000 lines of C++ code, and took one person only one month to write.

6.1 Experimental Results

We ran experiments to load a database with 5, 20, and 50 Mb of data. All the objects were 200 bytes and we increased the number of objects to increase the database size. Due to metadata overhead and Shore's logical OID index, the databases created were actually 7, 27, and 66 Mb. The memory used by each test reflects the sum of the id table (in transient memory) and the buffer pool, since in the analytic model we

did not distinguish between the two.

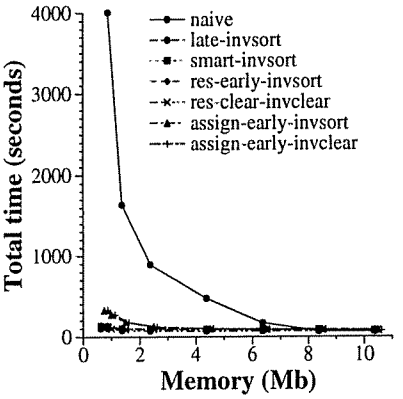

Figure 14: 5 Mb database with 90-10 locality.

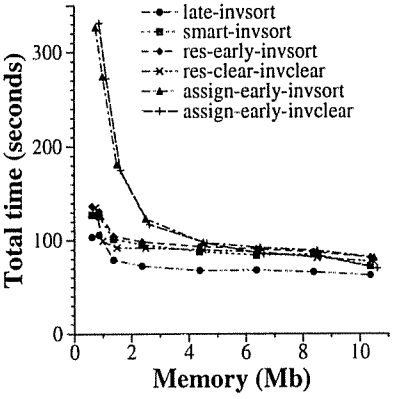

Figure 15: 5 Mb database with 90-10 locality, without naive.

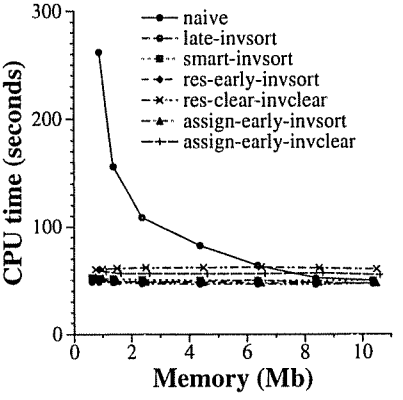

Figure 16: 5 Mb database: CPU time.

For the first set of experiments, we created a database with 5 Mb of data, which was actually 7 Mb when created and hence first fits in the buffer pool at 7 Mb. In the first experiment, shown in Figure 14, we loaded a 5 Mb of data with 90-10 locality. As predicted by the analytic model, the times for the naive algorithm dominate by an order of magnitude. We therefore present the results again without naive in Figure 15. The anomalous performance of the assign-early algorithms with a small buffer pool is caused by the logical OID index. The two-pass and res-early algorithms assign OIDs to objects as the objects are created, and hence the OIDs are inserted into the logical OID index in clustered order. The assign-early algorithms, in direct contrast, assign OIDs to objects as the objects' surrogates are encountered. As the objects are created, their OIDs are entered in the logical OID index in a random order (i.e., not clustered by OID). Since the logical OID index did not fit in the buffer pool, each object creation caused (on average) an extra disk I/O to insert the OID into the index.

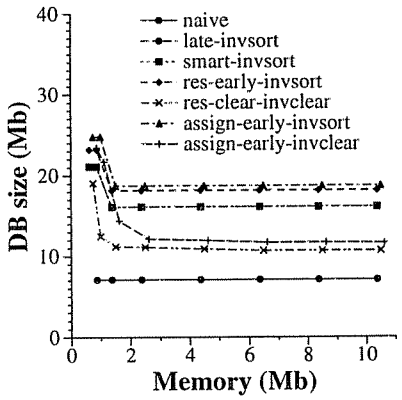

Figure 17: 5 Mb database: Disk space used by database and todo lists.

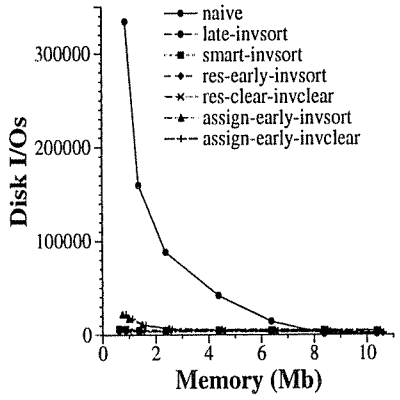

Figure 18: 5 Mb database: Disk I/Os.

