DECOMPOSITION OF STRUCTURED LARGE-SCALE OPTIMIZATION PROBLEMS AND PARALLEL OPTIMIZATION

by

Deepankar Medhi

Computer Sciences Technical Report #718

September 1987

Decomposition of Structured Large-scale Optimization Problems and Parallel Optimization

by

DEEPANKAR MEDHI

A thesis submitted in partial fulfillment of the requirement for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

1987

To my school teacher,

Srijut Harendra Nath Das

Abstract

DECOMPOSITION OF STRUCTURED LARGE-SCALE

OPTIMIZATION PROBLEMS

AND

PARALLEL OPTIMIZATION

Deepankar Medhi

Under the supervision of Professor Stephen M. Robinson

In this dissertation, we present serial and parallel algorithms to solve efficiently the problem : inf $\{\sum_{i=1}^{N} f_i(x_i) \mid \sum_{i=1}^{N} A_i x_i = a\}$. Here, $a \in \mathbb{R}^m$, and for $i = 1, \ldots, N$, $A_i \in \mathbb{R}^{m \times n_i}$, $x_i \in \mathbb{R}^{n_i}$, and, f_i's are closed proper convex functions (not necessarily differentiable) taking values in the extended real line $(-\infty, \infty]$. For example, block-angular linear programming problems and linear multi-commodity network optimization problems can be cast into the above form. In our approach, we take the Rockafellar dual of the problem to arrive at an unconstrained nonsmooth maximization problem. The difficulty arises from the nonsmoothness of the dual objective. Traditional subgradient methods are not good enough as they do not have implementable stopping criterion and are reported to have slow convergence. One also may not obtain a primal optimal solution at the

end. Instead, we apply a modified bundle algorithm, which has an implementable stopping criterion, and more importantly, one can recover an approximate primal optimal solution. We also obtain some theoretical *a posteriori* error information on the approximate solution. We have implemented this algorithm on randomly generated block-angular linear programming problems of size up to 4,000 equality constraints and 10,000 variables. Our implementation ran up to seventy times faster than MINOS version 5.0, and did substantially better than an advanced implementation of the Dantzig-Wolfe decomposition method. Thus we think that for this type of problem, our algorithm is very promising.

A nice feature of the dual problem is that it breaks up the original problem into smaller *independent* subproblems. Exploiting this fact, we present parallel algorithms implemented on the CRYSTAL multicomputer. We considered two groups of test problems for these algorithms, one in which the subproblems required approximately equal amounts of time to solve, and another in which the solution times varied. In the first group, we obtained 70% - 80% efficiency with up to eleven processors. In the second group, we obtained 60% or more efficiency with relatively small problems and with up to five processors.

Acknowledgements

I would like to express my sincere thanks to my advisor, Professor Stephen M. Robinson, for his advice, encouragement and support in the preparation of this dissertation and also throughout my graduate studies.

I would also like to thank Professors Olvi L. Mangasarian, Robert R. Meyer, James G. Morris and Jerry L. Sanders for serving on the Examination Committee. In addition, I would like to thank Dr. C. Lemaréchal, Dr. J. K. Ho and Renato De Leone for providing me with computer software which aided me in my research.

My sister, Rumjhum's letters kept me informed of what was happening on the other side of the world. Thank you, Rumjhum, for your nice letters. Also, thanks to my parents, my brother, Bablu, and my other sister, Moonmoon, for their support.

This dissertation is dedicated to my school teacher, Srijut Harendra Nath Das, whose guidance in the early stage of my schooling has always been of great help to me, even today.

Finally, thank you, Karen, for everything.

This research was supported by the National Science Foundation under grant DCR-8502202.

Contents

ABSTRACT iii

ACKNOWLEDGEMENTS v

CHAPTER 1: INTRODUCTION 1

1.1 Overview 3

1.2 Notations and Definitions 6

CHAPTER 2: EXAMPLES AND PREVIOUS WORK 11

2.1 The Problem 12

2.2 Block-angular Linear Programming Problem 13

2.3 Two-stage Stochastic Linear Programming 14

2.4 Linear Multi-commodity Network Problem 17

2.5 A Doubly-coupled Linear Program 18

2.6 Previous Work 19

2.7 What is Ahead 21

CHAPTER 3: A DECOMPOSITION ALGORITHM
BASED ON THE BUNDLE METHOD 22

3.0 Introduction 22

3.1 A Decomposition Method 23

3.2 Ha's Approach 26

3.3 Subgradient Methods 28

3.4 The Bundle-based Approach 29

3.5 The Algorithm 42

3.6 Summary 46

CHAPTER 4: BLOCK-ANGULAR LINEAR PROGRAMMING PROBLEM:
IMPLEMENTATION AND COMPUTATIONAL EXPERIENCE 47
4.1 Block-angular Linear Programming 48
4.2 The Algorithm 50
4.3 Implementation of the Algorithm 53
4.4 Generation of Test Data 60
4.5 Computational Experience 60
4.5.1 Experiment I 61
4.5.2 Experiment II 84
4.6 Conclusion 101

CHAPTER 5: PARALLEL ALGORITHMS:
IMPLEMENTATION AND COMPUTATIONAL EXPERIENCE 102
5.1 Parallel Algorithms 104
5.1.1 Subproblems of Even Size 106
5.1.2 Uneven Subproblems 110
5.2 The CRYSTAL multicomputer and SAP 116
5.3 Implementation on CRYSTAL 121
5.4 Computational Experience 124
5.4.1 Case I 125
5.4.2 Case II 134
5.5 Summary 139

CHAPTER 6: A DOUBLY-COUPLED LINEAR PROGRAM 140
6.1 The First Approach 141
6.2 The Second Approach 147

APPENDIX 150

BIBLIOGRAPHY 154

Chapter 1

INTRODUCTION

Large-scale optimization problems arise in various real-life situations, for example airline scheduling, transportation, logistics, production and economic decision making. Because of immense size of these problems a general algorithm that is used for a smaller-scale problem may not be suitable for these larger problems. However, very often large-scale optimization problems have special structure. Since a general algorithm for a smaller-scale problem may not be able to use effectively the special nature of a large problem in solving it, one hopes to arive at a specialized algorithm by exploiting the structure of the large-scale problem and one again hopes this algorithm is better than general algorithms or other existing algorithms. For example, multi-period modeling gives rise to a linear programming problem whose technology matrix has block-angular structure or staircase structure. Using the specific information about the technology matrix, special algorithms have been proposed to solve these optimization problems; for example, the Dantzig-Wolfe decomposition algorithm ([Dan60], [Dan61a]), and the nested decomposition algorithm [Ho74].

In this dissertation, we present sequential and parallel algorithms to solve efficiently a certain type of large-scale optimization problem. This type of large-scale problem can be put in the following form :

$$\inf_{x_1,\ldots,x_N} \quad \sum_{i=1}^{N} f_i(x_i) \tag{1.1a}$$

subject to

$$\sum_{i=1}^{N} A_i x_i = a, \tag{1.1b}$$

where $a \in \mathbb{R}^m$, and for $i = 1,\ldots,N$, $A_i \in \mathbb{R}^{m\times n_i}, x_i \in \mathbb{R}^{n_i}$, and, f_i's are closed proper convex functions taking values in the extended real line $(-\infty,\infty]$. We shall call the constraints (1.1*b*) *coupling constraints*.

Our interest in this type of problem arises from the following three facts.

1) Several well-known optimization problems, like the block-angular linear programming problem ([Dan60], [Dan61a]), the discretized stochastic linear programming problem with recourse [Kal79] and the linear multicommodity network optimization problem [Ken80] can be expressed in this form.
2) Usually, problems of type (1.1) are attacked by the well-known Dantzig-Wolfe decomposition method ([Dan60], [Dan61a]). However, the computational experience with the Dantzig-Wolfe decomposition method is somewhat disappointing ([Las78], [Dir79], [Ho83]).
3) We can develop parallel algorithms which can exploit the special nature of these problem, and thus solve the problems considerably faster.

In the next section, we give an overview of the work that we have done in this dissertation where we have addressed these issues.

1.1 Overview

In Chapter 2, we give examples of problems mentioned in (1) above and show how they fit into the model (1.1). We also discuss previous work on solution of these problems.

We present, in Chapter 3, an alternative decomposition algorithm for the problem type (1.1). In our approach, we take the Rockafellar dual of the problem (1.1) to arrive at an unconstrained maximization problem. For this type of problem, this dualization results in a problem which is usually much smaller in dimension compared to the original problem. Though the fact that it is unconstrained tempts one to try some existing algorithms from nonlinear optimization, one cannot apply them as the transformed problem turns out to be nonsmooth. Another possibility is the traditional subgradient methods [Pol78]. However, the disadvantage with this approach is that it does not have an implementable termination criterion, and it is reported to have slow convergence. Moreover, though one may obtain a dual optimal solution, one may *not* be able to obtain a primal optimal solution at the end. Our approach is based on a relatively recent method, known as the bundle method ([Wol75], [Lem78a], [Lem81]), for nonsmooth optimization problems. In this approach, we apply a modified bundle algorithm to the nonsmooth dual problem. The advantage of this algorithm is that it has an implementable termination criterion, and more importantly, one can recover an approximate primal optimal solution. We also obtain some theoretical *a posteriori* error information on this approximate primal optimal solution. Unlike the

memoryless nature of gradient-type methods in smooth optimization, the bundle method needs to keep some information from previous iterations. However, the nice feature is that one does not need to store all the previous information, and one can work with finite storage by deleting some of the old information.

To investigate this algorithm and to compare it with existing solution methods, we have implemented it in Fortran 77 on block-angular linear programming problems ([Dan60], [Dan61a]). We present our experience in Chapter 4. First, we investigate this algorithm by varying factors of a block-angular linear programming problem, e.g., the size of the whole problem, number of coupling constraints and the number of subproblems (i.e., N of (1.1)). From our investigation, we found that if the problem size and N are kept fixed, then the solution time grows linearly as the number of coupling constraints increases. We observed that when the size of the whole problem and the number of coupling constraints are kept fixed, the time grows between linearly and quadratically as the density of the non-coupling part (which is essentially $1/N$) increases. We also observed that, contrary to the best known theoretical result that the bundle method is sublinear [Lem86a,p.229], it behaves in practice more like a *linearly* convergent method. For some of the randomly generated test problems, our implementation, which is still in an experimental stage, turns out to be in the range of fifteen to seventy times faster than the MINOS 5.0 package [Mur83]. Our implementation did substantially better than a sparse implementation of the sparsity preserving LPSOR2 algorithm [Man84] for

linear programming, and its proximal point variant ([Del85], [Del87]). Our implementation also did substantially better than an advanced implementation [Ho] of the Dantzig-Wolfe decomposition method. So we conclude that for this type of problem the bundle-based algorithm is very promising.

A nice feature of the decomposition technique described in Chapter 3 is that it breaks up the original problem, obtaining the dual as a collection of smaller *independent* subproblems. Exploiting this fact, in Chapter 5, we present parallel algorithms and our computational experience on the CRYSTAL multicomputer [DeW84] for the problem type (1.1), specifically, for block-angular linear programming problems. In that chapter, we briefly review the CRYSTAL multicomputer and the communication package, SAP, that we use for sending/receiving information to/from one processor to another and/or to/from the host machine, and give a description of how the parallel algorithms are implemented. If we call subproblems which take almost equal amounts of solution time *subproblems of even size*, and subproblems which take different amounts of solution time *uneven subproblems*, we can classify the problem (1.1) into two classes, and we have proposed different parallel algorithms for these two classes. In the first one, where the subproblems are of even size, the scheduling of the subproblems among different processors of a multicomputer is relatively simple. We allocate the subproblems by 'equally' dividing them among the available processors (For a precise statement of the 'equal' division, please refer to §5.1.1). On implementation of this scheme, we found that for large problems we have obtained about 70% - 80% efficiency with

up to eleven processors. For problems with uneven subproblems, we identified the problem of allocation of the subproblems to different processors with the *independent task-scheduling* problem [Gar78] for scheduling independent jobs in a number of identical processors. There are well-known algorithms available for the independent task-scheduling problem. However, these algorithms require knowledge of solution time of the subproblems beforehand, which we do not have in our case. Instead, we estimate the solution time of subproblems and then apply these algorithms. With this estimation approach, our implementations of these algorithms give 60 % or more efficiency with relatively small test problems and with up to five processors.

Finally, in Chapter 6, we propose an extension of the decomposition algorithm (of Chapter 3) to a doubly-coupled linear programming problem [Rit67]. In this approach, the bundle-based decomposition algorithm is applied at two stages.

In the next section, we give notations and definitions that will be used throughout this dissertation.

1.2 Notations and Definitions

Most of the definitions given in this section can be found, for example, in [Roc70]. We first give some notations.

$\mathbb{R}^n$ is the n-dimensional Euclidean space.

The usual inner product is denoted by $\langle \cdot, \cdot \rangle$, i.e., if $x, y \in \mathbb{R}^n$, then $\langle x, y \rangle := \sum_{\ell=1}^{n} x_\ell y_\ell$, where x_ℓ denoted the ℓ^{th} component of the vector x.

The Euclidean norm of vector x is given by $\| x \|_2$ or $\| x \|$. The p-norm of x is denoted by $\| x \|_p$.

For a vector $x \in \mathbb{R}^n$, x_+ denotes the vector whose ℓ^{th} component is x_ℓ if $x_\ell \geq 0$, and 0 if $x_\ell < 0$.

In literature, usually subscripts are used for component of a vector; but here, in most cases, $x_i \in \mathbb{R}^{n_i}$, i.e., x_i is an n_i-dimensional vector, and $x = (x_1, \ldots, x_N) \in \mathbb{R}^{n_1+\cdots+n_N}$.

The transpose of an $m \times n$ matrix A is denoted by A^T. Unless otherwise stated, the matrix A_i is an $m \times n_i$ matrix, and the matrix B_i is an $m_i \times n_i$ matrix.

If α is a real number, then we write $\lfloor \alpha \rfloor$ to denote the greatest integer less than or equal to α (the 'floor' of α), and we write $\lceil \alpha \rceil$ to denote the least integer greater than or equal to α (the 'ceiling' of α).

The end of the proof of a theorem or a lemma is denoted by ∎

Definition 1.1. *Convex set.* A subset C of $\mathbb{R}^n$ is convex if $\lambda x + (1-\lambda)y \in C$ for every $\lambda \in [0,1]$ and every $x, y \in C$.

Definition 1.2. *Convex hull.* The convex hull of a subset A of $\mathbb{R}^n$, written conv A, is the intersection of all convex sets containing A.

Definition 1.3. *Affine set.* A subset M of $\mathbb{R}^n$ is affine if for each $x, y \in M$ and any $\lambda \in \mathbb{R}$, $\quad \lambda x + (1-\lambda)y \in M$.

Definition 1.4. *Affine hull.* The affine hull of a subset S of $\mathbb{R}^n$, written aff S, is the intersection of all affine sets containing S.

Definition 1.5. *Relative interior.* Let C be a set in $\mathbb{R}^n$. The relative interior of C is

$$\text{ri } C = \{x \in \text{ aff } C \mid \text{ for some } \varepsilon > 0, (x + \varepsilon B) \cap (\text{aff } C) \subset C\},$$

where B is the Euclidean closed unit ball in $\mathbb{R}^n$.

Definition 1.6. *Effective domain of a function.* Let $C \subset \mathbb{R}^n$ and f be a function such that $f: C \to [-\infty, +\infty]$. The effective domain of f on C, written dom f, is given by

$$\text{dom } f := \{x \in C \mid f(x) < +\infty\}.$$

Definition 1.7. *Convex function.* Let $f: \mathbb{R}^n \to [-\infty, +\infty]$. Then f is convex if its epigraph, given by

$$\text{epi } f := \{(x, \mu) \mid x \in \mathbb{R}^n, \mu \in \mathbb{R}, \mu \geq f(x)\},$$

is a convex set. f is concave if $-f$ is convex.

Definition 1.8. *Proper convex function.* Let $f: \mathbb{R}^n \to [-\infty, +\infty]$ be a convex function. Then f is proper convex if $f(x) < +\infty$ for at least one $x \in \mathbb{R}^n$, and $f(x) > -\infty$ for every x.

Hereafter, we shall restrict ourselves to proper convex functions.

Definition 1.9. *Strongly convex function.* Let $f: \mathbb{R}^n \to (-\infty, +\infty]$. Then f is strongly convex with modulus δ if there exists $\delta > 0$ such that for each $x, y \in \mathbb{R}^n$ and each $\lambda \in (0, 1)$,

$$f((1-\lambda)x + \lambda y) \leq (1-\lambda)f(x) + \lambda f(y) - \delta\lambda(1-\lambda) \parallel x - y \parallel^2 .$$

Definition 1.10. *Closed proper convex function.* A proper convex function $f: \mathbb{R}^n \to (-\infty, +\infty]$ is closed if the epigraph of f is a closed set in $\mathbb{R}^{n+1}$.

Definition 1.11. *Conjugate of a convex function.* The conjugate of a convex function f, denoted by f^*, is given by

$$f^*(z) := \sup_{x \in \mathbb{R}^n} \{\langle z, x \rangle - f(x)\}.$$

Note that f^* is a closed convex function, and f^* is proper if and only if f is proper. If f is closed convex, then $f^{**} = f$, where f^{**} denotes the conjugate of the conjugate of f.

Definition 1.12. *Subdifferential of a convex function.* For a convex function f, the subdifferential of f at $\bar{x}$ is given by

$$\partial f(\bar{x}) := \{\pi \in \mathbb{R}^n \mid \forall\, x, \quad f(x) \geq f(\bar{x}) + \langle \pi, x - \bar{x} \rangle\}.$$

Definition 1.13. *Subdifferential of a concave function.* For a concave function g, the subdifferential of g at $\bar{y}$ is given by

$$\partial g(\bar{y}) := \{\pi \in \mathbb{R}^n \mid \forall\, y, \quad g(y) \leq g(\bar{y}) + \langle \pi, y - \bar{y} \rangle\}.$$

An element of the subdifferential set will be called a *subgradient.*

Definition 1.14. *ε-subdifferential of a convex function.* For a convex function f, the ε-subdifferential of f at $\bar{x}$ (where $\varepsilon \geq 0$) is given by

$$\partial_\varepsilon f(\bar{x}) := \{\pi \in \mathbb{R}^n \mid \forall\, x, \quad f(x) \geq f(\bar{x}) + \langle \pi, x - \bar{x} \rangle - \varepsilon\}.$$

Definition 1.15. *ε-subdifferential of a concave function.* For a concave function g, the ε-subdifferential of g at $\bar{y}$ (where $\varepsilon \geq 0$) is given by

$$\partial_\varepsilon g(\bar{y}) := \{\pi \in \mathbb{R}^n \mid \forall\, y, \quad g(y) \leq g(\bar{y}) + \langle \pi, y - \bar{y} \rangle + \varepsilon\}.$$

An element of the ε-subdifferential set will be called an *ε-subgradient.* Note that the 0-subdifferential is the standard subdifferential defined in Definitions 1.12 and 1.13.

Definition 1.16. *Locally Lipschitzian.* A function $h : \mathbb{R}^n \rightarrow \mathbb{R}$ is said to be locally Lipschitzian if for each bounded subset S of $\mathbb{R}^n$, there exists $L < \infty$ such that

$$|h(x^1) - h(x^2)| \leq L \parallel x^1 - x^2 \parallel, \text{ for } x^1, x^2 \in S.$$

Definition 1.17. *Weakly semi-smooth.* [Lem78b] A convex function f is weakly semi-smooth if it is locally Lipschitz continuous and for scalar t and $\pi \in \partial f(x+td)$, $\langle \pi, d \rangle$ has exactly one cluster point as $t \rightarrow 0^+$.

Chapter 2

EXAMPLES AND PREVIOUS WORK

In our introductory chapter, we mentioned that in this dissertation we investigate a decomposition algorithm and parallel algorithms for the problem (1.1). In latter chapters we develop these algorithms and present computational experience. In this chapter we give examples of problems that fit into the model (1.1). We also give an example which does not directly fit into the form (1.1), but whose subproblems have a structure similar to (1.1). Finally, we briefly discuss previous algorthims for solving these examples.

2.1 The problem

For convenience, we display the problem (1.1) again :

$$\inf_{x_1,\ldots,x_N} \quad \sum_{i=1}^{N} f_i(x_i) \tag{1.1a}$$

subject to

$$\sum_{i=1}^{N} A_i x_i = a, \tag{1.1b}$$

where $a \in \mathbb{R}^m$, and for $i = 1, \ldots, N$, $A_i \in \mathbb{R}^{m \times n_i}, x_i \in \mathbb{R}^{n_i}$, and the f_i's are closed proper convex functions taking values in the extended real line $(-\infty, \infty]$. We do not assume the f_i to be differentiable.

The problem (1.1) can be visualized as follows : each f_i represents the cost of some activity using m shared resources. How should the shared resources, a, be allocated so as to minimize the sum of the total costs of all the activities ?

We first present here some examples that fit into the model (1.1).