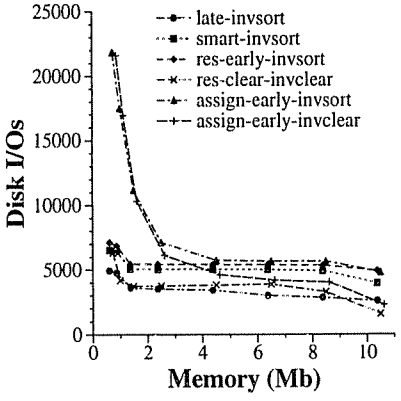

Figure 19: 5 Mb database: Disk I/O, without naive.

In all cases, late-invsort is the fastest algorithm. As the buffer pool grows to hold nearly the entire

database, we see the most improvement in performance by the algorithms that take advantage of the contents of the buffer pool, namely, the clearing algorithms, assign-early-invclear and res-clear-invclear. However, the improvement is not as dramatic as the analytic model predicts, and hence late-invsort is still better. This difference is explained by the relative CPU costs of the algorithms, shown in Figure 16. The clearing algorithms perform significantly more work to check the buffer pool for each entry on the todo and inverse todo list. In addition, while clearing an entry has no associated I/O cost, there is a fair amount of overhead involved in pinning the corresponding object in the buffer pool and updating it. The clearing algorithms pin the object for each "free" update. The updates done in the second phase, however, only pin each object once, no matter how many updates to a given object there are.

Figure 17 shows the amount of disk space needed by each algorithm. The includes the size of the database, the logical OID index, and the auxiliary data structures (the todo list and inverse todo list) used. (The auxiliary data is deleted at the end of the load.) Naive uses the least amount of disk because it has no auxiliary structures. For the 5 Mb database, the logical OID index accounts for approximately 1.5 Mb of the 7 Mb stored. Like the size of the id table, the size of the logical OID index corresponds to the number of objects, rather than the absolute size of the database.

In Figure 18 we show the I/O cost of each algorithm; in Figure 19 we repeat the results without the naive algorithm. Except for the anomalies in the assign-early algorithms with a small buffer pool, due to the logical OID index, we note that the actual I/O cost of each algorithm is extremely close to the I/O cost predicted by our analytic model. For example, in Figure 9 we predicted 3597 I/Os for late-revsort with 1 Mb memory. In our experiment, late-revsort took 3667 I/Os, which is less than a 5% deviation.

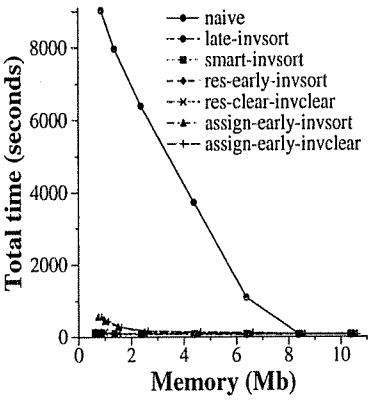

Figure 20: 5 Mb database with no locality.

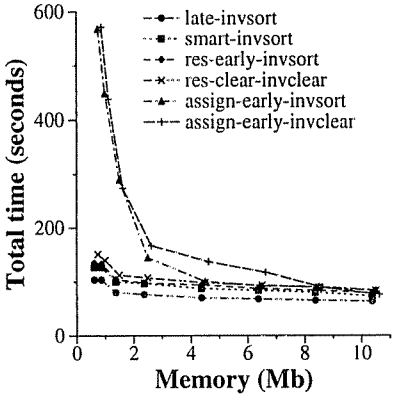

Figure 21: 5 Mb database with no locality, without naive.

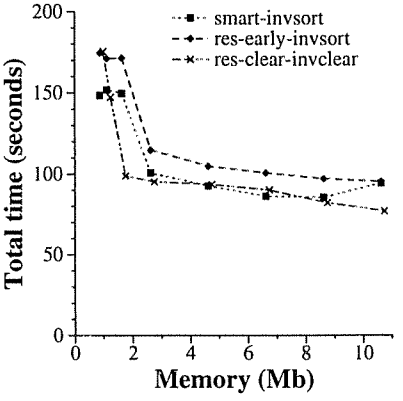

Figure 22: 5 Mb database with physical OIDs.

We next experimented with a 5 Mb data file with no locality of reference. As we predicted in the analytic model, naive becomes an even worse choice, taking 2 hours to complete the load with 1 Mb of memory, and 1 hour with 4 Mb. All the other algorithms, in contrast, take 1 to 2 minutes. The relative performance

of the algorithms is similar to that with 90-10 locality, but the assign-early algorithms pay an even greater penalty for inserting into the logical OID index out of order.