If we now consider instead of ξ a discrete random vector $\tilde{\xi}$ attaining values :

$$\xi^1 = (h_1, T_1) \text{ with probability } \eta_1 > 0,$$

$$\xi^2 = (h_2, T_2) \text{ with probability } \eta_2 > 0,$$

$$\cdots$$

$$\xi^N = (h_N, T_N) \text{ with probability } \eta_N > 0,$$

where

$$\sum_{i=1}^{N} \eta_i = 1,$$

then the problem (2.2) takes the form

$$\min_x \quad \langle c, x \rangle + \sum_{i=1}^{N} \eta_i Q(x, \xi^i) \tag{2.3a}$$

subject to

$$Ax = b, \quad x \geq \mathbf{0}, \tag{2.3b}$$

where, for $i = 1, \ldots, N$

$$Q(x, \xi^i) = \min_y \{ \langle q, y \rangle \mid \quad Wy = h_i - T_i x, \; y \geq \mathbf{0} \}. \tag{2.3c}$$

Putting together the first stage problem (2.3a)-(2.3b) and each realization of the second stage problem (2.3c), we get a large-scale linear programming problem

$$\min_{x,y_1,\ldots,y_N} \langle c,x\rangle + \eta_1\langle q,y_1\rangle + \cdots + \eta_N\langle q,y_N\rangle$$

subject to

$$\begin{aligned}
A\,x &= b\\
T_1\,x + W\,y_1 &= h_1\\
T_2\,x + W\,y_2 &= h_2\\
&\ddots\\
T_N\,x + W\,y_N &= h_N\\
x \geq \mathbf{0}, y_i &\geq \mathbf{0}, i = 1,\ldots,N.
\end{aligned}$$

Its linear programming dual is

$$\max_{v,u_1,\ldots,u_N} \langle b,v\rangle + \langle h_1,u_1\rangle + \cdots + \langle h_N,u_N\rangle$$

subject to

$$\begin{aligned}
u_1\,W &\leq \eta_1 q\\
u_2\,W &\leq \eta_2 q\\
&\ddots\\
u_N\,W &\leq \eta_N q\\
v\,A + u_1\,T_1 + \cdots + u_N\,T_N &\leq c\\
v,\ u_i\,(i = 1,\ldots,N) &\quad \text{unrestricted.}
\end{aligned} \tag{2.4}$$

This dual problem is of the type (1.1) except that minimization has been replaced by maximization.

2.4 Linear Multicommodity Network Problem

Another special instance of a problem with structure (2.1) is the linear multi-commodity network problem. Consider a directed graph $\mathcal{G} = [\mathcal{N}, \mathcal{A}]$ with nodes $\mathcal{N}$ and arcs $\mathcal{A}$. Let E be the ($\mid \mathcal{N} \mid \times \mid \mathcal{A} \mid$) node-arc incidence matrix for the graph $\mathcal{G}$. Here, $\mid \mathcal{N} \mid$ and $\mid \mathcal{A} \mid$ represent the number of nodes and the number of arcs in the network, respectively. Then the linear multi-commodity network problem is

$$\min_{x_0,\ldots,x_N} \quad \sum_{i=1}^{N} \langle c_i, x_i \rangle$$

subject to

$$E\, x_i = b_i, i = 1, \ldots, N \tag{2.5}$$

$$\sum_{i=1}^{N} A_i\, x_i \leq a,$$

where $A_i, i = 1, \ldots, N$, are diagonal matrices [Ken80]. Here, $x_i \in \mathbb{R}^{|\mathcal{A}|}$ are flows in the network for commodities $i = 1, \ldots, N$.

This problem can be generalized to have different node-arc incidence matrices for different commodities, instead of just one.

2.5 A Doubly-coupled Linear Program

A second type of problem can be thought of as an extension of problem (2.1). Suppose that, instead of the divisions having control over all of their internal resources, each of them shares the internal resources *only* with the headquarters. Then the problem has the form

$$\min_{x_0,\ldots,x_N} \quad \langle c_0, x_0\rangle + \langle c_1, x_1\rangle + \cdots + \langle c_N, x_N\rangle$$

subject to

$$\begin{aligned}
D_1\ x_0 + B_1\ x_1 &= b_1 \\
D_2\ x_0 + \qquad B_2\ x_2 &= b_2 \\
&\ddots \\
D_N\ x_0 + \qquad\qquad B_N\ x_N &= b_N \\
A_0\ x_0 + A_1\ x_1 + \quad \cdots \quad + A_N\ x_N &= a \\
x_i \geq 0, i = 0, 1, \ldots, N.&
\end{aligned} \tag{2.6}$$

The above problem does not fit into the form (1.1). Later we shall mention how problem (2.6) can be decomposed so that the subproblems associated with it have the form of problem (1.1).

2.6 Previous Work

Dantzig and Wolfe ([Dan60], [Dan61a]) first suggested a decomposition technique, well known as the Dantzig-Wolfe decomposition, to solve the problem (2.1). It builds upon the notion of column generation, which means the nonbasic columns of the whole matrix needed for computing the reduced cost coefficient are generated only when they are required. The idea of Dantzig-Wolfe decomposition is appealing, but the implementation, application and refinement of this method have been somewhat disappointing in terms of computational efficiency ([Kut73], [Las78], [Dir79]). There have been attempts to improve the efficiency of this method, including the advanced implementation of Ho and Loute ([Ho81], [Ho83], [Ho84]).

Another approach to solve problem (2.1) has been to exploit the special structure of the problem, and use the revised simplex method. There are ways to reduce computational effort and the storage requirement of the inverse of the basis at each pivot. Notable among them are the generalized upper bounding technique by Dantzig and Van Slyke [Dan67], basis factorization [Win74] , and approaches based on LU decomposition by Bartels and Golub [Bar69], Forrest and Tomlin [For72], Saunders [Sau76]. The partitioning method of Rosen [Ros64] is another way to solve the problem (2.1).

For the stochastic linear programming problem (2.2), the first solution method was proposed by Dantzig and Madansky [Dan61b] using the Dantzig-Wolfe decomposition principle applied to the dual problem (2.4). Van Slyke and Wets [Van69]

suggested the L-shaped algorithm which is based on a cutting hyperplane algorithm (outer linearization) and is a partial decomposition method in nature. Another method, known as the compact basis technique or the basis reduction technique, was developed by Strazicky ([Str74], [Str80]) and further treated by Kall [Kal79]. This method takes advantage of the special structure of the basis of the dual problem to obtain a working basis with fewer elements than the standard simplex method would require. Recently, Ruszczyński [Rus86] suggested a method which followed the general principle of the Dantzig-Wolfe principle but used a quadratic regularizing term in the approximate problem (which is close to the idea behind the proximal point algorithm [Roc76]) and restricted the number of columns that needed to be stored. All the above methods for solving problem (2.2) are based on the discretization of the problem. Another approach to solve the problem (2.2) is the stochastic quasi-gradient method [Erm83] based on the stochastic approximation method. The main idea of this method is to make random steps in directions calculated on the basis of some statistical information about the problem gained at each step. For a summary of various methods for solving the stochastic linear programming problem (2.1), see Wets [Wet83].

There are several approaches for solving the linear multicommodity network problem. The primal partitioning method is based on partioning the basis so as to exploit the network structure and is a specialization of the primal simplex algorithm. Another approach is based on the Dantzig-Wolfe decomposition method.

A third approach is based on subgradient optimization [Hel74]. A discussion of the different approaches can be found in Kennington and Helgason [Ken80].

Finally, for problems of the structure (2.6), there are two different methods based on the above ideas : the algorithm of Ritter [Rit67] which is a generalization of Rosen's partitioning method, and the algorithm of Hartman and Lasdon [Har70], which is an extension of the generalized upper bounding technique.

2.7 What is ahead

In the preceeding discussion, we have given different examples of problems that fit into the model (1.1), and also an example which does not directly fit into the model (1.1). Also, we have briefly discussed previous work associated with these examples.

In the next chapter, we will present a decomposition algorithm based on the bundle method for solving the problem (1.1). Our computational experience with this approach for solving instances of the block-angular linear programming problem (2.1) will be presented in Chapter 4. Then, in Chapter 5, we will discuss parallel algorithms for solving the problem (1.1) based on the decomposition method described in Chapter 3, and will present computational experience with block-angular linear programming problems on the CRYSTAL multicomputer [DeW84]. Finally, in Chapter 6, we will give an extension of the decomposition method of Chapter 3 to the doubly-coupled linear programming problem (2.6).

Chapter 3

A DECOMPOSITION ALGORITHM BASED ON THE BUNDLE METHOD

3.0 Introduction

In this chapter, we present a decomposition method based on the bundle method to solve the general problem (1.1). The decomposition technique arising from duality is described in § 3.1. We apply this technique in the hope that the transformed problem will be easier to solve. One major difficulty with the transformed (dual) problem is that it is nonsmooth. In § 3.2 and § 3.3, we present existing methods to address this situation for the problem (1.1) (different approaches for solving various examples described in § 2.1 were mentioned in § 2.2). Then, in § 3.4, we describe the bundle-based approach to solve the problem (1.1). Finally, we present the algorithm in § 3.5.

For expositional convenience, we present the problem (1.1) here again :

$$\inf_{x_1,\ldots,x_N} \quad \sum_{i=1}^{N} f_i(x_i) \tag{1.1a}$$

subject to

$$\sum_{i=1}^{N} A_i\, x_i = a, \tag{1.1b}$$

where $a \in \mathbb{R}^m$, and for $i = 1, \ldots, N$, $A_i \in \mathbb{R}^{m \times n_i}, x_i \in \mathbb{R}^{n_i}$, and, f_i's are closed proper convex functions taking values in the extended real line $(-\infty, \infty]$.

3.1 A Decomposition Method

Here, we present a decomposition technique that has been developed in Robinson [Rob78] and Ha [Ha80]. We present it here somewhat differently than did Robinson and Ha. This decomposition method is based on the Rockafellar dual, and exploits the special structure of the problem.

For the problem (1.1), we define a perturbation function $F(x, p)$ as :

$$F(x,p) = \begin{cases} \sum_{i=1}^{N} f_i(x_i), & \text{if} \quad a - \sum_{i=1}^{N} A_i x_i = p; \\ \infty, & \text{otherwise.} \end{cases} \tag{3.1}$$

Note that $\inf_x F(x, \mathbf{0})$ is the same as the problem (1.1), and that $F(x, p)$ is jointly convex in (x, p). The Lagrangian $L(x, y)$ is defined as

$$L(x,y) := \inf_p \{\langle y, p\rangle + F(x,p)\}$$

$$= \sum_{i=1}^{N} f_i(x_i) + \langle\, y, a - \sum_{i=1}^{N} A_i x_i \,\rangle$$

$$= \langle y, a\rangle + \sum_{i=1}^{N} \{f_i(x_i) - \langle y, A_i x_i\rangle\}.$$

Under the constraint qualification,

$$a \in \sum_{i=1}^{N} A_i(\text{ri dom } f_i^*),$$

the infimum in problem (1.1) is equal to

$$\max_{y} \quad g(y) \tag{3.2}$$

where

$$\begin{aligned} g(y) &:= \inf_{x} L(x,y) \\ &= \langle y, a\rangle - \sum_{i=1}^{N} \sup_{x_i}\{\langle A_i^T y, x_i\rangle - f_i(x_i)\} \\ &= \langle y, a\rangle - \sum_{i=1}^{N} f_i^*(A_i^T y). \end{aligned}$$

The problem (3.2) is known as the dual problem. Notice that the dual maximum value is attained. This dual is a nonsmooth concave maximization problem in m dimensions and usually m is much smaller than $\mathbf{n}$ ($= n_1 + \cdots + n_N$), the number of variables of problem (1.1). For given y, we have

$$f_i^*(A_i^T y) = \sup_{x_i}\{\langle A_i^T y, x_i\rangle - f_i(x_i)\} \tag{3.3}$$

for $i = 1, \ldots, N$, which are N smaller subproblems. We shall call the dual problem (3.2) the *master* problem. A general scheme for solving this unconstrained dual problem can be stated as :

Choose a value of y and for this fixed y solve the N subproblems (3.3).

If y is an optimal solution of (3.2), stop;

Otherwise, update y and solve (3.3) again.

Repeat until an optimal solution is found.

Since the master (dual) problem is a nonsmooth problem, we cannot directly apply any unconstrained methods for solving smooth optimization problems. If we wish to apply a subgradient-type algorithm, we should be able to compute at least one subgradient of $g(\cdot)$ at each y. Rockafellar [Roc70,p.223,p.225] showed that under the condition

$$\bigcap_{i=1}^{N} (\text{ im } A_i^T \cap \text{ri dom } f_i^*) \neq \phi,$$

one has

$$\partial g(y) = a - \sum_{i=1}^{N} A_i \, \partial f_i^*(A_i^T y),$$

where

$$\partial f_i^*(A_i^T y) = \underset{x_i}{\text{argmin}} \{ f_i(x_i) - \langle A_i^T y, x_i \rangle \}.$$

Thus, if $x_i(y)$ is a solution of the i^{th} subproblem

$$\min_{x_i} \{ f_i(x_i) - \langle A_i^T y, x_i \rangle \},$$

then

$$a - \sum_{i=1}^{N} A_i \, (x_i(y)) \in \partial g(y).$$

So, a subgradient of $g(\cdot)$ can be computed at a point y by solving the N subproblems (3.3) at that y.

Now the next question is : can we get a primal optimal solution at the end ? Note that for given y, the problems (3.3) may not have unique solutions. Suppose we know a dual optimal solution $\bar{y}$. Then for this $\bar{y}$, the corresponding solutions $x_i(\bar{y})$ of the subproblems may produce

$$a - \sum_{i=1}^{N} A_i \, x_i(\bar{y}) \neq 0.$$

Then, $x = (x_1(\bar{y}), \ldots, x_N(\bar{y}))$ would not be a feasible solution of the problem (1.1).

3.2 Ha's approach

To overcome the difficulties mentioned in the previous section, Ha [Ha80] in his dissertation applied the proximal point algorithm [Roc76] to the primal problem (1.1). We give a brief description of his approach here.

Let $x^0 = (x_1^0, \ldots, x_N^0)$ be a given starting point, which may not be feasible. Suppose, we choose a bounded sequence of positive numbers $\{\gamma_k\}$. Consider the following modified problem :

$$\inf_{x_1,\ldots,x_N} \quad \sum_{i=1}^{N} \left(f_i(x_i) + \frac{\gamma_{k-1}}{2} \parallel x_i - x_i^{k-1} \parallel^2\right) \qquad (3.4^k)$$

subject to

$$\sum_{i=1}^{N} A_i\, x_i = a.$$

Given x^{k-1}, the problem (3.4^k) produces the solution x^k. Under certain conditions ([Roc76]), the sequence $\{x^k\}$ converges to a solution $\bar{x}$ of the original problem. If the norm of solutions of two successive modified problems is less than some preassigned tolerance, then the procedure is terminated; else, x^{k-1} is replaced by x^k in the modified problem and the procedure is repeated. The modified function

$$\tilde{f}_i(x_i) := f_i(x_i) + \frac{\gamma_{k-1}}{2} \parallel x_i - x_i^{k-1} \parallel^2$$

is strongly convex, and using the following result :

The conjugate of a strongly convex function is a Lipschitz continuously differentiable function,

the modified dual objective, $\tilde{g}$, is now Lipschitz continuously differentiable. The gradient of g, $\nabla\tilde{g}$, is given by

$$\nabla\tilde{g}(y) = a - \sum_{i=1}^{N} A_i\,\tilde{x}_i(y),$$

where $\tilde{x}_i(y)$ are the unique solutions of the modified subproblems

$$\min_{x_i}\{\tilde{f}_i(x_i) - \langle A_i^T y, x_i\rangle\},$$

for $i = 1,\ldots,N$. After making the dual objective differentiable Ha applied the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method [Gil81] to solve the modified dual problem.

The disadvantage of this approach is that, instead of solving one optimization problem, one needs to solve a sequence of them (cf. 3.4^k). This is due to the incorporation of the proximal point algorithm, and it may increase the solution time considerably.

3.3 Subgradient methods

Another approach to solve problem (3.2) would be to use a subgradient method. Typically, a subgradient algorithm can be stated as follows :

At k^{th} iteration, compute a subgradient of g at y^k; call it π^k. Find $y^{k+1} = y^k + \theta_k \pi^k$, where θ_k is a step length determined by some step size rule,

[Pol78]. There are many disadvantages of this method. This is not an ascent algorithm and so there is no line search to obtain a better iterate. There is no implementable termination criterion and the convergence rate is reported to be poor. Another problem is that, even if a dual optimal solution can be found, we want to obtain a primal optimal solution such that $\sum_{i=1}^{N} A_i x_i = a$. Thus we need a method to solve the dual master problem so that we can get a primal optimal solution which satisfies primal feasibility (1.1*b*). For a summary of subgradient methods, see [Pol78], [Zow84].

Marsten *et al* [Mar75] introduced an approach known as the boxstep method. In this method, at each dual iteration, the problem

$$P(y^k;\beta): \qquad \max_{y} \; g(y) \quad \text{subject to} \quad \| y - y^k \|_\infty \leq \beta$$

is solved. Since g is concave, it is clear that if the solution to $P(y^k;\beta)$ is in the interior of the box, then it is the global solution. Otherwise, a new box is considered with the updated dual iterate y^{k+1}. Sometimes, it is easier to solve the problem $P(y^k;\beta)$ than the dual problem as a whole. Again with this method, a primal optimal (feasible) solution may not be obtainable.

3.4 The bundle-based approach

In this section, we present our approach to solve the problem (1.1). Our approach is based on solving the nonsmooth concave dual problem (3.2) by the bundle method ([Wol75], [Lem78a], [Lem81]). Using this bundle-based approach, one can obtain an approximate primal optimal solution. To solve the dual problem, the bundle method uses the concept of an ε-subdifferential of a function. We first state and prove an important result.

Theorem 3.1. *[Zow84] Let g be locally Lipschitzian. Then for every y there exist a neighborhood $\mathcal{N}$ of y, and some $\bar{\varepsilon} > 0$ such that*

$$\bigcup_{v \in \mathcal{N}} \partial g(v) \subset \partial_{\varepsilon} g(y), \qquad \text{for } \varepsilon \geq \bar{\varepsilon}.$$

Proof :

Let y be given. Let B_r be a closed ball of radius r with center at y in int dom g. Let g be Lipschitzian on B_r with constant K. Choose $\mathcal{N}$ small enough so that it is contained in B_r. Then for every $v \in \mathcal{N}$, and $\pi \in \partial g(v)$, we can always find w in B_r such that for some suitable scalar $\alpha > 0$, $\pi = \alpha(v - w)$.

Thus,

$$\begin{aligned} \| \pi \| \cdot \| v - w \| = \langle \pi, v - w \rangle &\leq g(v) - g(w) \qquad (\text{ as } \pi \in \partial g(v)) \\ &\leq K \| v - w \| . \end{aligned}$$

So,

$$\begin{aligned} \bar{\varepsilon} &:= \max_{v \in \mathcal{N}} | g(y) - g(v) | + \max_{\pi \in \partial g(v), v \in \mathcal{N}} \| \pi \| \cdot \max_{v \in \mathcal{N}} \| y - v \| \\ &\leq Kr + Kr = 2Kr. \end{aligned}$$

Now, let $v \in \mathcal{N}$ and $\pi \in \partial g(v)$. Then $\forall\, z$,

$$\begin{aligned}
\langle \pi, z - y \rangle &= \langle \pi, z - v \rangle + \langle \pi, v - y \rangle \\
&\geq g(z) - g(v) + \langle \pi, v - y \rangle \\
&= g(z) - g(y) + \{ g(y) - g(v) + \langle \pi, v - y \rangle \} \\
&= g(z) - g(y) - \{ g(v) - g(y) + \langle \pi, y - v \rangle \} \\
&\geq g(z) - g(y) - \bar{\varepsilon},
\end{aligned}$$

and so,

$$\pi \in \partial_\varepsilon g(y), \qquad \text{for } \varepsilon \geq \bar{\varepsilon}. \quad \blacksquare$$

This implies that the set $\partial_\varepsilon g(y)$ contains the subgradient information from a neighborhood of y. In practice, this set is replaced by some inner approximating polytope.

First, we define here a weight function that we shall be using frequently. For two points $y, u \in \mathbb{R}^m$, define

$$\alpha(y, u, \pi) := g(u) - g(y) + \langle\, \pi(u), y - u \,\rangle.$$

Here, $\pi(u)$ is a subgradient of g at u. Note that $\alpha(y, u, \pi) \geq 0$.

Suppose, for a sequence of iterates $y^j, j = 1, 2, \ldots, k$, the computed subgradients are $\pi^j, j = 1, 2, \ldots, k$, respectively. Here, π^j is of the form

$$\pi^j := a - \sum_{i=1}^{N} A_i x_i^j,$$

where x_i^j is the computed solution of the i^{th} subproblem at the j^{th} iteration. A consequence of the above result is the following.

Corollary 3.2. *If* $\pi^j \in \partial g(y^j)$, *then*

$$\pi^j \in \partial_{\alpha(y^k, y^j, \pi^j)} g(y^k).$$

Proof :

$\pi^j \in \partial g(y^j)$ implies that

$$\begin{aligned} \forall\, y, \qquad g(y) &\leq g(y^j) + \langle \pi^j, y - y^j \rangle \\ &= g(y^k) + \langle \pi^j, y - y^k \rangle - g(y^k) + g(y^j) + \langle \pi^j, y^k - y^j \rangle. \end{aligned}$$

Using the definition of α, we can rewrite

$$\forall\, y, \qquad g(y) \leq g(y^k) + \langle \pi^j, y - y^k \rangle + \alpha(y^k, y^j, \pi^j).$$

This, by the definition of the ε-subdifferential, implies that

$$\pi^j \in \partial_{\alpha(y^k, y^j, \pi^j)} g(y^k). \quad \blacksquare$$

For brevity, we shall write $\alpha_{jk} := \alpha(y^k, y^j, \pi^j)$.

This says that for some suitable ε (i.e., α_{jk}), the subgradients at the previous iterates ($y^1, y^2, \ldots, y^{k-1}$) are α_{jk}-subgradients at the current iterate y^k. Now, consider the convex polyhedron

$$P_k(\varepsilon_k) := \Big\{ \pi = \sum_{j=1}^{k} \lambda_j \pi^j \mid \sum_{j=1}^{k} \lambda_j = 1, \lambda_j \geq 0, \sum_{j=1}^{k} \lambda_j \alpha_{jk} \leq \varepsilon_k \Big\}. \qquad (3.5)$$

Lemma 3.3.

$$P_k(\varepsilon_k) \subseteq \partial_{\varepsilon_k} g(y^k).$$

Proof :

Let $\lambda_1, \ldots, \lambda_k$ be convex weights. By lemma 3.2, we know that

$$\forall\, y, \qquad g(y) \;\leq\; g(y^k) + \langle \pi^j, y - y^k \rangle + \alpha_{jk}, \qquad \text{for } \; j = 1, \ldots, k.$$

Multiplying the above j^{th} inequality by λ_j, and then adding for $j = 1, \ldots, k$, we get

$$\forall\, y, \qquad g(y) \;\leq\; g(y^k) + \langle \sum_{j=1}^{k} \lambda_j\, \pi^j, y - y^k \rangle + \sum_{j=1}^{k} \lambda_j\, \alpha_{jk}$$

which shows that

$$P_k(\varepsilon_k) \subseteq \partial_{\varepsilon_k} g(y^k). \quad \blacksquare$$

In practice, we replace $\partial_{\varepsilon_k} g(y^k)$ by $P_k(\varepsilon_k)$ as given in (3.5).

Now, we can find an ascent direction by finding the least ε_k-subgradient of g at y^k in $P_k(\varepsilon_k)$; this means we need to solve the following quadratic programming problem :

$$\begin{aligned} \min_{\lambda_1, \cdots, \lambda_k} \; & \frac{1}{2} \parallel \sum_{j=1}^{k} \lambda_j \pi^j \parallel_2^2 \\ & \sum_{j=1}^{k} \lambda_j = 1 \\ & \lambda_j \geq 0, \quad j = 1, \ldots, k \\ & \sum_{j=1}^{k} \alpha_{jk} \lambda_j \leq \varepsilon_k. \end{aligned} \tag{3.6}$$

Let the solution of problem (3.6) be $\lambda^k = (\lambda_1^k, \cdots, \lambda_k^k)$. Then, we set

$$d^k := \sum_{j=1}^{k} \lambda_j^k \pi^j \tag{3.7}$$

as the ascent direction. Also, we would like to get an estimate for the increase in the objective function. This estimate, which we call v_k and is used in the line search procedure, is obtained as follows :

If v_k is the dual multiplier vector associated with the convexity constraints of problem (3.6), s_k is associated with the last constraints of problem (3.6), and w_j is associated with λ_j^k, then the optimality conditions for the quadratic programming problem (3.6) are

$$\langle \pi^j, \sum_{\ell=1}^{k} \lambda_\ell^k \pi^\ell \rangle + s_k \alpha_{jk} - v_k = w_j \geq 0, \quad j = 1, \ldots, k \tag{3.8a}$$

$$\sum_{j=1}^{k} \lambda_j^k = 1 \tag{3.8b}$$

$$\lambda_j^k \geq 0, \quad j = 1, \ldots, k \tag{3.8c}$$

$$\sum_{j=1}^{k} \alpha_{jk} \lambda_j^k \leq \varepsilon_k \tag{3.8d}$$

$$s_k \geq 0 \tag{3.8e}$$

$$\langle w, \lambda^k \rangle = 0. \tag{3.8f}$$

$$s_k (\sum_{j=1}^{k} \alpha_{jk} \lambda_j^k - \varepsilon_k) = 0. \tag{3.8g}$$

Multiplying (3.8a) by λ_j^k and adding, and using (3.8b) and (3.8f), we get

$$\| d^k \|^2 + s_k \sum_{j=1}^{k} \lambda_j^k \alpha_{jk} - v_k = 0,$$

which means

$$\begin{aligned} v_k &= \| d^k \|^2 + s_k \sum_{j=1}^{k} \lambda_j^k \alpha_{jk} \\ &= \| d^k \|^2 + s_k \varepsilon_k > 0. \quad (\text{ using } (3.8g)) \end{aligned} \tag{3.9}$$

Once the direction is decided and an estimate v_k is obtained, we need to do a line search along this direction. This line search procedure is due to Lemaréchal [Lem78b]. We briefly describe it here.

Let β, γ, μ be given numbers such that

$$0 < \gamma < \beta < 1, \quad \beta + \mu < 1, \quad \mu > 0.$$

By doing line search, we try to get a dual trial point $u^{k+1} = y^k + t_k d^k$ and its subgradient π^{k+1} so that

$$\langle \pi^{k+1}, d^k \rangle \leq \beta v_k \tag{3.10}$$

and, either

$$g(u^{k+1}) \geq g(y^k) + \gamma\, t_k v_k \tag{3.11}$$

called a *serious step*, or

$$\alpha(y^k, u^{k+1}, \pi^{k+1}) \leq \mu \varepsilon_k \tag{3.12}$$

called a *null step*.

If we have (3.10) and (3.11), then the dual iterate y^k is updated to $y^{k+1} = u^{k+1}$, and the weights α_{jk} to $\alpha_{j,k+1} = \alpha(y^{k+1}, y^j, \pi^{k+1})$, $j = 1, \ldots, k$, with $\alpha_{k+1,k+1} = 0$ and pick a new ε_{k+1}.

On the other hand, if we have (3.10) and (3.12), we stay at y^k (i.e., $y^{k+1} = y^k$), the weights α_{jk} remain the same, and we add π^{k+1} to the bundle. (Note that $\alpha_{k+1,k+1} \leq \mu \varepsilon_k$, so the subgradient at the trial point, π^{k+1}, is an element of $\partial_{\varepsilon_k} g(y^k)$.)

0. *Set* $t_L \leftarrow 0$, $t_R \leftarrow +\infty$, $t_1 \leftarrow 1$
and $k \leftarrow 1$.
1. *At the trial point* $u^{k+1} = y^k + t_k \pi^k$,
compute $g(u^{k+1})$ *and* π^{k+1};
2. *If* { *(3.11) is true* } *then*
if { *(3.10) is true* } *then*
serious step : $y^{k+1} \leftarrow u^{k+1}$,
$k \leftarrow k + 1$
else
$t_L \leftarrow t_k$
if { $t_R = +\infty$ } *then*
$t_k \leftarrow$ *extrapolate* (t_L)
go to 1
else
$t_k \leftarrow$ *interpolate* (t_L, t_R)
go to 1
endif
endif
else
$t_R \leftarrow t_k$
if { $t_L = 0$ } *then*
if { *(3.12) holds* } *then*
if { *(3.10) holds* } *then*
null step : stay at y^k *and enlarge bundle*
else
$t_k \leftarrow$ *interpolate* (t_L, t_R)
go to 1
endif
else
$t_k \leftarrow$ *interpolate* (t_L, t_R)
go to 1
endif
else
$t_k \leftarrow$ *interpolate* (t_L, t_R)
go to 1
endif
endif

Figure 3.1 *Line search method*

The line search method is presented in Figure 3.1. In this method, for extrapolation one may use any formula that increases t_k in such a way that $t_k \to \infty$ if the number of cycles (i.e., the number of tries to go from one serious or null step to the next serious or null step) is unbounded. Similarly, for interpolation one may use any formula so that $(t_R - t_L) \to 0$ if the number of cycles is unbounded. This line search is proved to reach either a null step or a serious step in finite number of iterations if g is weakly semi-smooth [Lem78b].

Finally, we get an approximate optimality condition for the dual from the following result, when both ε_k and $\| d^k \|$ are small.

Theorem 3.4. *For all* $y \in \mathbb{R}^m$,

$$\begin{aligned} g(y) &\leq g(y^k) + \varepsilon_k + \langle d^k, y - y^k \rangle \\ &\leq g(y^k) + \varepsilon_k + \| d^k \| \cdot \| y - y^k \| . \end{aligned}$$

Proof :

From Cor. (3.2), we have, for each y,

$$g(y) \leq g(y^k) + \alpha_{jk} + \langle \pi^j, y - y^k \rangle.$$

Multiplying by the solution $\lambda^k = (\lambda_1^k, \ldots, \lambda_k^k)$ of (3.6), and adding, we get

$$g(y) \leq g(y^k) + \sum_{j=1}^{k} \lambda_j^k \alpha_{jk} + \langle \sum_{j=1}^{k} \lambda_j^k \pi^j, y - y^k \rangle.$$

Using (3.7) and the last constraint of the problem (3.6), we have

$$\begin{aligned} g(y) &\leq g(y^k) + \varepsilon_k + \langle d^k, y - y^k \rangle \\ &\leq g(y^k) + \varepsilon_k + \| d^k \| \cdot \| y - y^k \| . \quad \blacksquare \end{aligned}$$

Thus, the algorithm terminates, when for preassigned $\bar{\varepsilon} > 0$, and $\delta > 0$, we have

$$\varepsilon_k \leq \bar{\varepsilon} \quad \text{and} \quad \| d^k \| \leq \delta,$$

and y^k is an approximate optimal solution for the dual problem.

Now, how do we get a primal optimal solution ? Before we discuss that we need the following two lemmas.

Recall that x_i^j is the computed solution of the i^{th} subproblem at the j^{th} iteration.

Lemma 3.5. *For* $i = 1, \ldots, N$,

$$x_i^j \in \partial f_i^*(A_i^T y^j)$$

implies

$$x_i^j \in \partial_{\alpha_{jki}} f_i^*(A_i^T y^k),$$

where

$$\alpha_{jki} := f_i^*(A_i^T y^k) - f_i^*(A_i^T y^j) - \langle A_i^T y^k - A_i^T y^j, x_i^j \rangle \geq 0.$$

Proof :

It suffices to show the result for a particular i. By definition, $x_i^j \in \partial f_i^*(A_i^T y^j)$ implies that

$$\begin{aligned} \forall\, z_i, \quad f_i^*(z_i) &\geq f_i^*(A_i^T y^j) + \langle z_i - A_i^T y^j, x_i^j \rangle \\ &= f_i^*(A_i^T y^k) + \langle z_i - A_i^T y^k, x_i^j \rangle \\ &\quad - \{ f_i^*(A_i^T y^k) - f_i^*(A_i^T y^j) - \langle A_i^T y^k - A_i^T y^j, x_i^j \rangle \}. \end{aligned}$$

Again, by the definition of the subdifferential,

$$0 \leq f_i^*(A_i^T y^k) - f_i^*(A_i^T y^j) - \langle A_i^T y^k - A_i^T y^j, x_i^j \rangle := \alpha_{jki}.$$

Thus,

$$\forall\, z_i, \qquad f_i^*(z_i) \geq f_i^*(A_i^T y^k) + \langle z_i - A_i^T y^k, x_i^j \rangle - \alpha_{jki},$$

which implies that

$$x_i^j \in \partial_{\alpha_{jki}} f_i^*(A_i^T y^k). \quad \blacksquare$$

Lemma 3.6. *Let α_{jki} be as defined in the previous lemma. Then*

$$\sum_{i=1}^{N} \alpha_{jki} = \alpha_{jk}.$$

Proof :

$$\begin{aligned}
\alpha_{jk} &= \alpha(y^k, y^j, \pi^j) \\
&= g(y^j) - g(y^k) + \langle y^k - y^j, \pi^j \rangle \\
&= \langle y^j, a \rangle - \sum_{i=1}^{N} f_i^*(A_i^T y^j) - \langle y^k, a \rangle + f_i^*(A_i^T y^k) \\
&\qquad + \langle y^k - y^j, a - \sum_{i=1}^{N} A_i x_i^j \rangle \\
&\qquad\qquad (\text{ using the definitions of } g \text{ and } \pi^j) \\
&= \sum_{i=1}^{N} \{ f_i^*(A_i^T y^k) - f_i^*(A_i^T y^j) - \langle A_i^T y^k - A_i^T y^j, x_i^j \rangle \} \\
&= \sum_{i=1}^{N} \alpha_{jki}. \quad \blacksquare
\end{aligned}$$

Now we are ready to prove the main result regarding obtaining a primal optimal solution. We assume that a dual iterate has been obtained which satisfies the termination criterion and we write this approximate dual optimal solution y^k as $\bar{y}$.

Theorem 3.7. *Let α_{jki} and α_{jk} be as defined before. Let $\bar{\lambda} = (\bar{\lambda}_1, \ldots, \bar{\lambda}_k)$ be the solution of the quadratic programming problem (3.6), and*

$$\sum_{j=1}^{k} \bar{\lambda}_j \alpha_{jk} \leq \bar{\varepsilon}.$$

Then $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_N)$, given by

$$\bar{x}_i = \sum_{j=1}^{k} \bar{\lambda}_j x_i^j, \tag{3.13}$$

is an approximate primal optimal solution such that for each $(x_1, \ldots, x_N)$ with $\sum_{i=1}^{N} A_i x_i = a$,

$$\begin{aligned} \sum_{i=1}^{N} f_i(x_i) &\geq \sum_{i=1}^{N} f_i(\bar{x}_i) + \langle \bar{d}, \bar{y} \rangle - \bar{\varepsilon}, \\ &\geq \sum_{i=1}^{N} f_i(\bar{x}_i) - (\| \bar{d} \| \, \| \bar{y} \| + \bar{\varepsilon}), \end{aligned}$$

where $\bar{d} = a - \sum_{i=1}^{N} A_i \bar{x}_i = \sum_{j=1}^{k} \bar{\lambda}_j \pi^j$.

Proof :

By Lemma 3.5, we have

$$x_i^j \in \partial_{\alpha_{jki}} f_i^*(A_i^T \bar{y})$$

which says that for each z_i,

$$f_i^*(z_i) \geq f_i^*(A_i^T \bar{y}) + \langle z_i - A_i^T \bar{y}, x_i^j \rangle - \alpha_{jki}.$$

Multiplying by $\bar{\lambda}_j$, adding over $j = 1, \ldots, k$, and using (3.13), we get for each z_i,

$$f_i^*(z_i) \geq f_i^*(A_i^T \bar{y}) + \langle z_i - A_i^T \bar{y}, \bar{x}_i \rangle - \sum_{j=1}^{k} \bar{\lambda}_j \alpha_{jki}.$$

Writing $\nu_i = \sum_{j=1}^{k} \bar{\lambda}_j \alpha_{jki}$, we can write

$$\bar{x}_i \in \partial_{\nu_i} f_i^*(A_i^T \bar{y}).$$

This can be rewritten as

$$A_i^T \bar{y} \in \partial_{\nu_i} f_i(\bar{x}_i).$$

By definition, this means that for each x_i,

$$f_i(x_i) \geq f_i(\bar{x}_i) + \langle x_i - \bar{x}_i, A_i^T \bar{y} \rangle - \nu_i.$$

Summing over $i = 1, \ldots, N$, we find that for each $x = (x_1, \ldots, x_n)$,

$$\sum_{i=1}^{N} f_i(x_i) \geq \sum_{i=1}^{N} f_i(\bar{x}_i) + \langle \sum_{i=1}^{N} A_i(x_i - \bar{x}_i), \bar{y} \rangle - \bar{\varepsilon},$$

since (using Lemma 3.6),

$$\sum_{i=1}^{N} \nu_i = \sum_{i=1}^{N} \sum_{j=1}^{k} \bar{\lambda}_j \alpha_{jki} = \sum_{j=1}^{k} \bar{\lambda}_j \sum_{i=1}^{N} \alpha_{jki} = \sum_{j=1}^{k} \bar{\lambda}_j \alpha_{jk} \leq \bar{\varepsilon}.$$

Now consider all $x = (x_1, \ldots, x_N)$ such that $\sum_{i=1}^{N} A_i x_i = a$. Then, we have

$$\begin{aligned} \sum_{i=1}^{N} f_i(x_i) &\geq \sum_{i=1}^{N} f_i(\bar{x}_i) + \langle a - a + \bar{d}, \bar{y} \rangle - \bar{\varepsilon} \\ &= \sum_{i=1}^{N} f_i(\bar{x}_i) + \langle \bar{d}, \bar{y} \rangle - \bar{\varepsilon} \\ &\geq \sum_{i=1}^{N} f_i(\bar{x}_i) - (\| \bar{d} \| \, \| \bar{y} \| + \bar{\varepsilon}). \quad \blacksquare \end{aligned}$$

Note that the error bound given by Theorem 3.7 is computable *a posteriori*, since $\bar{d}$, $\bar{y}$, and $\bar{\varepsilon}$ will all be known.

Thus we can obtain an approximate primal optimal solution to the problem (1.1) using the modified bundle method, and it is given by the expression (3.13). This means that along with the dual iterates we need to store the corresponding 'primal' iterates to obtain an approximate primal optimal solution at termination. This modified bundle method also adapts to the situation wherein the bundle becomes too large. In that case, the bundle can be reduced by removing some of its elements and keeping one or more artificial elements. For details, see Lemaréchal *et al* [Lem81]. Thus we need only finite storage for storing dual and 'primal' iterates.

Finally, we get the following result regarding the duality gap.

Corollary 3.8. *Let* $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_N)$ *and* $\bar{y}$ *be a pair of approximate optimal solutions to the primal and the dual, respectively. Let* $\bar{d}$ *be as given in the Theorem 3.7. Then,*

$$\sum_{i=1}^{N} f_i(\bar{x}_i) - g(\bar{y}) \leq \bar{\varepsilon} - \langle \bar{y}, \bar{d} \rangle.$$

Proof :

While proving the Theorem 3.7, we obtained the result

$$A_i^T \bar{y} \in \partial_{\nu_i} f_i(\bar{x}_i).$$

This implies that

$$\nu_i \geq f_i(\bar{x}_i) + f_i^*(A_i^T \bar{y}) - \langle A_i^T \bar{y}, \bar{x}_i \rangle.$$

Summing over, $i = 1, \ldots, N$, we get

$$\sum_{i=1}^{N} \nu_i \geq \sum_{i=1}^{N} f_i(\bar{x}_i) + \sum_{i=1}^{N} f_i^*(A_i^T \bar{y}) - \langle \sum_{i=1}^{N} A_i \bar{x}_i, \bar{y} \rangle.$$

Using the fact that, $\sum_{i=1}^{N} \nu_i \leq \bar{\varepsilon}$ and that, $\bar{d} = a - \sum_{i=1}^{N} A_i \bar{x}_i$, we get

$$\bar{\varepsilon} \geq \sum_{i=1}^{N} f_i(\bar{x}_i) - \left[\langle \bar{y}, a \rangle - \sum_{i=1}^{N} f_i^*(A_i^T \bar{y}) \right] + \langle \bar{y}, \bar{d} \rangle.$$

This implies that

$$\sum_{i=1}^{N} f_i(\bar{x}_i) - g(\bar{y}) \leq \bar{\varepsilon} - \langle \bar{y}, \bar{d} \rangle. \quad \blacksquare$$

Thus, we obtain an *a posteriori* bound on the duality gap.

3.5 The Algorithm

In the context of our problem, we need to mention some nomenclature we will be using here and in the subsequent chapters. By a *dual bundle*, we mean the bundle of dual subgradients computed at dual iterates. A *pseudo-primal point* is the primal "solution" obtained by solving the subproblems $(3.3)_i$ for a given y. So, if, for a given y, x_i's are solutions of the subproblems $(3.3)_i$, for $i = 1, \ldots, N$, then $x = (x_1, \ldots, x_N)$ is a *pseudo-primal point*. Finally, a *primal bundle* is the bundle containing the *pseudo-primal points* obtained from different dual iterate y.

Now we present the method just described formally as the following algorithm.

Algorithm ALG 3.1

Step 0. *Initialization*

Select initialization parameters, $\delta, \bar{\varepsilon} > 0$. Also, an initial $\varepsilon \geq \bar{\varepsilon}$, and a number b for maximum size of the bundle. Choose β, γ and μ such that

$$0 < \gamma < \beta < 1, \quad \beta + \mu < 1, \quad \mu > 0.$$

Pick a starting dual point y^1. Set $k \leftarrow 1, \varepsilon^1 \leftarrow \varepsilon$, and $\alpha_{11} \leftarrow 0$.

Step 1. *Compute the dual function and a subgradient*

For given y^1, solve the N subproblems (3.3). Let x_i be an optimal solution and z_i be the optimal objective function value for the i^{th} subproblem $(3.3)_i$. Compute the dual function and a subgradient

$$g(y^1) = \langle a, y^1 \rangle - \sum_{i=1}^{N} z_i$$

$$\pi^1 = a - \sum_{i=1}^{N} A_i x_i \in \partial g(y^1)$$

and store the *pseudo-primal point* $x = (x_1, \cdots, x_N)$ in the *primal bundle*.

Step 2. *Compute a search direction*

$\varepsilon_k \leftarrow \max\{\bar{\varepsilon}, \min\{\varepsilon_k, \varepsilon\}\}$.

Solve the quadratic programming problem (3.6) to obtain the solution $\lambda_1^k, \cdots, \lambda_k^k$, and v_k (cf. (3.9)).

Let the direction be $d^k = \sum_{j=1}^{k} \lambda_j^k \pi^j$.