We therefore decided to run some experiments to see how the algorithms perform with physical OIDs. Figure 22 show the results of these experiments. Late-invsort and assign-early depend on logical OIDs and could not be run; we also omitted naive. Contrary to our expectations, the tests with physical OIDs took *longer* to run than their logical OID counterparts. In Shore, logical OIDs are 8 bytes but physical OIDs are 12 bytes. The size of the objects thus grew from 200 bytes to 280 bytes to store the same information. The physical OID tests thus incurred many more I/Os to create the database, and since I/O costs dominate loading, the physical OID tests were slower.

In Figure 23 we show how the size of the database, todo list, and inverse todo list grew; where we needed 16 Mb of disk space for smart-invsort with logical OIDs, we needed 19 Mb with physical OIDs. The actual database grew from 7 Mb (including the logical OID index) to 7.9 Mb.

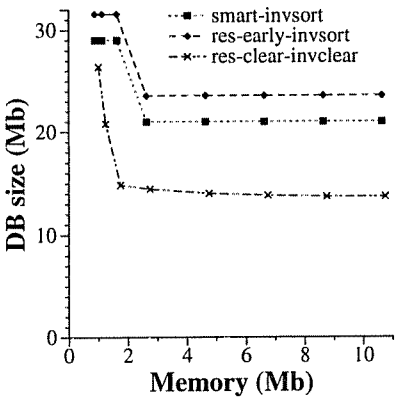

Figure 23: DB size, including auxiliary structures, with physical OIDs.

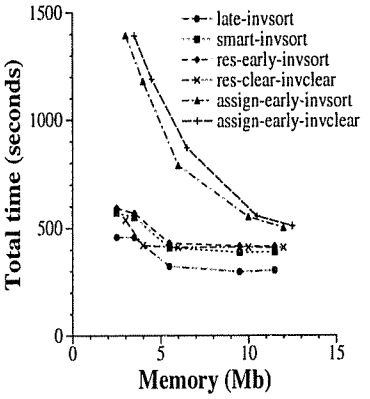

Figure 24: 20 Mb database with 90-10 locality.

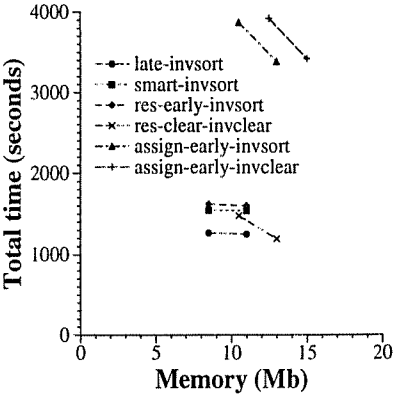

Figure 25: 50 Mb database with 90-10 locality.

Figures 24 and 25 show the results of loading 20 Mb and 50 Mb of data, respectively, with 90-10 locality in the data file. We present these graphs primarily to show that the performance of the algorithms scales as we increase the amount of data to load. Note, however, that because of the 16 Mb physical limitations on combined buffer pool and heap memory size for the load process, we could test only a small and medium buffer pool for 20 Mb, and only a small buffer pool for 50 Mb.

Although we do not present the graphs, when we ran experiments with 20 relationships but no inverse relationships, we found that the analytic model was correct: all the algorithms run much faster. For example, late-invsort loaded the 5 Mb database with 0.9 Mb of Memory in 38 seconds; naive took 50 and res-early-invsort ran in 67 seconds. The fastest time to load the same database with inverse relationships was 105 seconds.

6.2 Discussion

The implementation results confirm that the analytic model predicts the actual disk I/Os for each algorithms accurately. However, because the analytic model does not account for CPU time, and did not take such factors as the logical id index under consideration, it is only a moderate predictor of actual algorithm performance.

For example, although the analytic model predicted that assign-early-invclear would sometimes beat late-invsort, the logical OID index imposed substantial I/O overhead for assign-early and thus it did not perform well. While the analytic model predicted that res-clear-invclear would beat late-invsort with high locality and a medium or large buffer pool, the CPU time involved in clearing made up for the savings in disk I/Os, and late-invsort still proved faster.