Step 3. *Test for convergence*

If $\| d^k \| < \delta$, and $\varepsilon = \sum \lambda_j^k \alpha_{jk} < \bar{\varepsilon}$, then we are done. Compute an approximate primal optimal solution by using the optimal convex weights λ_j^k's with the *pseudo-primal points* in the *primal bundle* (as given in (3.13)), and then compute the primal objective function, and stop.

Else, if $\| d^k \| < \delta$ but $\varepsilon = \sum \lambda_j^k \alpha_{jk} \geq \bar{\varepsilon}$, then $\varepsilon \leftarrow \mu\varepsilon$, and go to step 2.

Else, go to step 4.

Step 4. *Reduction of bundle*

If the bundle has reached its maximum number b, then remove the dual subgradients and the *pseudo-primal points* from the *dual bundle* and the *primal bundle*, respectively, corresponding to $\lambda_j^k = 0$. If the bundle is still too large, keep only the singleton element $\pi^k = d^k$ in the *dual bundle* (and the corresponding aggregated *pseudo-primal point* in the *primal bundle*) and its associated weight $\alpha_{kk} (= \sum_{j=1}^{k} \alpha_{jk} \lambda_j^k)$.

Step 5. *Perform line search*

Compute a point $u^{k+1} = y^k + t^k d^k, \quad t^k > 0$, by the line search method. For given u^{k+1}, compute the dual objective, $g(u^{k+1})$, and a subgradient, π^{k+1} (same as in step 1 with u^{k+1} instead of y^1).

If it is a serious step (cf. (3.10) and (3.11)), update

$$y^{k+1} \leftarrow u^{k+1},$$

$$\alpha_{j,k+1} \leftarrow \alpha_{jk} + g(y^k) - g(y^{k+1}) + \langle \pi^j, y^{k+1} - y^k \rangle,$$

$$\alpha_{k+1,k+1} \leftarrow 0,$$

$$k \leftarrow k+1,$$

store the computed subgradient in the *dual bundle* and the corresponding *pseudo-primal point* in the *primal bundle*, and go to step 2.

If it is a null step (cf. (3.10) and (3.12)), stay at y^k (i.e. $y^{k+1} \leftarrow y^k$ and $\alpha_{j,k+1} \leftarrow \alpha_{jk}, j = 1, \ldots, k$), however, add the computed subgradient π^{k+1} at u^{k+1} to the *dual bundle*, with associated weight

$$\alpha_{k+1,k+1} \leftarrow \alpha(y^{k+1}, u^{k+1}, \pi^{k+1}).$$

Set $k \leftarrow k+1$, and go to step 2.

Remarks

(1) The quadratic programming subproblem in step 2 is solved by an efficient method due to Mifflin [Mif79].

(2) For a detailed description of the line search method, refer to Lemaréchal's paper on line search [Lem78b].

(3) Note that α_{jk} can be computed recursively, so there is no need to store the dual iterates explicitly.

3.6 Summary

In this chapter, we presented a decomposition technique for the problem (1.1) based on duality. We discussed some existing methods to solve this problem. Then we presented our approach based on the bundle method. We showed how we can obtain both approximate primal and dual optimal solutions by applying the modified bundle method. The success of an algorithm of course depends on how it performs when it is implemented. In the next chapter, we describe implementation and present computational experience for a particular example of the problem (1.1), and make comparison with other existing methods.

Chapter 4

BLOCK-ANGULAR LINEAR PROGRAMMING : IMPLEMENTATION AND COMPUTATIONAL EXPERIENCE

In this chapter, we investigate the application of the algorithm described in §3.5 to block-angular linear programming problems. First we review the decomposition method of the previous chapter in the context of block-angular linear programming problem in §4.1. After describing a variant of *ALG 3.1* in §4.2, we describe implementation of the method in §4.3. In §4.4, we report how we generate test data. We were interested in observing the behavior of the algorithm depending on the size of the whole problem, the number of subproblems and the number of coupling constraints. So, we report our computational experience with these factors in the first part of §4.5. Also, we were interested in knowing how this algorithm behaves compared to other methods. So, in the second part on computational experience, we compare this algorithm with MINOS version 5.0, a standard linear programming package [Mur83], and with Mangasarian's sparsity

preserving SOR based method for solving large LPs [Man84], and its proximal point variant ([DeL85], [Del87]) and finally with DECOMP, an implementation of the Dantzig-Wolfe decomposition method.

4.1 Block-angular Linear Programming

The block-angular linear programming problem [Dan60] has the following form :

$$\min_{x_1,\ldots,x_N} \quad \langle c_1, x_1\rangle + \cdots + \langle c_N, x_N\rangle$$

subject to

$$\begin{aligned} B_1\, x_1 \qquad\qquad\qquad\qquad &= b_1 \\ B_2\, x_2 \qquad\qquad\qquad\qquad &= b_2 \\ \ddots \qquad\qquad\qquad\qquad & \\ B_N\, x_N &= b_N \\ A_1\, x_1 + \quad \cdots \quad + A_N\, x_N &= a \\ x_i \geq 0, i = 1,\ldots,N. & \end{aligned} \tag{4.1}$$

Here, $c_i \in \mathbb{R}^{n_i}$, $b_i \in \mathbb{R}^{m_i}$, $B_i \in \mathbb{R}^{m_i \times n_i}$, $A_i \in \mathbb{R}^{m \times n_i}$, $i = 1,\ldots,N$. Assume that the problem is feasible and bounded. As mentioned before in §2.1, if we define

$$f_i(x_i) := \begin{cases} \langle c_i, x_i\rangle, & \text{if} \quad B_i\, x_i = b_i, x_i \geq 0; \\ +\infty, & \text{otherwise,} \end{cases}$$

then, clearly, each $f_i(x_i)$ is a closed proper convex function. Now, the block-angular linear programming problem (4.1) fits into the form of the model (1.1). Here, the subproblems (3.3) become

$$\min_{x_i} \langle c_i - yA_i \, , \, x_i \rangle$$

subject to $(4.2)_i$

$$B_i \, x_i = b_i, \quad x_i \geq 0,$$

for $i = 1, \ldots, N$.

Applying the decomposition technique described in the previous chapter, we get the following dual problem :

$$\max_{y} g(y),$$

where the dual objective, $g(y)$, is given by

$$g(y) := \langle y, a \rangle + \sum_{i=1}^{N} \min_{x_i} \{ \langle c_i - yA_i \, , \, x_i \rangle \mid B_i x_i = b_i, x_i \geq \mathbf{0} \}.$$

As before, if for given y, $x_i(y)$ is a solution of the i^{th} subproblem $(4.2)_i$, then a subgradient of $g(\cdot)$ at y is given by

$$a - \sum_{i=1}^{N} A_i x_i(y) \in \partial g(y).$$

4.2 The Algorithm

An interesting fact to note about the subproblems $(4.2)_i$ is that for a change in y, the feasible regions of the subproblems remain the same; only the objective functions are changing. We capitalize on this fact by using parametric programming at subsequent dual iterates to reduce computational time considerably.

We present a slight variant of the algorithm *ALG 3.1* for block-angular linear programming problems, taking into account parametric programming for solving the linear programming subproblems. It differs from *ALG 3.1* in step 1 and step 5. However, for the sake of completeness, we present here the entire algorithm. Here the parameters and variables are the same as the ones defined in Chapter 3.

Algorithm ALG 4.1

Step 0. *Initialization*

Select initialization parameters, $\delta, \bar{\varepsilon} > 0$. Also, an initial $\varepsilon \geq \bar{\varepsilon}$, and a number b for maximum size of the bundle. Choose β, γ and μ such that

$$0 < \gamma < \beta < 1, \quad \beta + \mu < 1, \quad \mu > 0.$$

Pick a starting dual point y^1. Set $k \leftarrow 1, \varepsilon^1 \leftarrow \varepsilon$, and $\alpha_{11} \leftarrow 0$.

Step 1. *Compute the dual function and a subgradient*

For given y^1, solve the N linear programming subproblems (4.2) from scratch. Let x_i be an optimal solution and z_i be the optimal objective function value for the i^{th} subproblem $(4.2)_i$. Compute the dual function and a subgradient

$$g(y^1) = \langle a, y^1 \rangle + \sum_{i=1}^{N} z_i \tag{4.3a}$$

$$\pi^1 = a - \sum_{i=1}^{N} A_i x_i \in \partial g(y^1) \qquad (4.3b)$$

and store the *pseudo-primal point* $x = (x_1, \cdots, x_N)$ in the *primal bundle*.

Step 2. *Compute a search direction*

$\varepsilon_k \leftarrow \max\{\ \bar{\varepsilon}, \min\{\varepsilon_k, \varepsilon\}\ \}$.

Solve the quadratic programming problem (3.6) to obtain the solution $\lambda_1^k, \ldots, \lambda_k^k$, and v_k (cf. (3.9)).

Let the direction be $d^k = \sum \lambda_j^k \pi^j$.

Step 3. *Test for convergence*

If $\| d^k \| < \delta$, and $\varepsilon = \sum \lambda_j^k \alpha_{jk} < \bar{\varepsilon}$, then we are done. Compute an approximate primal optimal solution by using the optimal convex weights λ_j^k's with the *pseudo-primal points* in the *primal bundle* (as given in (3.13)), and then compute the primal objective function, and stop;

Else, if $\| d^k \| < \delta$, but $\varepsilon = \sum \lambda_j^k \alpha_{jk} \geq \bar{\varepsilon}$, then $\varepsilon \leftarrow \mu\varepsilon$, and go to step 2.

Else, go to next step (step 4).

Step 4. *Reduction of bundle*

If the bundle has reached its maximum number b, then remove the dual subgradients and the *pseudo-primal points* from the *dual bundle* and the *primal bundle*, respectively, corresponding to $\lambda_j^k = 0$. If the bundle is still too large, keep only the singleton element $\pi^k = d^k$ in th edula bundle (and the corresponding aggregated *pseudo-primal point* in the *primal bundle*) and its associated weight $\alpha_{kk} (= \sum_{j=1}^{k} \alpha_{jk} \lambda_j^k)$.

Step 5. *Perform line search*

Compute a point $u^{k+1} = y^k + t^k d^k, \quad t^k > 0$, by a line search method.

For given u^{k+1}, solve the linear programming subproblems (4.2) by parametric programming, starting from the last basis for the new objective functions. Compute $g(u^{k+1})$ and π^{k+1}, using (4.3) with y^1 being replaced by u^{k+1}, and z_i and x_i being the new objective function value and the solution, respectively, of the i^{th} subproblem $(4.2)_i$.

If it is a serious step (cf. (3.10) and (3.11)), update

$$y^{k+1} \leftarrow u^{k+1},$$

$$\alpha_{j,k+1} \leftarrow \alpha_{jk} + g(y^k) - g(y^{k+1}) + \langle \pi^j, y^{k+1} - y^k \rangle,$$

$$\alpha_{k+1,k+1} \leftarrow 0,$$

$$k \leftarrow k + 1,$$

store the computed subgradient in the dual bundle and the corresponding *pseudo-primal point* in the primal bundle. Go to step 2.

If it is a null step (cf. (3.10) and (3.12)), stay at y^k (i.e. $y^{k+1} \leftarrow y^k$ and $\alpha_{j,k+1} \leftarrow \alpha_{jk}$, $j = 1, \ldots, k$), however, add the computed subgradient π^{k+1} at u^{k+1} to the *dual bundle*, with associated weight

$$\alpha_{k+1,k+1} \leftarrow \alpha(y^{k+1}, u^{k+1}, \pi^{k+1}).$$

Set $k \leftarrow k + 1$, and go to step 2.

4.3 Implementation of the algorithm

We have implemented the algorithm *ALG 4.1* in Fortran 77. We shall call our code BUNDECOMP. It is written using two available routines, one for solving the dual master problem by the bundle method, and the other for solving a linear programming problem by the revised simplex method.

The code for the bundle method, M1FC1, is made available to us by C. Lemaréchal [Lem85]. M1FC1 is written for solving a convex minimization problem. The user needs to provide two subroutines which are passed as arguments to M1FC1 and so needs to be declared as *external* in the routine which calls M1FC1. One of the subroutines is SIMUL, which computes the function value and a subgradient of the dual objective function at a given point y. The other routine, PROSCA, is a subroutine which computed the inner product of two vectors. Besides these, the user needs to provide the following main parameters : `N`, `DX`, `DF1`, `EPS`, `ITER`, `NSIM`, `MEMAX`. We describe below what they stand for.

`N` – the dimension of the problem, In our case, it is the dimension of the dual problem or equivalently, the number of coupling constraints (1.1b),

`DX` – the required accuracy on the dual variable y,

`DF1` – a positive number which is used for expected change in the objective function at the first iteration. It is also used for initializing the stepsize and the ϵ of the theory,

`EPS` – the required accuracy on the objective function,

`ITER` – the maximum number of dual iterations allowed,

NSIM – the cumulative maximum number of calls to the subroutine SIMUL.

MEMAX – the maximum number of subgradients allowed to be stored in the bundle at any time.

In the original version of the code, M1FC1, the calculations were performed both in single and in double precision. The main working space required for M1FC1 is (1) $2 \times \texttt{MEMAX} + 2$ for integers, (2) $5 \times \texttt{N} + (\texttt{N} + 4) \times \texttt{MEMAX}$ for single precision numbers, and (3) $(\texttt{MEMAX} + 9) \times \texttt{MEMAX} + 8$ for double precision numbers. In our modified version of M1FC1, we have changed all the computation to double precision along with changing all the variables to double precision variables. Thus, for this modified version, the array for (2), mentioned above, is also changed to double precision.

The other available routine we used is ZX0LP from the IMSL library [Imsl84]. The routine, ZX0LP, solves a linear programming problem by the revised simplex method. We use this routine to solve the linear programming subproblems $(4.2)_i$. It can handle both equality and inequality constraints. But, we restrict it to equality constraints as we generate only equality constrained problems. In this routine, the coefficient matrix of a linear programming problem is stored in its entire matrix form. By this, we mean, it does not use any special data structure to store only the non-zero data of the coefficient matrix. Also, the inverse of the basis is stored in its entire form. Thus, as far as we are concerned, this routine is indifferent to the structure of the subproblems. Thus, almost in all the test

problems we generated for testing algorithm *ALG 4.1*, the matrix, B_i, generated for the subproblem is fully dense.

We decided to use the routine ZX0LP for its availability, reliability and flexibility. We have tested linear programming problems of size up to 100 by 160 on this routine, and it seems to be reliable. Besides that, we especially needed a routine which can solve a parametric programming subproblem. In our case, this is crucial. Recall that, for changes in y, the feasible region of each subproblem $(4.2)_i$, by itself, remains the same; only, the objective function changes. So, only in the very first computation of the function value and a subgradient for the dual objective function, do we need to solve the linear programming subproblems $(4.2)_i$ from infeasibility by doing both phase I and phase II of the simplex method. Afterward, we solve each subproblem for a new objective function by doing phase II starting from the last basis. This can be done by parametric programming. Thus, we needed a code, which can solve a parametric linear programming problem given the information about the solution for the last objective function. If we can use this, it will reduce the computational time considerably, as on average, for doing the parametric programming part only a few pivots are needed to reach at a new solution from the last vertex, compared to solving a linear programming problem by doing both the phase I and phase II of the simplex method. From our computational experience, we have observed that, using parametric programming, the computation of the dual objective and its (sub)gradient is reduced by a factor of about seven to ten. This is a significant amount when one has to compute the

function and the subgradient sometimes about a hundred times (though it varies depending on number of dual variables). For example, suppose in computing the function and the gradient from scratch, it takes, on average, about 50 seconds. In subsequent steps, suppose that it takes, on average, about 7 seconds when we use previous information and do parametric programming. Further, suppose, for a particular problem, it needs, say, 60 function and gradient evaluations. If we solve from scratch every time to compute the function and the gradient, the total time will add up to $50 \times 60 = 3000$ seconds, whereas if we compute from scratch the first time and then employ parametric programming at the subsequent steps, the total time needed will be $50 + 59 \times 7 = 463$ seconds. This represents a considerable saving. We found the routine ZX0LP flexible enough to implement parametric programming and save computational time.

In our package, BUNDECOMP, we have made some modifications to the original code, M1FC1, to suit our situations, thanks to the suggestions made by Lemaréchal [Lem86b]. Recall that for our problem we want to obtain an approximate primal optimal solution to the original problem at the end. For this, we need to store the *pseudo-primal points* in a *primal bundle*, corresponding to the dual subgradients stored in the *dual bundle*. Thus, whenever, a subgradient is stored in the *dual bundle* at an iterate, we store the corresponding *pseudo-primal point*. Also, whenever we remove members of the subgradients from the *dual bundle* as we have reached the maximum size of the bundle `MEMAX`, we remove the corresponding *pseudo-primal point* (cf. step 4 of the algorithm *ALG 4.1*). Finally, when we are

done, we compute an approximate primal optimal solution by using (3.13). We shall call the modified code M1FC1M.

As a part of BUNDECOMP, we also wrote the routine, SIMUL, which computes and returns the value of the dual objective function and a subgradient of this objective function for the master dual problem. SIMUL calls the IMSL routine, ZX0LP, to solve the linear programming subproblems. The data structure we used to store the non-zero elements is easy to implement, as we generate fully dense matrices, B_i, A_i, $i = 1, 2, \ldots, N$. All the elements for the matrices, B_1, B_2, $\ldots$, B_N, are stored in a long array, vec, in which the first $m_1 \times n_1$ locations contain the matrix, B_1, the next $m_2 \times n_2$ locations contain the matrix, B_2, and so on. In Fortran, a matrix is stored by column-major. So, here, in the first $m_1 \times n_1$ entries of vec, the elements, $vec(i)$, $i = 1, \ldots, m_1$, correspond to the elements, $B_1(i, 1)$, $i = 1, \ldots, m_1$, of the matrix, B_1, respectively; the elements, $vec(m_1 + i)$, $i = 1$, $\ldots$, m_1, correspond to the elements, $B_1(i, 2)$, $i = 1, \ldots, m_1$, respectively; and so on. The same situation is repeated for each matrix, B_2, B_3, $\ldots$, B_N, starting in locations $vec(m_1 \times n_1 + 1)$, $vec(\sum_{i=1}^{2} m_i \times n_i + 1)$, $\ldots$, $vec(\sum_{i=1}^{N-1} m_i \times n_i + 1)$, respectively. We use a similar data structure to store the elements of the matrices, A_1, A_2, $\ldots$, A_N, of the coupling constraints using another array.

Recall that M1FC1 is a code for solving a convex minimization problem. In our situation, we are maximizing a concave function. We made the changes in computing the function and a (sub)gradient by using the fact : if g is a concave function, then $-g$ is a convex function and similarly, for its gradients. Also, we

used the single precision version of the routine ZX0LP for most of our calculation as we did not have access to the double precision version in the machine on which we did most of our testing. However, in the procedure SIMUL of BUNDECOMP, the dual objective and the computed subgradient are returned in double precision by computing the inner product and the matrix multiplication in double precision (cf. (4.3)).

The program can terminate in any one of several ways. The most desirable one is where the termination criterion of the theory is met for preassigned values of `EPS`, `DX`, and `DF1`; we shall call this *Normal End*. The other termination criteria are : a) when the maximum number of iterations is reached (*MaxIter*), b) when the maximum number of calls to SIMUL (to compute dual function and subgradient) is reached (*MaxSim*), c) when the accuracy `DX` (i.e., input on required accuracy on successive y's) is reached without making any more improvement on g (*DxEnd*). We added another termination criterion after observing outputs from sample runs. We noticed that, on average, it takes two to three calls to SIMUL to go from one dual iteration to the next one. Very rarely does it require more than nine or ten. Sometimes when the program tries to compute a new dual iterate from the last iterate, it keeps calling SIMUL in the line search procedure again and again. This happens only when the procedure has almost converged. We think that this occurs due to round off errors. So, instead of letting the computation continue, we have put a limit of maximum number of 15 calls to SIMUL in the

line search procedure. We call this termination criterion, *Max15*. In any event, we compute the absolute and relative accuracy of the final dual gradient.

Our implementation is system dependent in the sense that it uses the IMSL routine ZX0LP. So, our code can run on any system that has the IMSL library. We have compiled our program on a DEC VAX 11/780 and on a DEC VAXstation II workstation, both running under the Berkeley UNIX (4.3 bsd) operating system, using the f77 compiler invoking -O option (optimizer). The version on these two machines uses the single precision ZX0LP. We have also compiled our code on a DEC VAX 11/780 running the VMS operating system using the VMS Fortran compiler 'FOR'. This one uses the double precision ZX0LP.

Our implementation is mainly designed to study the performance of the algorithm. We are mostly interested in observing the CPU time for various factors, depending on the size of the whole problem, the number of subproblems and the number of coupling constraints, and in observing its performance compared to existing methods. So, our code is still in the experimental stage, and it does not have all the sophistication of a commercial code (like storage conservation, numerical stability and accuracy etc.).

In the next section, we describe how we generated the test data.

4.4 Generation of test data

The data for the test problems are generated using a uniform [0,1] pseudo-random number generator. Then, different sections of the input data are mapped to the desired range. Initially, we decide on the different sizes of the subproblems. Next, we generate a feasible solution for the subproblems in the interval [0.0, 4.0]. Then, the data for the coefficient matrices, B_i,of the subproblems are generated in the interval [-8.0, 9.0], except for the last row. The last row is generated in the interval [5.0, 13.0]. We did this to keep the subproblems bounded. We multiply B_i by the generated feasible solution to obtain the right hand side, b_i. Next, we generate the objective coefficients, c_i, in the interval [-7.0, 7.0]. Finally, we generate the matrices in the coupling constraints, A_1, A_2, ..., A_N, in the interval [-8.0, 8.0]. The multiplication of A_i with the generated feasible solution is cumulated to obtain the right hand side of the coupling constraints, a. Please note that we do not have *a priori* knowledge of the solution of the problem.

4.5 Computational experience

We undertook two sets of experiments. In the first, we observed the behavior of the algorithm *ALG 4.1* when different factors are varied; in the second set, we compared BUNDECOMP with other existing algorithms to solve large-scale sparse linear programming problems : MINOS [Mur83], LPSOR2 [Man84], PROSOR [DeL85], [Del87] and DECOMP [Ho].

4.5.1 Experiment I

We had several questions in mind. How does the algorithm behave when you increase the number of coupling constraints for fixed number of subproblem ? How does the time grow as the number of subproblems varied when the number of coupling constraints is kept fixed ? How is the density related to time ? What is the convergence behavior of the bundle method for our type of problem ?

To help answer these questions, we selected five different sizes of the whole problem. They are 350 $\times$ 500, 850 $\times$ 1500, 1250 $\times$ 2800, 1500 $\times$ 4000 and 2050 $\times$ 5000. Though these problems are not extremely large, we worked with them mainly as we wanted to find out whether our method was workable, by LP standards, and how it behaved. Due to limitation of machine memory, we restricted most of our computational testing to the first three sizes. However, we did testing for certain cases for the last two sizes, which we shall report in our comparison with the other methods. We also ran a problem of size 4000 $\times$ 10000. Once we decided the sizes of the problems, we restricted our attention to two factors : number of coupling constraints and number of subproblems. First, we decided the number of coupling constraints. The ones we have chosen are 10, 20, 30, 40 and 50. Then we decided the sizes of the subproblems by trying various number of subproblems. We attempted four cases for number of subproblems : 10, 25, 40 and 100. Once we had the number of coupling constraints and the number of subproblems, we decided the size(s) of the subproblems by 'equally' dividing the size of the whole problem minus the coupling part, by the number

of subproblems. To illustrate this, let us consider a problem of size 350 $\times$ 500. Suppose we decided that the number of coupling constraints is to be 20 and the number of subproblems is to be 40. Now, 350 $\times$ 500 minus the coupling part gives us the reduced size 330 $\times$ 500. If we divide 330 $\times$ 500 by 40, we get 8.25 $\times$ 12.5. Since, the subproblems cannot be of non-whole number of rows and/or columns, we rounded off to the nearest two whole numbers. Thus, we can get the following four sizes : 9 $\times$ 13, 8 $\times$ 13, 9 $\times$ 12, 8 $\times$ 12. Then fractional parts 0.25 and 0.5 (from 8.25 and 12.5, respectively) for 40 subproblems tell us that $0.25 \cdot 40 = 10$ subproblems can have 9 rows each (and the rest 8 each), and $0.5 \cdot 40 = 20$ subproblems can have 13 columns each (and the rest 12 each). We allocated the bigger numbers to the first few problems. Thus, here, the first 10 subproblems have 9 rows each, and the last 30 have 8 rows each, and the first 20 subproblems have 13 rows each and the rest have 12 rows each. So, we now have that first 10 subproblems are of size 9 $\times$ 13, the next 10 of size 8 $\times$ 13, and the last 20 of size 8 $\times$ 12. We used this scheme so as to decide easily about the sizes of the subproblems. An important point to remember is that since the subproblems are of almost equal size, we can estimate the density of the coefficient matrix for the *non-coupling* part just by the number of subproblems. Suppose m_i, n_i denote the size of subproblems (rows and columns), and N denotes the number of subproblems. Then the number of non-zero elements in the non-coupling parts is $N \cdot m_i \cdot n_i$ and the total number of elements is $(N \cdot m_i) \cdot (N \cdot n_i)$. Thus the fractional density is $(N \cdot m_i \cdot n_i)/(N \cdot m_i \cdot N \cdot n_i) = \frac{1}{N}$. So, the density (for

non-coupling part) depends (almost) solely on the number of subproblems. Thus, 10 subproblems means 10 % dense, 25 means 4 % dense, 40 means 2.5 % dense and 100 means 1 % dense for the non-coupling part.

In Table 4.1, we give the parameter specifications of the problems that we considered for testing. In this table, the sizes of the subproblem is a rough estimate. For instance, with the example mentioned above (size 350 $\times$ 500, coupling constraints—20, number of subproblems—40), we write 9 $\times$ 13, under the column for 'size of subproblems'.

We designate the different test problems by names consisting of three characters. The first number stands for the different sizes of the whole problem : 1 for 350 $\times$ 500, 2 for 850 $\times$ 1500, 3 for 1250 $\times$ 2800, and so on. The second letter (letter in the middle) stands for number of coupling constraints : a for 10, b for 20, c for 30, d for 40 and e for 50. The third and final number stands for number of subproblems : 1 for 100, 2 for 40, 3 for 25 and 4 for 10. Thus, the problem name 3b2 stands for a problem of size 1250 $\times$ 2800 with 20 coupling constraints and 40 subproblems.

So far, we have discussed what problems sizes we consider, how we decide the sizes of subproblems and so on. We already discussed in §4.4 how we generate data once we have subproblem sizes. Next we discuss the values we tried for the three main input parameters for M1FC1M, the code for solving the dual master problem. They are `EPS`, `DX`, and `DF1`. For most of the test problems, we did three runs where we kept `DX` and `DF1` fixed, and tried different values of `EPS`. We did

try different values of DX and DF1. However, in a particular set of three runs for a problem, we varied only EPS. The values of DX we tried are 10^{-6} or 10^{-7}. When we first started testing, we used small values for DF1, say, 1.0, 2.0 or 4.0. Then, due to suggestions by C. Lemaréchal [Lem86c] that bigger values of DF1 might help in convergence, we started trying bigger values of DF1 (mostly, 1000 or 10000). His suggestion turned out to be helpful as this increased number of times we achieved *Normal End*. The values of EPS we tried are mostly 0.5, 1.0, 2.0, 5.0 for smaller number of coupling constraints (10, 20), and 1.0, 5.0, 10.0, 20.0, 50.0 for larger ones (30, 40, 50).

In Table 4.2, we present the solution time in CPU minutes with other pertinent information for some of the problems of Table 4.1. Output information for most of the rest of the problems can be found in Appendix A. We report the number of dual iterations and the number of dual objective function (and subgradient) evaluations to reach the final solution, the final primal and dual objective value, the absolute and relative accuracy of the final dual subgradient, and the termination criterion. The table shows three different times : the total CPU time in minutes for solving the whole problem, the time for solving all the subproblems for the first time (from scratch), i.e, first call to SIMUL, and the average of the times for calls to SIMUL at subsequent steps. This shows how much faster it is to use parametric programming at subsequent steps. The absolute gradient accuracy is the inf-norm, i.e., $\| \pi \|_\infty = \max_{1 \le i \le m} \{|\pi_i|\}$, where m is the dimension of the vector π. The relative accuracy is the measure $ra = \max_{1 \le i \le m} \{|\frac{\pi_i}{a_i}|\}$, where a is

Problem Name	Size of whole problem	Number of coupling constraints	Number of sub-problems	Size of sub-problems	Density %
1a1	350 × 500	10	100	3 × 5	3.829
1a2	350 × 500	10	40	9 × 13	5.291
1a3	350 × 500	10	25	14 × 20	6.743
1a4	350 × 500	10	10	34 × 50	12.571
1b1	350 × 500	20	100	3 × 5	6.657
1b2	350 × 500	20	40	8 × 13	8.074
1b3	350 × 500	20	25	14 × 20	9.486
1b4	350 × 500	20	10	33 × 50	15.143
1c1	350 × 500	30	100	3 × 5	9.488
1c2	350 × 500	30	40	8 × 13	10.857
1d1	350 × 500	40	100	3 × 5	12.314
1d2	350 × 500	40	40	8 × 13	13.646
1e1	350 × 500	50	100	3 × 5	15.143
1e2	350 × 500	50	40	8 × 13	16.434
1e3	350 × 500	50	25	12 × 20	17.714
1e4	350 × 500	50	10	30 × 50	22.857
2a1	850 × 1500	10	100	8 × 15	2.165
2a2	850 × 1500	10	40	21 × 38	3.647
2a3	850 × 1500	10	25	34 × 60	5.129
2a4	850 × 1500	10	10	84 × 150	11.059
2b1	850 × 1500	20	100	8 × 15	3.329
2b2	850 × 1500	20	40	21 × 38	4.794
2b3	850 × 1500	20	25	34 × 60	6.259
2b4	850 × 1500	20	10	83 × 150	12.118
2c1	850 × 1500	30	100	8 × 15	4.494
2c2	850 × 1500	30	40	21 × 38	5.942

Table 4.1 *Problem Specifications*

(Continued on the next page)

Problem Name	Size of whole problem	Number of coupling constraints	Number of sub-problems	Size of sub-problems	Density %
2d1	850 × 1500	40	100	8 × 15	5.659
2d2	850 × 1500	40	40	21 × 38	7.089
2e1	850 × 1500	50	100	8 × 15	6.824
2e2	850 × 1500	50	40	20 × 38	8.235
2e3	850 × 1500	50	25	32 × 60	9.647
2e4	850 × 1500	50	10	80 × 150	15.294
3a1	1250 × 2800	10	100	13 × 28	1.792
3a2	1250 × 2800	10	40	31 × 70	3.280
3a3	1250 × 2800	10	25	50 × 112	4.768
3b1	1250 × 2800	20	100	13 × 28	2.584
3b2	1250 × 2800	20	40	31 × 70	4.060
3b3	1250 × 2800	20	25	50 × 112	5.536
3c1	1250 × 2800	30	100	13 × 28	3.376
3c2	1250 × 2800	30	40	31 × 70	4.840
3d1	1250 × 2800	40	100	12 × 28	4.168
3d2	1250 × 2800	40	40	30 × 70	5.620
3e1	1250 × 2800	50	100	12 × 28	4.960
3e2	1250 × 2800	50	40	30 × 70	6.400
4a1	1500 × 4000	10	100	15 × 40	1.660
4b1	1500 × 4000	20	100	15 × 40	2.320
5a1	2050 × 5000	10	100	20 × 50	1.483
5b1	2050 × 5000	20	100	20 × 50	1.966
6a1	4000 × 10000	10	100	40 × 100	1.247

Table 4.1 *Problem Specifications*

the right hand side of the coupling constraints. The starting dual point is arbitrarily chosen. For most of the problems, we report output information from three runs.

Graphs and Interpretation

We first plotted CPU time against number of coupling constraints when the number of subproblems was kept fixed. Figure 4.1 corresponds to the number of subproblems being fixed at 100, while Figure 4.2 corresponds to the number of subproblems being fixed at 40. From the graphs, we can see that the time is increasing linearly as number of coupling constraints increases, with slope less than one. So, we can conclude that for increase in number of coupling constraints, computing time would probably increase linearly with slope less than one.

Figure 4.3 and 4.4 correspond to time plotted against density (%) when the number of coupling constraints are kept fixed at 10 and 20, respectively. Here density means the density of the *non-coupling* parts. It appears from the graphs that for smaller problem (350 $\times$ 500), time is increasing linearly with density, while for bigger problems (850 $\times$ 1500, 1250 $\times$ 2800), time is increasing quadratically. To see why this difference is appearing consider Table 4.2 and Table A1 (of Appendix A). It is clear that once the dual function and subgradient are computed (calling SIMUL), the rest of the work in the modified bundle algorithm takes about one percent of the total computing time. This means most of the time is spent in computing the dual function and a subgradient. Inside SIMUL most of the time is spent in solving the subproblems by the simplex method. For a

particular subproblem, we denote m for number of constraints and n for number of variables. From our computational experience, we have observed that to solve a subproblem from scratch, it takes *about*

$$c\,m^2 n$$

units of time, where c is a constant (refer to §5.4.2). Again from Table 4.2, we can see that to compute the function and a subgradient in subsequent steps it takes, on average, 1/10 of the time of the first call to SIMUL. In our experiments, all the subproblems are of almost equal size. If N denotes number of subproblems, and K is the average number of calls to SIMUL to compute the function and a subgradient to solve the whole problem (after the very first call), then total time to compute the whole problem will be :

$$\begin{aligned} time &= cNm^2n + K\frac{c}{10}Nm^2n + e \\ &= c(1 + \frac{K}{10})Nm^2n + e, \end{aligned}$$

where e is the error amount.

The above expression gives us a rough estimate of the time required to solve the whole problem. Suppose we fix the size of the whole problem, say, at 850 $\times$ 1500. Now, if we have 100 subproblems, the size of the subproblems is then 8×15. On the other hand if we have 25 subproblems, the size of the subproblems is 34×60. In any case, it is clear that, for the fixed size of the whole problem,

$N \cdot n$ is fixed (e.g. $1500 = 100 \cdot 15 = 25 \cdot 60 = 10 \cdot 150$). Hence, if in the above expression we replace $N \cdot n$ by a constant (α), then time can be expressed as :

$$time = \alpha c(1 + \frac{K}{10})m^2 + e.$$

Thus, we can infer that, for fixed size of the whole problem, time increases quadratically as density increases (note : density is proportional to m). Thus, we think that though the plots for 350×500 problem appear to be linear in Figures 4.3 and 4.4, it may be so because (i) the coefficient with the quadratic term is very small compared to that of 850×1500 and 1250×2800, and (ii) the scale in the graphs. So, to understand it better, we plotted 350×500, for both coupling constraints 10 and 20 in Figure 4.5 with different scale. From this graph, we can see that time is increasing almost quadratically (at least not linearly) as density increases for this size problem too.

Finally, we observed the convergence behavior of the modified bundle method. We plotted $log(error)$ against the number of iterations for some of the test problems, where error stands for the difference in the value of the dual objective at a specific dual iterate and at the dual optimal solution (refer to Figures 4.6, 4.7, 4.8, 4.9 and 4.10). From the graphs, we can conclude that, though the bundle method is a finitely convergent method, it behaves like a *linearly convergent* method for the type of problems we tested.

Problem Name	Size of whole problem	EPS/ DF1	Objective value : primal/dual	ITER/ EVAL	Gradient Accuracy/ Relative	Time in Min.	*Stopping Rule*
1a1	350 × 500	0.05	-1066.0138	35	0.519×10^{-14}	1.37	*Normal*
		1.5	-1066.0502	92	0.103×10^{-15}	0.07	*End*
						0.01	
1a1	350 × 500	0.5	-1065.9044	29	0.152×10^{-14}	1.03	*Normal*
		1.5	-1066.3411	67	0.121×10^{-15}	0.07	*End*
						0.01	
1a1	350 × 500	1.0	-1065.5763	24	0.569×10^{-14}	0.94	*Normal*
		1.5	-1066.5763	59	0.291×10^{-15}	0.07	*End*
						0.01	
1a2	350 × 500	1.0	-1260.5870	29	0.104×10^{-13}	1.81	*Normal*
		100	-1261.5869	73	0.113×10^{-15}	0.19	*End*
						0.02	
1a2	350 × 500	0.1	-1260.3163	28	0.397×10^{2}	1.81	*Max15*
		100	-1261.3693	78	0.649	0.19	
						0.02	
1c1	350 × 500	1.0	-1083.2228	102	0.217×10^{1}	6.14	*Max15*
		1000	-1084.1033	221	0.799×10^{-1}	0.06	
						0.02	
1c1	350 × 500	5.0	-1082.9426	122	0.296	6.57	*Max15*
		1000	-1086.8215	221	0.359×10^{-1}	0.06	
						0.02	
1c1	350 × 500	20.0	-1073.8361	142	0.758×10^{-1}	6.82	*Max15*
		1000	-1092.4553	237	0.875×10^{-2}	0.06	
						0.02	

Table 4.2 *Output information from BUNDECOMP*
* – *DX is set at* 10^{-6}*; rest are set at* 10^{-7}.
(Continued on the next page)

Problem Name	Size of whole problem	EPS/ DF1	Objective value : primal/dual	ITER/ EVAL	Gradient Accuracy/ Relative	Time in Min.	*Stopping Rule*
2a1	850 × 1500	0.5	-6362.6378	30	0.212×10^{1}	6.82	*Max15*
		1000	-6363.1406	91	0.101	0.70	
						0.07	
2a1	850 × 1500	1.0	-6362.1887	23	0.266×10^{-14}	5.14	*Normal*
		1000	-6363.1885	59	0.127×10^{-15}	0.70	*End*
						0.07	
2a1	850 × 1500	1.0	-6358.4196	17	0.918×10^{-14}	4.16	*Normal*
		1000	-6363.4194	41	0.601×10^{-15}	0.70	*End*
						0.08	
2a2	850 × 1500*	1.0	-7067.6398	20	0.151×10^{-13}	18.75	*Normal*
		1000	-7068.6396	51	0.557×10^{-16}	3.23	*End*
						0.31	
2a2	850 × 1500*	5.0	-7064.2126	20	0.133×10^{-13}	18.18	*Normal*
		1000	-7069.2124	46	0.515×10^{-16}	3.23	*End*
						0.33	
2a2	850 × 1500*	5.0	-7061.2687	26	0.124×10^{-13}	19.80	*Normal*
		1000	-7070.9414	59	0.666×10^{-16}	3.24	*End*
						0.29	
2c1	850 × 1500	1.0	-6453.8673	60	0.324×10^{1}	13.85	*Max15*
		1000	-6454.3823	147	0.552	0.60	
						0.09	
2c1	850 × 1500	5.0	-6451.0178	86	0.266×10^{-2}	13.88	*Normal*
		1000	-6456.0029	143	0.920×10^{-3}	0.60	*End*
						0.09	
2c1	850 × 1500	20.0	-6439.6217	83	0.451×10^{-2}	14.05	*Normal*
		1000	-6458.4658	143	0.145×10^{-2}	0.60	*End*
						0.09	

Table 4.2 *Output information from BUNDECOMP*
* – *DX is set at* 10^{-6}*; rest are set at* 10^{-7}.
(Continued on the next page)

Problem Name	Size of whole problem	EPS/ DF1	Objective value : primal/dual	ITER/ EVAL	Gradient Accuracy/ Relative	Time in Min.	*Stopping Rule*
3a1	1250 × 2800	1.0	-17194.2746	14	0.239×10^{-13}	11.55	*Normal*
		1000	-17195.2734	42	0.177×10^{-15}	2.05	*End*
						0.23	
3a1	1250 × 2800	5.0	-17191.4618	17	0.440×10^{-14}	11.15	*Normal*
		1000	-17196.2770	38	0.390×10^{-16}	2.06	*End*
						0.24	
3a1	1250 × 2800	10.0	-17186.5065	17	0.130×10^{-13}	10.96	*Normal*
		1000	-17196.5059	36	0.292×10^{-15}	2.05	*End*
						0.25	
3a2	1250 × 2800	1.0	-16687.6813	22	0.379×10^{1}	70.05	*Max15*
		1000	-16688.6406	63	0.206×10^{-1}	13.13	
						0.92	
3a2	1250 × 2800	5.0	-16684.2314	19	0.156×10^{-13}	65.79	*Normal*
		1000	-16689.2305	40	0.224×10^{-15}	13.10	*End*
						1.34	
3a2	1250 × 2800	10.0	-16679.3584	20	0.469×10^{-13}	66.92	*Normal*
		1000	-16689.3574	46	0.490×10^{-15}	13.09	*End*
						1.19	
3c1	1250 × 2800	5.0	-16926.6433	42	0.437×10^{1}	26.47	*Max15*
		10^5	-16931.6289	105	0.675×10^{-1}	2.12	
						0.23	
3c1	1250 × 2800	20.0	-16916.8087	64	0.111×10^{-1}	27.29	*Normal*
		10^5	-16935.4180	106	0.602×10^{-4}	2.12	*End*
						0.23	
3c1	1250 × 2800	50.0	-16889.5623	55	0.102×10^{-12}	24.30	*Normal*
		10^5	-16939.3730	87	0.221×10^{-14}	2.18	*End*
						0.25	

Table 4.2 *Output information from BUNDECOMP*

* – *DX is set at* 10^{-6}*; rest are set at* 10^{-7}.

(Continued on the next page)

Problem Name	Size of whole problem	EPS/ DF1	Objective value : primal/dual	ITER/ EVAL	Gradient Accuracy/ Relative	Time in Min.	*Stopping Rule*
4a1	1500 × 4000	1.0	-29111.0577	19	0.159×10^{-13}	29.54	*Normal*
		1000	-29112.0566	46	0.694×10^{-16}	4.72	*End*
						0.55	
4a1	1500 × 4000	5.0	-29107.2103	25	0.621×10^{-13}	30.24	*Normal*
		1000	-29112.2090	50	0.236×10^{-15}	4.71	*End*
						0.52	
4a1	1500 × 4000	10.0	-29102.6495	23	0.119×10^{-13}	29.91	*Normal*
		1000	-29112.1914	47	0.373×10^{-16}	4.73	*End*
						0.55	
4b1	1500 × 4000*	1.0	-29010.7416	33	0.420	39.86	*Max15*
		1000	-29011.7305	86	0.482×10^{-2}	4.75	
						0.41	
4b1	1500 × 4000*	5.0	-29007.4947	38	0.492	39.75	*Max15*
		1000	-29012.4766	85	0.292×10^{-2}	4.75	
						0.41	
4b1	1500 × 4000*	20.0	-28993.0379	37	0.291×10^{-13}	35.69	*Normal*
		1000	-29013.0371	65	0.562×10^{-16}	4.77	*End*
						0.48	
5a1	2050 × 5000	50.0	-33305.2326	15	0.372×10^{-13}	45.74	*Normal*
		1000	-33355.2305	29	0.120×10^{-15}	9.10	*End*
						1.31	
5b1	2050 × 5000	20.0	-33639.6700	32	0.488×10^{-13}	54.10	*Normal*
		1000	-33659.6680	48	0.139×10^{-14}	9.64	*End*
						0.94	

Table 4.2 *Output information from BUNDECOMP*

* – *DX is set at* 10^{-6}*; rest are set at* 10^{-7}.