Late-invsort is the clear best choice according to the implementation results. Of the other algorithms, both smart-invsort and res-early-invsort are good choices. Neither is affected much by the logical OID index, and neither wastes CPU time trying to clear entries on the todo lists when there are very few objects in the buffer pool that could be updated. We expect that res-early-invsort will be better when there are relatively few relationships in the data file, e.g., if much of the file describes images or other bulk data. There is not much advantage to implementing the more complicated res-clear-invclear. We therefore recommend that users implement late-invsort if pre-assigning of OIDs is possible, and either smart-invsort or res-early-invsort if not.

7 Conclusions

A bulk loading utility is critical to users of OODBs with significant amounts of data. These users include those switching from a relational or hierarchical database; those switching OODB products; those who want to recluster their OODB data for better performance; and scientists running applications that continually generate vast amounts of new data. However, loading in an OODB may be very slow due to relationships among the objects; inverse relationships exacerbate the problem. In our performance study we showed that the best algorithms solve the problems due to relationships by (1) using a sorted inverse todo list to avoid random reads and updates and (2) using pre-allocation of OIDs to avoid updates in the first place. Of the algorithms we explored, we recommend that users implement late-invsort if logical OIDs are available, and either smart-invsort or res-early-invsort otherwise. We also note that naive's abominable performance is to be expected if objects are loaded one at a time, e.g., by **insert** or **new** statements, since inverse updates cannot be batched.

Our future work includes running experiments that load 1 Gb of data (when we get a larger disk to

store the database); looking at techniques to decrease the cost of loading when the id table does not fit in memory; creating techniques to connect newly loaded objects to existing objects in the database; investigating algorithms for a loading in parallel on one or more servers with multiple database volumes; looking into reclustering algorithms that dump and then reload objects; and adding smart integrity checking to the load algorithms. In addition, we plan to integrate the load implementation with the higher levels of Shore and turn it into a utility to be distributed with a future release of Shore.

8 Acknowledgements

We would like to thank David Maier for the original inspiration to study loading and for feedback on our early algorithm ideas; Mike Zwilling and C. K. Tan for advice and support for our implementation as a Shore value-added server; and Mark McAuliffe, Praveen Seshadri, Yannis Ioannidis, C. Mohan, and Dan Weinreb for their helpful comments on the content and presentation of this paper.

References

[Cat93] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan-Kaufman, Inc., San Mateo, CA, 1993.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394, 1994.

[CMR92] J. B. Cushing, D. Maier, and M. Rao. Computational Proxies: Modeling Scientific Applications in Object Databases. Technical Report 92-020, Oregon Graduate Institute, December 1992. Revised May, 1993.

[DLP+93] R. Drach, S. Louis, G. Potter, G. Richmond, D. Rotem, H. Samet, A. Segev, and A. Shoshani. Optimizing Mass Storage Organization and Access for Multi-Dimensional Scientific Data. In *Proceedings of the IEEE Symposium on Mass Storage Systems*, Monterey, CA, April 1993.

[Mai94] David Maier, January 1994. Personal communication.

[Moh93a] C. Mohan. A Survey of DBMS Research Issues in Supporting Very Large Tables. In *Proceedings of the International Conference on Foundations of Data Organization and Algorithms*, pages 279–300, Chicago, Il., 1993. Springer-Verlag.

[Moh93b] C. Mohan. IBM's Relational DBMS Products: Features and Technologies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 445–448, 1993.

[Nel91] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Obj92] Objectivity, Inc. *Objectivity/DB Documentation*, 2.0 edition, September 1992.

[OHMS92] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query Processing in the ObjectStore Database System. In M. Stonebreaker, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 403–412, 1992.

[Ont92] Ontos, Inc. *Ontos DB Reference Manual*, release 2.2 edition, February 1992.

[PG88] N. W. Paton and P. M. D. Gray. Identification of Database Objects by Key. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems: 2nd International Workshop on Object-Oriented Database Systems*, pages 280–285, Berlin, Germany, September 1988. Springer-Verlag.

[Sho93] A. Shoshani. A Layered Apporach to Scientific Data Management at Lawrence Berkeley Laboratory. *IEEE Data Engineering Bulletin*, 16(1):4–8, March 1993.

[Sno89] R. Snodgrass. *The Interface Description Language: Definition and Use.* Computer Science Press, 1989.

[Veg86] S. R. Vegdahl. Moving Structures between Smalltalk Images. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 466–471, 1986.

[Ver93] Versant Object Technology. *Versant Object Database Management System C++ Versant Manual*, release 2 edition, July 1993.

[WI93] J. L. Wiener and Y. Ioannidis. A Moose and a Fox Can Aid Scientists with Data Management Problems. In *Proceedings of the International Workshop on Database Programming Languages*, pages 376–398, New York, NY, 1993. Springer-Verlag.