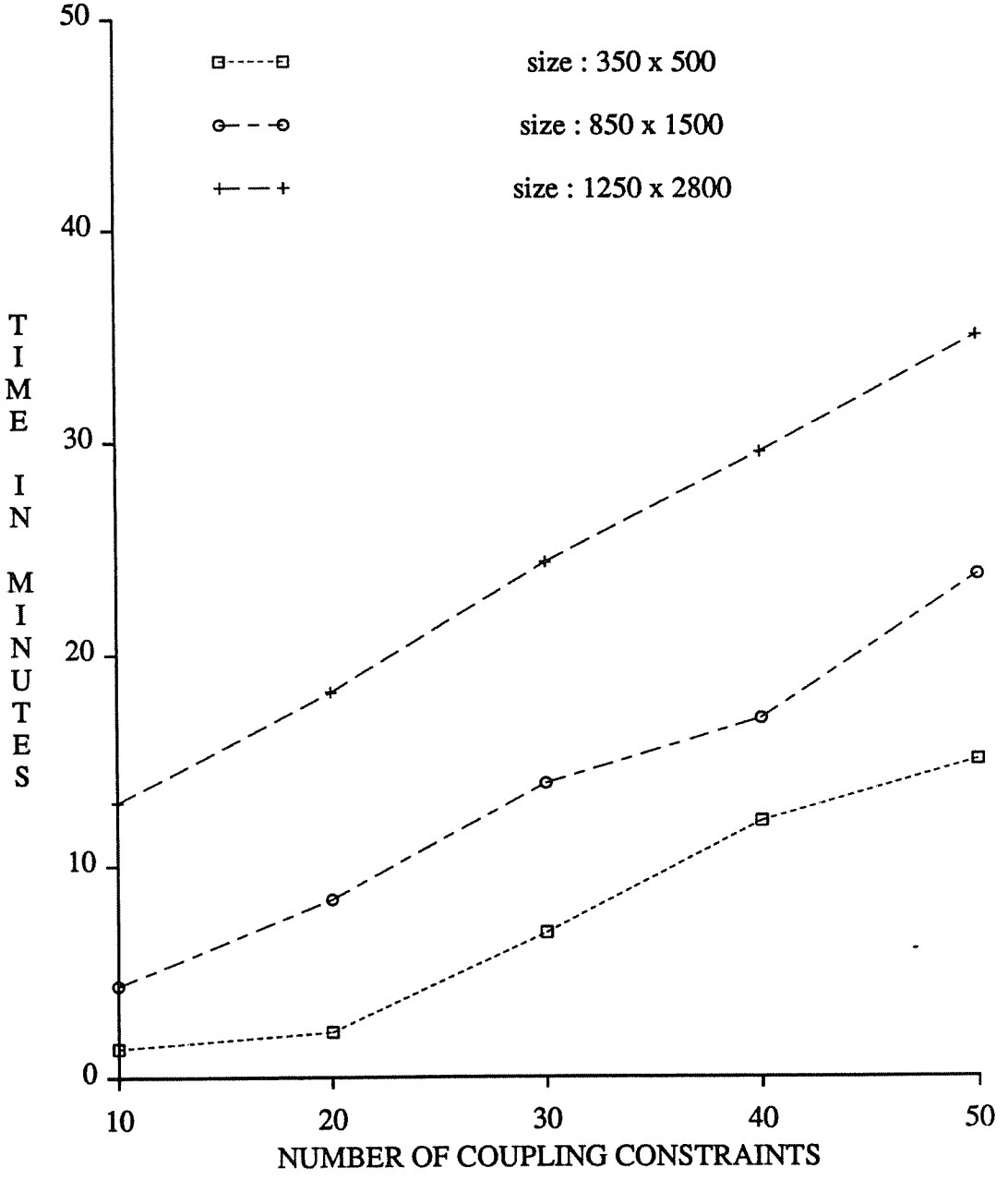

Figure 4.1 For fixed number of subproblems (100) : CPU Time in minutes against number of coupling constraints

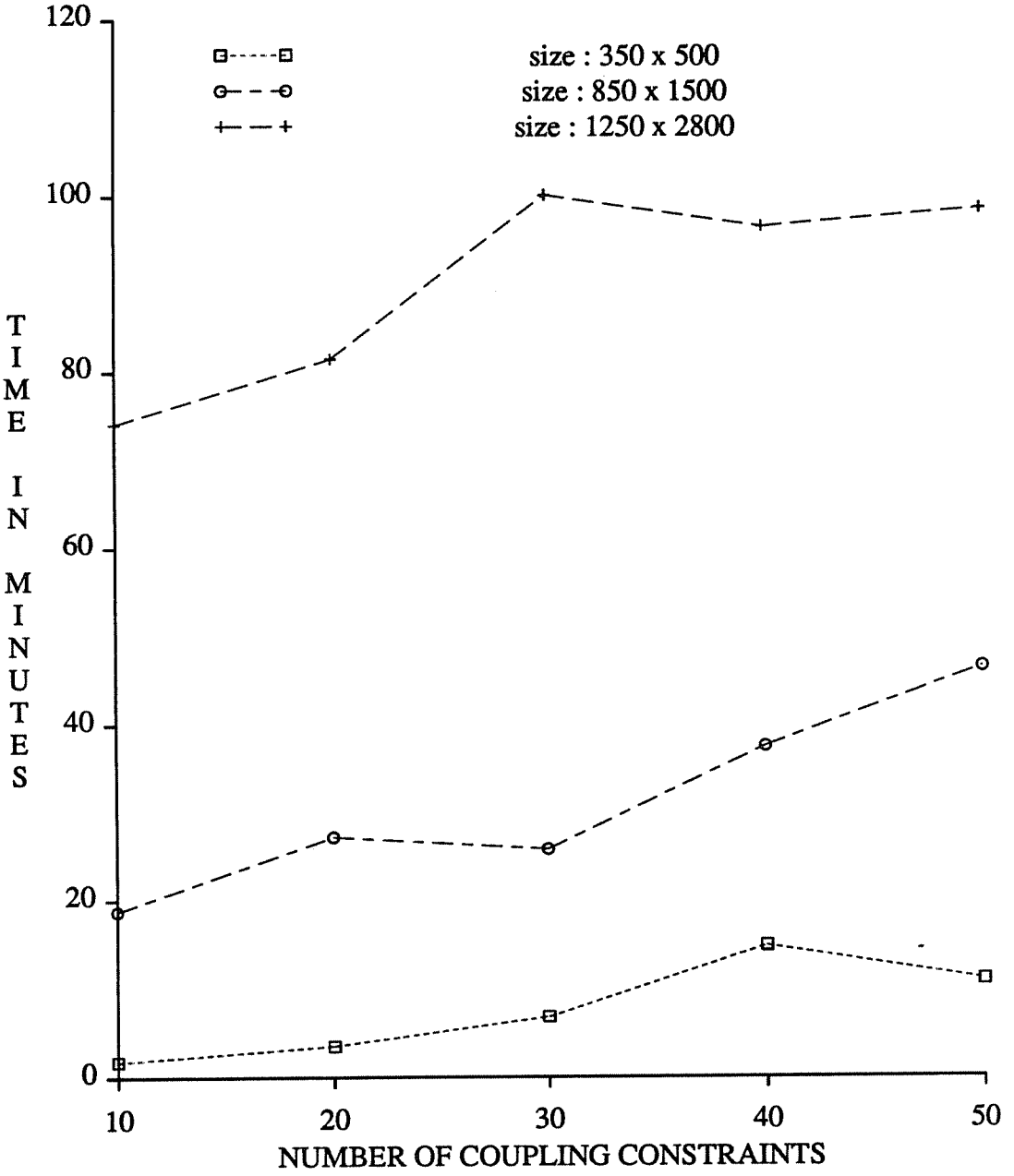

Figure 4.2 For fixed number of subproblems (40) : CPU Time in minutes against number of coupling constraints

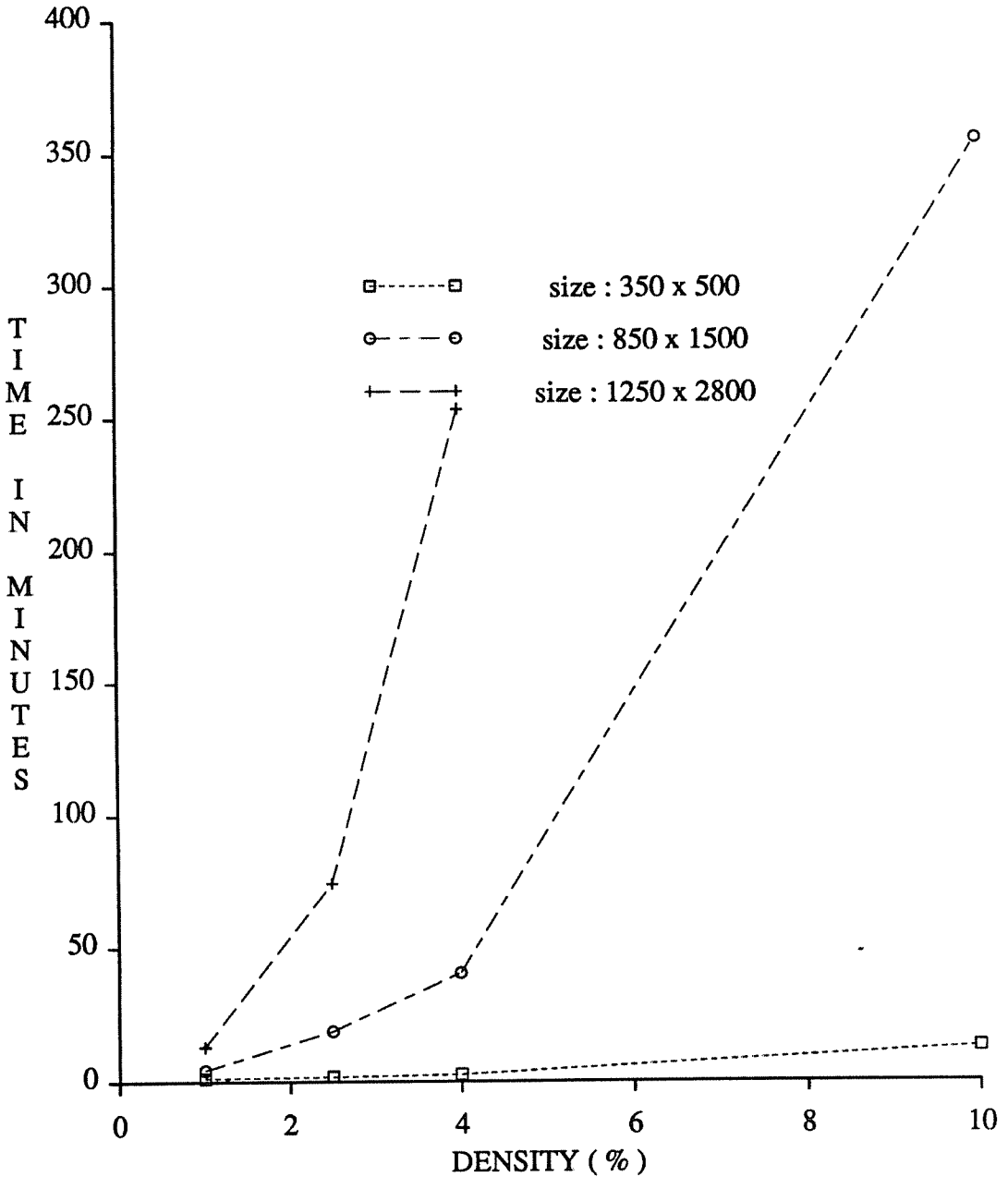

Figure 4.3 Time against density (coupling constraints : 10)

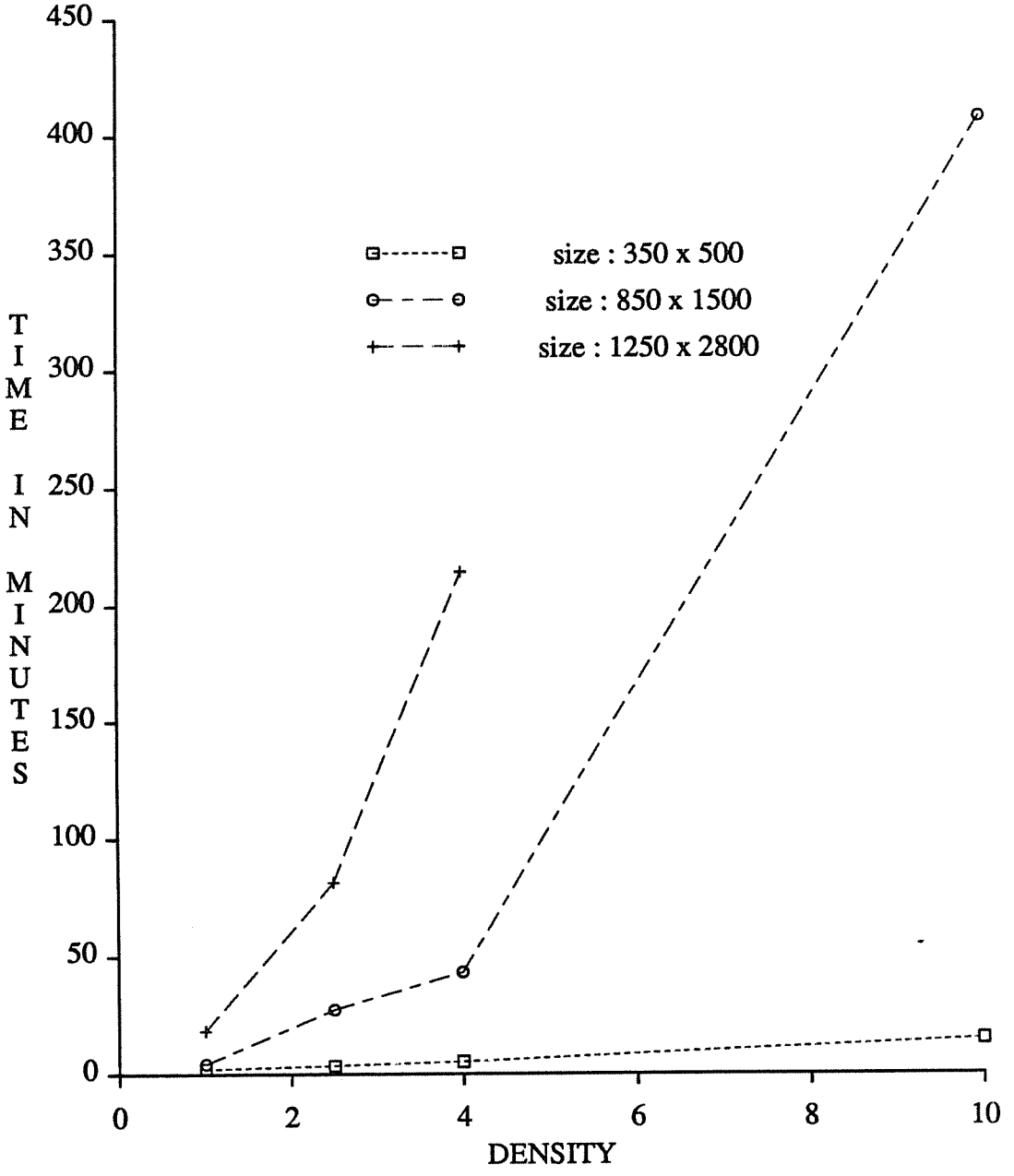

Figure 4.4 Time against density (for coupling constraints : 20).

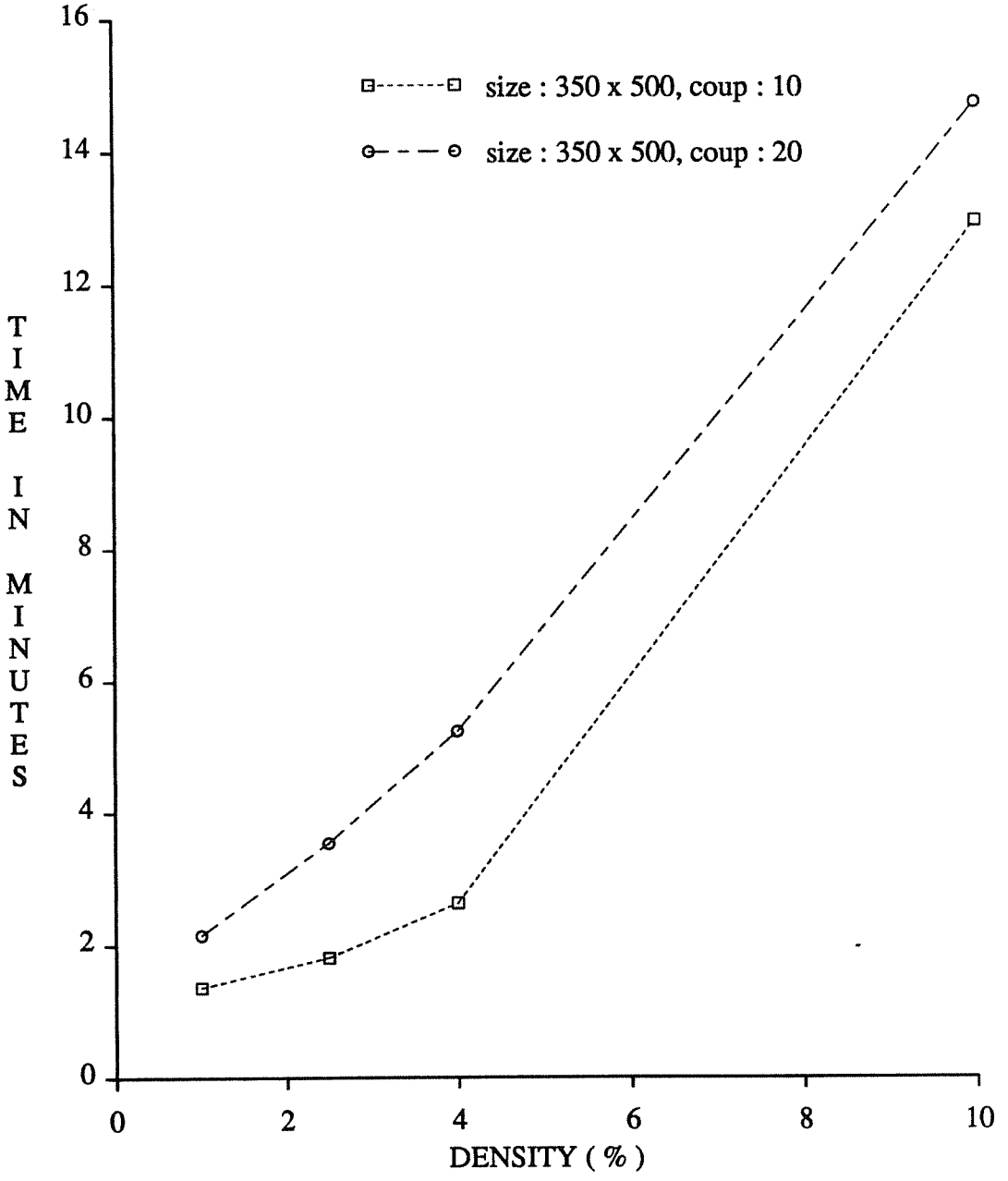

Figure 4.5 Time against density (%) for 350 x 500 problems

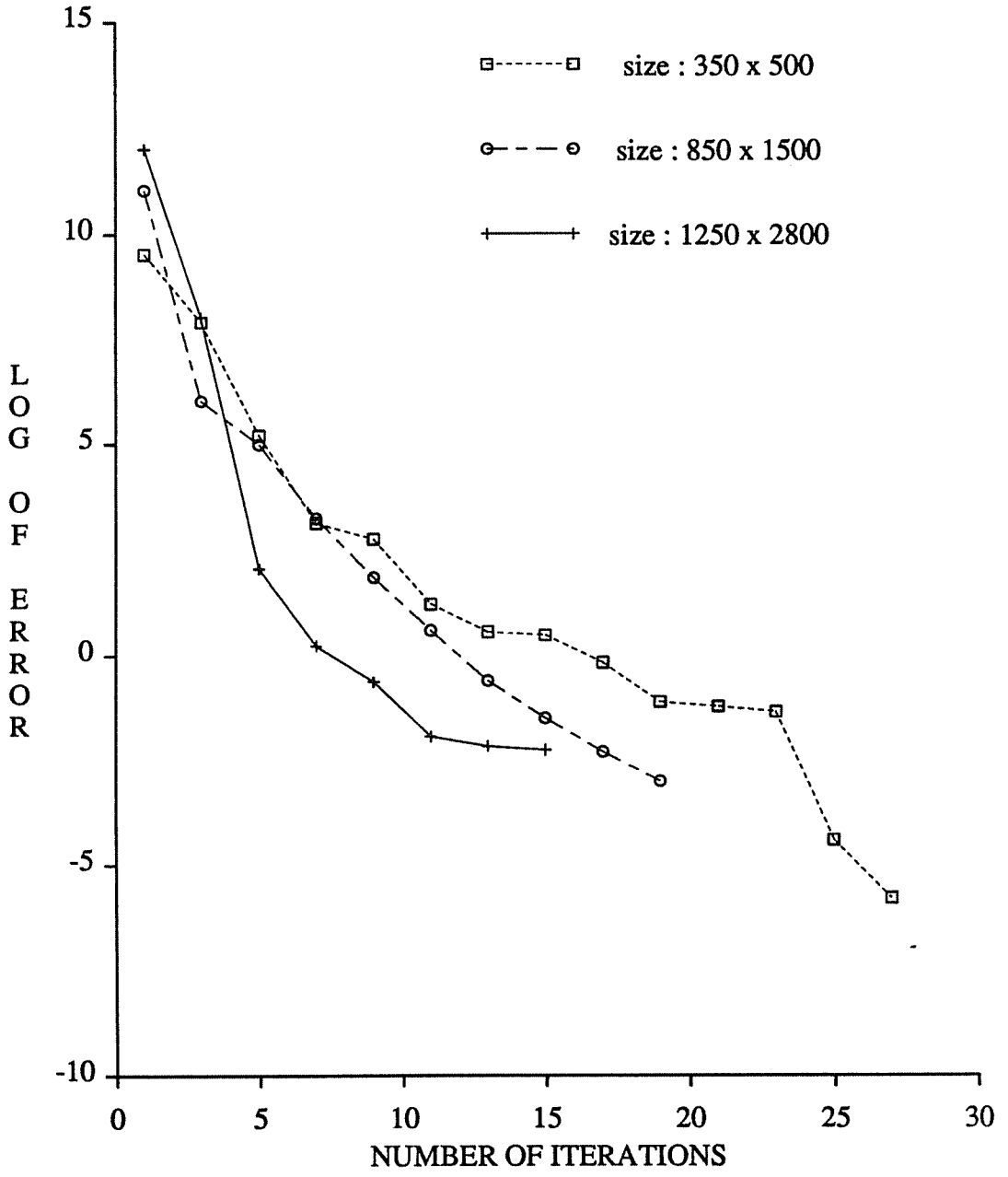

Figure 4.6 Log of error against number of iterations, where error is the difference between the dual objective value at the current iterate to the final dual objective value (number of coupling constraints = 10)

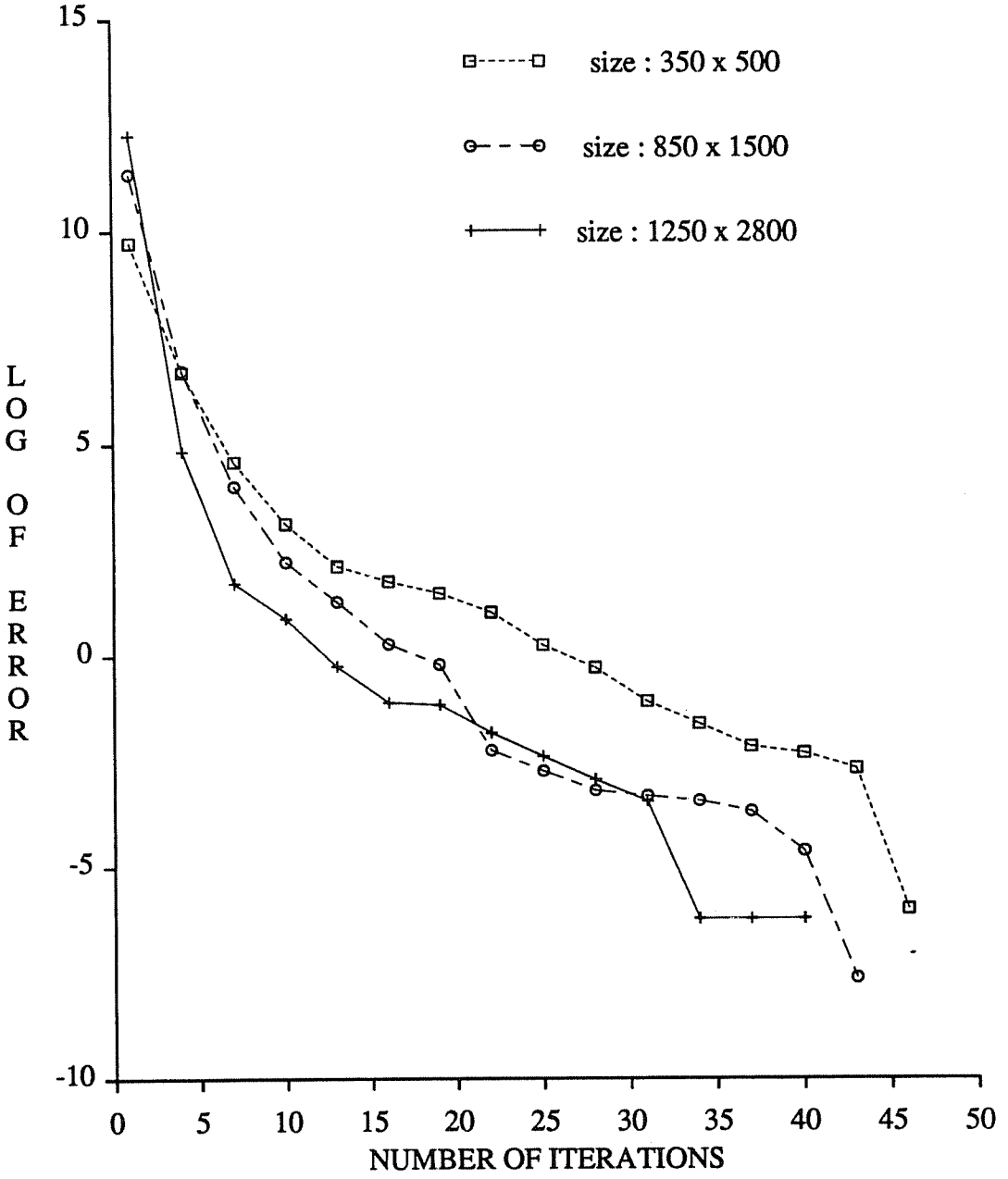

Figure 4.7 Log of error against number of iterations, where error is the difference between the dual objective value at the current iterate to the final dual objective value : number of coupling constraints is 20

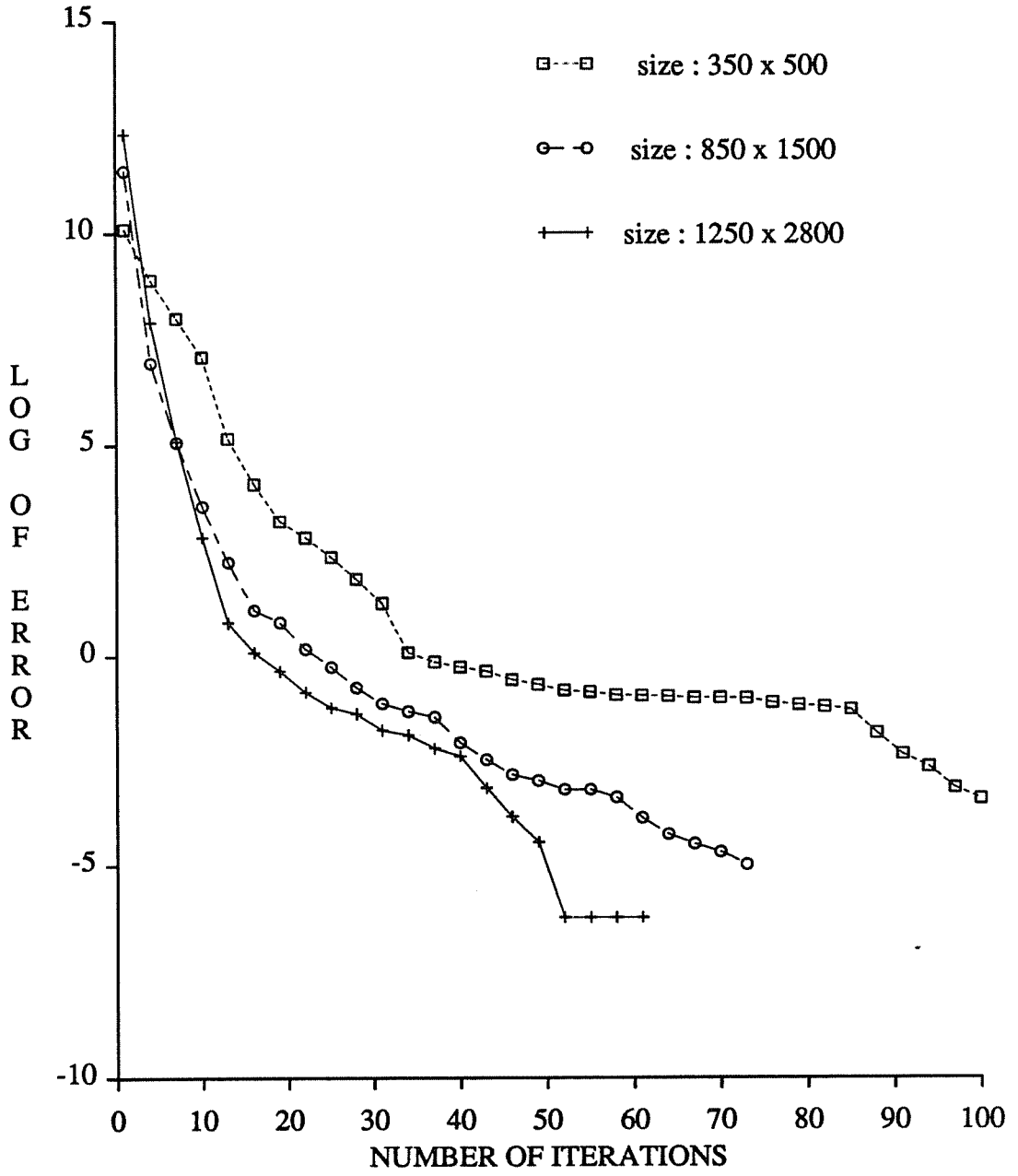

Figure 4.8 Log of error against number of iterations, where error is the difference between the dual objective value at the current iterate to the final dual objective value : number of coupling constraints is 30

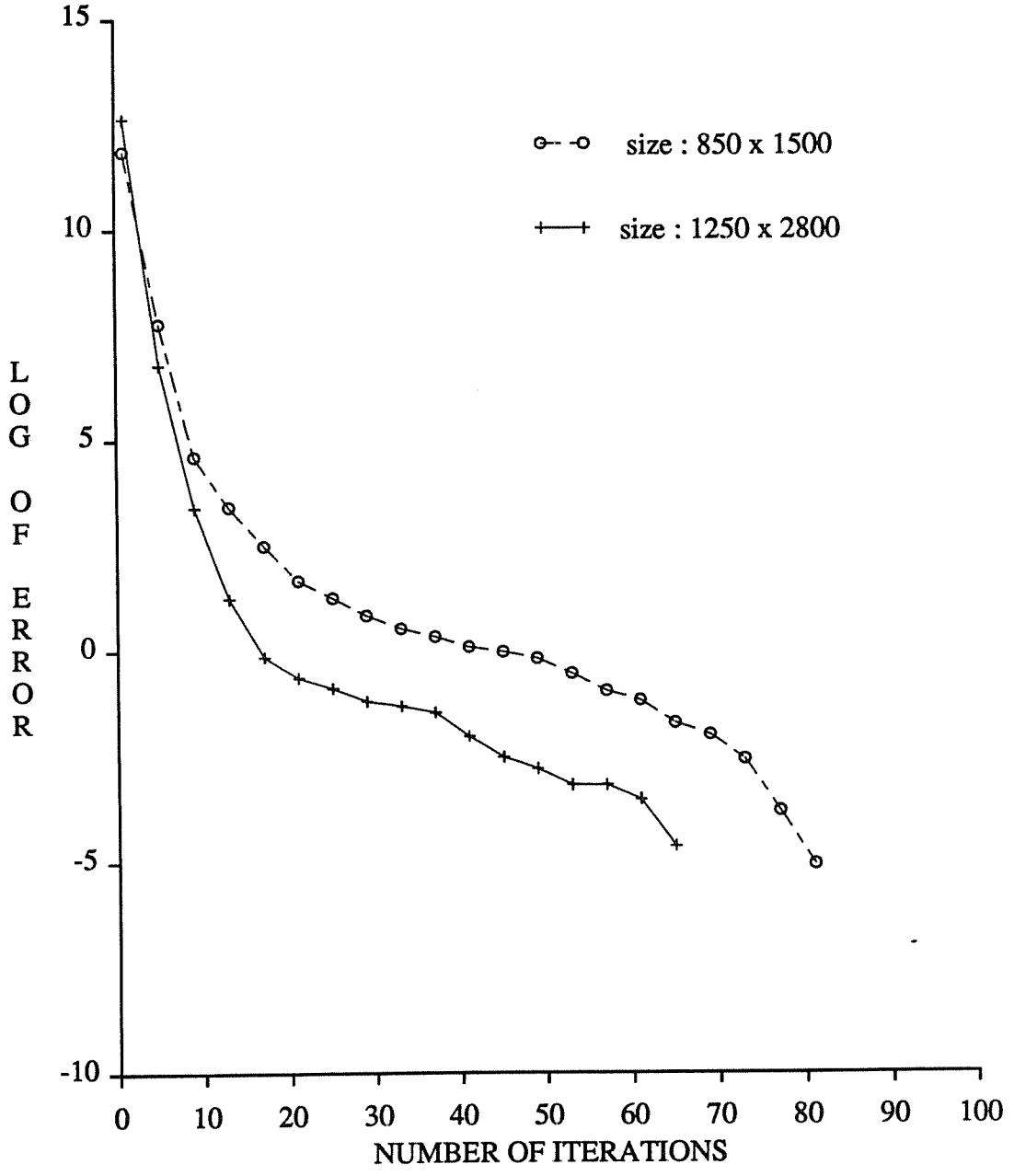

Figure 4.9 Log of error against number of iterations, where error is the difference between the dual objective value at the current iterate to the final dual objective value : number of coupling constraints is 40

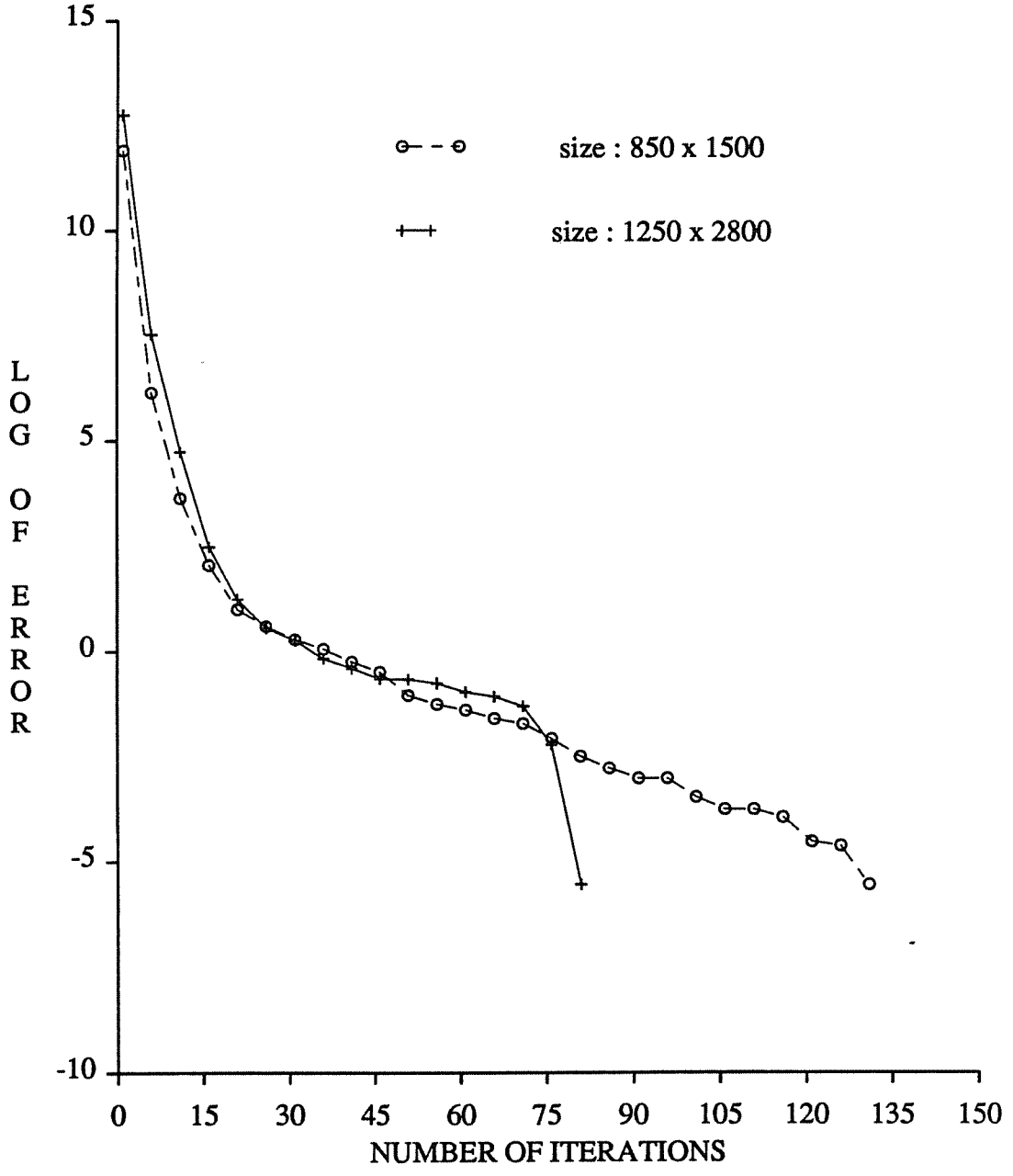

Figure 4.10 Log of error against number of iterations, where error is the difference between the dual objective value at the current iterate to the final dual objective value : number of coupling constraints is 50

4.5.2 Experiment II

In this set of experiments, we compared the algorithm *ALG 4.1* with other available methods for large-scale linear programming, namely, MINOS, a standard simplex method package [Mur83], LPSOR2 [Man84], PROSOR [DeL85], and DECOMP [Ho]. We briefly describe them here.

MINOS [Mur83] is a set of routines to solve large scale optimization problem, developed at the System Optimization Laboratory, Stanford University, by B. Murtagh and M. Saunders. It also solves linear programming problems using a reliable implementation of the simplex method maintaining a sparse LU factorization of the basis matrix. MINOS takes input data for LP in MPS format. Thus we generate our test problems first in MPS format, and then MINOS reads it from the MPS file.

LPSOR2 is a successive overrelaxation (SOR) based sparsity preserving algorithm for linear programming problems, due to Mangasarian [Man84]. It is based on the fact [Man79] that the quadratic program

$$\min_x \langle c, x\rangle + \frac{\varepsilon}{2}\langle x, x\rangle \quad \text{subject to} \quad Ax = b, x \geq 0 \tag{4.4}$$

is solvable for all $\varepsilon \in (0, \bar{\varepsilon})$ for some $\bar{\varepsilon} > 0$ if and only if the linear program

$$\min_x \langle c, x\rangle \quad \text{subject to} \quad Ax = b, x \geq 0 \tag{4.5}$$

is solvable. Also, the unique solution of (4.4) is the least 2-norm solution of the linear program (4.5), independent of ε for $\varepsilon \in (0, \bar{\varepsilon})$. The solution of (4.5) is given by

$$\bar{x} = \frac{1}{\varepsilon}(A^T u(\varepsilon) + v(\varepsilon) - c),$$

where $(u(\varepsilon), v(\varepsilon))$ is a solution of the Wolfe dual

$$\min_{(u,v)} \frac{1}{2} \parallel A^T u + v - c \parallel^2 - \varepsilon \langle b, u \rangle \quad \text{subject to} \quad v \geq 0 \tag{4.6}$$

of the problem (4.4) for $\varepsilon \in (0, \bar{\varepsilon})$. Note that $(u(\varepsilon), v(\varepsilon))$ may not be a solution of the dual of the linear program (4.5) for $\varepsilon \in (0, \bar{\varepsilon})$. The LPSOR2 algorithm is as follows.

Algorithm LPSOR2 *[Man84]*

Choose $u^0 \in \mathbb{R}^m$, $v^0 \in \mathbb{R}^n_+$, $\omega \in (0,2)$ and $\varepsilon > 0$.

Having (u^i, v^i) determine u^{i+1}, v^{i+1} as follows :

$$u_j^{i+1} = u_j^i - \frac{\omega}{\parallel A_j \parallel^2} \times \Big(A_j \big(\sum_{\substack{l=1 \\ j>1}}^{j-1} (A^T)_{\cdot l} u_l^{i+1} + \sum_{l=j}^{m} (A^T)_{\cdot l} u_l^i + v^i - c \big) - \varepsilon \, b_j \Big)$$

$$v^{i+1} = \big(v^i - \omega (A^T u^{i+1} + v^i - c) \big)_+ .$$

Here, ω is the relaxation parameter, A_j denotes the j^{th} row of A,-and $A_{\cdot l}$ is the l^{th} column of A.

A nice feature of this algorithm is that the sparsity of A is preserved as there is no need to compute AA^T directly. Also, it requires much less storage compared to a pivotal method like the simplex method. The major difficulty we faced with this method is picking ε. We shall comment about this later. A Fortran package to solve the LPSOR2 algorithm, to be called LPSOR2 also, was provided to us by R. De Leone. Note that in our case the matrix A looks like

$$\begin{bmatrix} B_1 & & & \\ & B_2 & & \\ & & \ddots & \\ & & & B_N \\ A_1 & A_2 & \cdots & A_N \end{bmatrix}.$$

We generated our test problems so as to fit the data structure in the package, LPSOR2. The user is required to provide the input parameters : ε, ω, maximum iterations and tolerance. ω is updated at subsequent iterations. The termination criteria for this package are (i) if the inf-norm of subsequent (u, v)'s is less than tolerance (exit status - 0), (ii) maximum number of iterations reached (exit status - 2).

PROSOR ([DeL85], [Del87]) is the application of the proximal point algorithm [Roc76] to the quadratic program (4.4). Using this approach if we choose a sequence of positive number $\{\gamma_k\}$, and a starting point x^0 (not necessarily feasible), and we generate x^k, then x^{k+1} is the unique solution of the problem

$$\begin{gathered} \min_x \langle c, x\rangle + \frac{\varepsilon}{2}\langle x, x\rangle + \frac{\gamma_k}{2}\langle x - x^k, x - x^k\rangle \\ \text{subject to} \quad Ax = b, x \geq 0. \end{gathered} \tag{4.7}$$

Now, instead of solving the problem (4.6) once, one needs to solve a sequence of problems of the above form. One advantage of this approach is that it stabilizes the LPSOR2 algorithm as in the LPSOR2 algorithm one does not have *a priori* knowledge of ε whereas in the PROSOR algorithm the knowledge of γ_k compensates this fact. Note that here

$$x^{k+1} = \frac{1}{\gamma_k + \varepsilon}(A^T u + v - c + \gamma_k x^k),$$

where u, v are solutions of the Wolfe dual of the problem (4.7) (similar to the problem (4.6)) for given $\varepsilon, \gamma_k, x^k$. The package to solve PROSOR algorithm, to be called PROSOR also, was also provided to us by R. De Leone. The data structure is the same as that of LPSOR2. The user needs to provide ε, γ_0, tolerance and maximum number of iterations. An initial value for ω is decided by the package. The termination criteria are (i) the inf-norm of the difference between successive x's is less than tolerance (exit status - 0), (ii) maximum number of iterations reached (exit status - 2), and (iii) relative error in the primal-dual objective is less than tolerance (exit status - 3).

Finally, DECOMP [Ho] is an implementation of the Dantzig-Wolfe decomposition method to solve the block angular linear programming problems which was provided to us by J. K. Ho. The input data for DECOMP needs to be in MPS format, and so we generated test problems first in MPS format in files, and DECOMP then read them from the files (as we did with MINOS). One major limitation of this version of DECOMP is that it can handle only up to 6 subproblems. Each subproblem in DECOMP can have up to 400 rows and 5000 nonzero elements. For documentation on DECOMP, see [Ho]. Computational experience with DECOMP can be found also in Ho and Loute [Ho84].

Due to limitations of DECOMP, we performed two different experiments. In the first one, we ran MINOS, LPSOR2 and PROSOR for the problems tried with BUNDECOMP and listed in Table 4.2, except for the problems 4a1, 4b1, 5a1 and 5b1. In the second experiment, we compared DECOMP and BUNDECOMP for a totally different set of problems.

Problem Name	Size of whole problem	Objective value : primal	Iterations total (Phase I)	Time in Min.	Partial Price/ Factorization Frequency
1a1	350 × 500	-1066.0164	630 (425)	8.824	10/50
1a2	350 × 500	-1261.0509	926 (625)	16.201	10/75
1c1	350 × 500	-1083.3470	1107 (653)	36.106	10/50
2a1	850 × 1500	-6362.8862	2896 (1606)	130.871	10/100
2a2	850 × 1500	-7068.3768	4181 (2332)	330.937	10/75
2c1	850 × 1500	-6453.7552	4386 (2349)	465.963	4/75
2c1	850 × 1500	-6453.7552	4298 (2293)	378.374	10/100
3a1	1250 × 2800	-17195.1337	5799 (2713)	790.781	4/50
3a2	1250 × 2800	-16688.2157	8472 (3812)	1336.533	10/100
3c1	1250 × 2800	-16896.6143	10008 (4493)	1785.035[1]	10/100

[1] – *program stopped as the maximum iterations reached*

Table 4.3 *Output information from MINOS*

Problem Name	Size of whole problem	eps tol	Objective value : primal/dual	Iterations	Relative Accuracy	Time in Min.
1a1	350 × 500	0.1	-1078.012	2760	0.403	28.66
		10^{-3}	-644.415		0.116	
1a1	350 × 500	0.05	-1087.249	899	0.113×10^{1}	9.30
		10^{-3}	-867.555		0.114	
1a1	350 × 500	0.005	-1094.400	5234	0.120×10^{1}	54.18
		10^{-3}	-1064.090		0.112	
1a2	350 × 500*	0.1	-1257.695	8535	0.0	115.05
		10^{-4}	-819.78		0.592×10^{-3}	
1a2	350 × 500	0.05	-1264.879	7038	0.0	94.87
		10^{-3}	-1037.303		0.606×10^{-3}	
1a2	350 × 500	0.005	-1262.378	1898	0.0	25.59
		10^{-3}	-1289.879		0.630×10^{-3}	
1c1	350 × 500	0.1	-1082.968	1464	0.348	31.16
		10^{-3}	-688.616		0.175	
1c1	350 × 500	0.05	-1090.559	1831	0.713	38.99
		10^{-3}	-884.773		0.166	
1c1	350 × 500	0.005	-1142.958	7574	0.121×10^{1}	161.26
		10^{-3}	-1079.263		0.208	
2a1	850 × 1500	0.1	-6334.942	1042	0.202	41.34
		10^{-3}	-4701.913		0.178×10^{-1}	
2a1	850 × 1500	0.05	-6352.345	2421	0.561	96.18
		10^{-3}	-5511.859		0.179×10^{-1}	
2a1	850 × 1500	0.005	-6408.652	4373	0.431×10^{1}	96.18
		10^{-3}	-6292.063		0.198×10^{-1}	
2a2	850 × 1500	0.1	-7045.179	3575	0.861	217.94
		10^{-3}	-5284.626		0.524×10^{-1}	

Table 4.4 *Output information from LPSOR2*
'Relative accuracy' := relative primal feasibility and relative dual feasibility
Dual objective value := objective value of the problem (4.6)
(Continued on the next page)

Problem Name	Size of whole problem	eps tol	Objective value : primal/dual	Iterations	Relative Accuracy	Time in Min.
2a2	850 × 1500	0.05	-7061.424	1989	0.573×10^{1}	121.08
		10^{-3}	-6160.470		0.531×10^{-1}	
2a2	850 × 1500	0.005	-7119.111	2018	0.394×10^{3}	122.66
		10^{-3}	-7022.532		0.561×10^{-1}	
2c1	850 × 1500	0.1	-6431.999	1138	0.893×10^{-1}	83.36
		10^{-3}			0.164×10^{-1}	
2c1	850 × 1500	0.05	-6450.208	2426	0.767	176.92
		10^{-3}	-5641.924		0.146×10^{-1}	
2c1	850 × 1500	0.008	-6478.718		0.314×10^{1}	289.99
		10^{-3}	-6331.762		0.114×10^{-1}	
3a1	1250 × 2800	0.1	-17092.121	843	0.347	70.05
		10^{-3}	-13191.820		0.271	
3a2	1250 × 2800	0.1	-16593.20	1402	0.652×10^{-1}	201.08
		10^{-3}	-12731.32		0.366×10^{-2}	
3a2	1250 × 2800	0.05	-16666.410	1022	0.129×10^{1}	146.34
		10^{-3}	-14637.72		0.648×10^{-1}	
3a2	1250 × 2800	0.005	-16767.490	2092	0.194×10^{2}	299.43
		10^{-3}	-16523.30		0.734×10^{-2}	
3c1	1250 × 2800	0.1	-16865.66	876	0.324	133.601
		10^{-3}	-13113.29		0.418×10^{-1}	
3c1	1250 × 2800	0.05	-16914.18	892	0.183×10^{1}	136.25
		10^{-3}	-14973.80		0.367×10^{-1}	
3c1	1250 × 2800	0.008	-16934.59	3365	0.383×10^{1}	516.24
		10^{-3}	-16612.39		0.355×10^{-1}	

* – exit status is 2; rest are 0.

Table 4.4 *Output information from LPSOR2*
'Relative accuracy' := relative primal feasibility and relative dual feasibility
Dual objective value := objective value of the problem (4.6)

Problem Name	Size of whole problem	eps gamma tol	Objective value : primal/dual	ITER	Relative Accuracy	Time in Min.
1a1	350 × 500	10^{-8},100	-1088.666	1193	0.105×10^{1}	12.27
		10^{-5}	-1089.138		0.216×10^{-2}	
1a1	350 × 500	10^{-8},100	-1087.758	1794	0.455	18.44
		10^{-6}	-1089.094		0.137×10^{-3}	
1a1	350 × 500	10^{-9},50	-1069.040	2340	0.171×10^{1}	23.96
		10^{-6}	-1069.085		0.343×10^{-3}	
1a2	350 × 500	10^{-9},75	-1271.087	613	0.532	7.95
		10^{-5}	-1267.730		0.218×10^{-1}	
1c1	350 × 500	10^{-9},75	-1097.708	8502	0.122×10^{1}	169.62
		10^{-6}	-1108.020		0.767×10^{-5}	
2a1	850 × 1500	10^{-9},75	-6362.344	6184	0.158	235.68
		10^{-6}	-6360.713		0.128	
2a2	850 × 1500	10^{-9},75	-7066.426	2711	0.466×10^{-1}	158.81
		10^{-5}	-7061.095		0.204	

* – exit status is 2; rest are 3.

Table 4.5 *Output information from PROSOR*
'Relative Accuracy' := relative primal feasibility and relative dual feasibility
Dual objective value := objective value of the Wolfe dual of the problem (4.7)
(Continued on the next page)

Prob-lem Name	Size of whole problem	eps gamma tol	Objective value : primal/dual	ITER	Relative Accuracy	Time in Min.
2c1	850 × 1500	10^{-9},75	-6453.529	5348	0.199×10^{-1}	368.55
		10^{-6}	-6452.722		0.293×10^{-1}	
3a1	1250 × 2800*	10^{-9},100	-17192.48	6500	0.196	539.15
		10^{-7}	-17192.08		0.196	
3a2	1250 × 2800	10^{-9},75	-16683.47	2391	0.944	331.19
		10^{-5}	-16675.82		0.390	
3a2	1250 × 2800	10^{-9},75	-16675.61	1439	0.310	199.42
		10^{-4}	-16632.94		0.130×10^{2}	
3c1	1250 × 2800	10^{-9},75	-16927.97	5375	0.255	791.46
		10^{-6}	-16925.50		0.149×10^{1}	
3c1	1250 × 2800	10^{-9},75	-16915.54	1692	0.500	245.36
		10^{-4}	-16883.11		0.247×10^{1}	

* – exit status is 2; rest are 3.

Table 4.5 *Output information from PROSOR*
'Relative Accuracy' := relative primal feasibility and relative dual feasibility
Dual objective value := objective value of the Wolfe dual of the problem (4.7)

Comparison of BUNDECOMP with MINOS, LPSOR2 and PROSOR

BUNDECOMP, MINOS, LPSOR2 and PROSOR were all compiled using the f77 compiler with -O (optimizer) option and were run on a DEC VAXstation II workstation running the UNIX (bsd 4.3) operating system. Computational results from MINOS, LPSOR2 and PROSOR are reported in Tables 4.3, 4.4 and 4.5, respectively. It is clear that BUNDECOMP is faster than other methods. In the case of MINOS, we did some further tests to see if we could reduce computing time. For this, we used different values than the default values for the parameters, PARTIAL PRICE and FACTORIZATION FREQUENCY, in the SPECS file. For problem 2a1, we obtained the following results :

PARTIAL PRICE	FACTORIZATION FREQUENCY	ITERATIONS	TIME (min)
10	50	2896	180.70
25	50	3082	195.68
100	50	3324	214.76
10	75	2896	147.81
10	100	2896	130.87

Table 4.6 *Problem 2a1 run on MINOS with different parameters*

From the table, we see that by varying parameters, we can reduce number of iterations by about 14 % and time by not more than a half. Even at the 'best' of both, BUNDECOMP is faster by a factor of 25. Observing that partial price set at 10 gives the least number of iterations, we tried most of the test problems with this value of partial price.

The major difficulty we faced with LPSOR2 is choosing ε. We did three runs for each problem, mostly changing the value of ε. The values we tried are 0.1,

0.05, 0.005 or 0.008. Note that the dual objective value is not that of the LP dual, as the dual solutions are those of the problem (4.6). For most of the problems we obtained exit status 0. However, the relative primal feasiblity was not very good, while the relative dual feasibility was better. In most cases, we set the tolerance at 10^{-3}. We have observed that if we decrease the tolerance value, it takes many more iterations without making much improvement in the objective value.

PROSOR appears to be more reliable than LPSOR2 in the sense that with PROSOR, in almost all cases, the primal and the dual objective values were closer to the actual values compared to the ones obtained using LPSOR2. The initial value of γ_0 sometimes affects convergence as can be seen for problem 1a1. The dual objective value is quite close to the primal value, and so we may accept the dual solution u of the modified problem (4.6) to be an approximate solution of the LP dual. The tolerance value makes a lot of difference in terms of both iterations and computation time. For example, for problem 3c1, when we decreased tolerance from 10^{-4} to 10^{-6}, the number of iterations and the time increased by a factor of more than three. Though we did not obtain very good accuracy with PROSOR, De Leone and Mangasarian [DeL87] have reported better accuracy with bigger test problems with solution density of less than 33%. PROSOR is designed for enormous problems which cannot be handled by pivotal methods.

Finally, in Table 4.7, we present timing comparisons of the different methods. We decided to choose the times for which the primal objective values for

BUNDECOMP, LPSOR2 and PROSOR are the closest to that of MINOS, except for problem 3c1. From the table, we can see that with certain problems BUNDECOMP is about seventy times faster than MINOS. BUNDECOMP is also substantially faster than both LPSOR2 and PROSOR.

Problem	BUNDE-COMP	MINOS		LPSOR2		PROSOR	
Name	Time	Time	Factor	Time	Factor	Time	Factor
1a1	1.37	8.82	6.44	28.66	20.92	23.96	17.49
1a2	1.81	16.20	8.95	25.59	14.14	7.95	4.39
1c1	6.14	36.11	5.88	31.15	5.07	169.62	27.63
2a1	6.82	130.87	19.19	140.25	20.56	235.68	34.56
2a2	18.75	330.87	17.65	121.08	6.46	158.81	8.49
2c1	13.85	465.96	33.64	176.92	12.77	368.55	26.61
3a1	11.55	790.78	68.47	70.05	6.06	539.15	46.68
3a2	70.05	1336.53	19.08	299.43	4.27	331.19	4.73
3c1	26.47	1923.83*	72.68*	516.24	19.50	791.46	29.90

* – stopped at maximum iterations

Table 4.7 *Comparison of different methods*
Time in minutes

Comparison between BUNDECOMP and DECOMP

The test problems we tried both with DECOMP and BUNDECOMP are different than the once we tried with other methods. This is due to some restrictions of the version of DECOMP we used, e.g., the number of subproblems allowed and the number of nonzero elements in each subproblem. The working version of DECOMP that we used only allows up to six subproblems. So, with some test problems where we generated more than six subproblems, we combined two or more into one 'bigger' subproblem so that the 'modified' total number of

subproblems is not more than six. For example, if we originally generated twelve subproblems, then we put them into six bigger subproblems by combining two subproblems into one bigger subproblem. This means the first two blocks of the original subproblems, whose feasible regions were

$$S^1 := \{x_1 \in \mathbb{R}^{n_1} \mid B_1 x_1 = b_1, x_1 \geq 0\}$$

and

$$S^2 := \{x_2 \in \mathbb{R}^{n_2} \mid B_2 x_2 = b_2, x_2 \geq 0\},$$

are combined into one feasible region in $\mathbb{R}^{n_1+n_2}$ and the constraints are considered to be

$$\begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \quad \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq 0.$$

Similarly, we combine three and four together, five and six together and so on. So, a problem with 12 subproblems, each of size 10×20, is regarded as a problem with six 'bigger' subproblems, each of size 20×40.

With DECOMP the strategy parameter used for the Dantzig-Wolfe algorithm is the one where each subproblem sends one proposal at optimality [Ho]. This is the only strategy parameter that is working for the version of DECOMP we used. Also one should note that this version of DECOMP did *not* obtain a primal optimal solution whereas BUNDECOMP obtained an approximate primal optimal solution.

We give the list of the test problems in Table 4.8. All the test problems with both the algorithms were run on a VAX 11/780 running the VMS operating

system. The first four test problems have six subproblems each. So, as far as the first four problems are concerned, both the codes DECOMP and BUNDECOMP, have six subproblems to begin with. The computational results of these four subproblems (i.e., x1, x2, x3, x4) with both DECOMP and BUDECOMP are reported in Table 4.9. For these problems, BUNDECOMP appears to be about three times faster than DECOMP.

The rest of the test problems (x5, x6, x7, x8, x9) from Table 4.8 have more than six subproblems each. Since DECOMP can handle only up to six subproblems we aggregated in the way described above. With regard to the code BUNDECOMP, we ran it both ways, i.e., we ran it considering it as having six 'bigger' subproblems and also as having the original smaller subproblems. For example, the test problem x8 is solved considering it as having six subproblems, each of size 80×150, and also, with sixty subproblems, each of size 8×15 (note that this test problem has twenty coupling constraints). With both the methods (DECOMP and BUNDECOMP), the subproblems to be solved are linear programming subproblems of similar size and structure. In DECOMP, these subproblems are solved using a sophisticated implementation of the revised simplex method by J. Tomlin [Ho] and storing only the nonzero entries of the coefficient matrix. But in BUNDECOMP, the linear programming subproblems are solved using IMSL routine, ZX0LP, which stores the coefficient matrix in its entire form, and so is not taking full advantage of the nonzero/zero information. Suppose we run both the methods considering the problem to have six subproblems. Then

for the test problem x8 the size of each subproblem is 80 $\times$ 150; here, the linear programming subroutine of the DECOMP code to solve the subproblems takes advantage of the structure by storing only the nonzero elements, but the subroutine, ZX0LP, in BUNDECOMP, is working with the whole size 80 $\times$ 150. So, one way for the BUNDECOMP to take advantage of the nonzero/zero information is to have it solve the 'bigger' subproblems as several small ones together (with the test problem x8 this would be ten small ones for each big one). This means solving the original smaller linear programming subproblems by ZX0LP separately.

We ran test problems x4, x5, x6, x7, x8 and x9 from Table 4.8 with DECOMP and two versions of BUNDECOMP described above. From above discussion, we think that the comparison of DECOMP solving six 'bigger' subproblems (and the linear programming routine taking advantage of the structure) with BUNDECOMP solving the same problem (but solving each of the original subproblems separately by ZX0LP) is a more reasonable comparison than BUNDECOMP solving six 'bigger' subproblems (and ZX0LP not taking advantage of the structure). We report both the times in Table 4.10.

The number of iterations with the two versions of BUNDECOMP differ. This, we think, is due to the fact that the solutions of the aggregated linear programming subproblems may not be the same as the solutions of the smaller linear programming subproblems put together. Thus the two version may take different trajectories to the dual optimal solution. Also, please note that DECOMP did not reconstruct a primal optimal solution at the end. In the Dantzig-Wolfe

decomposition method, a primal optimal solution to the original problem is usually recovered by solving *again* N LP subproblems, this time each with $(m_i + m)$ constraints (instead of m_i constraints) and n_i variables ([Dir79], [Ho81]). This is done after reaching optimality conditions for the Dantzig-Wolfe decomposition method. So DECOMP would have taken some more time if it had to solve these LP subproblems to obtain a primal optimal solution. But with BUNDECOMP, the time reported includes recovering an approximate primal optimal solution and computing primal objective value with this solution. From the Table 4.10, we see that the version of BUNDECOMP solving the original LP subproblems separately is much faster than DECOMP.

Problem Name	Size of whole problem	Number of coupling constraints	Number of sub-problems	Size of sub-problems	Density %
x1	150 × 300	10	6	23 × 50	22.22
x2	150 × 300	20	6	23 × 50	27.78
x3	200 × 450	10	6	32 × 75	20.83
x4	200 × 450	20	6	32 × 75	25.00
x5	320 × 550	10	12	25 × 46	11.20
x6	320 × 550	20	12	25 × 46	14.06
x7	500 × 900	10	60	8 × 15	3.63
x8	500 × 900	20	60	8 × 15	5.60
x9	600 × 1200	10	60	10 × 20	3.31

Table 4.8 *Problem Specifications*

Problem Name	DECOMP Time in min, T_d	DECOMP Number of Cycles	BUNDECOMP Time in min, T_b	BUNDECOMP Number of Iterations	Factor $= T_d/T_b$
x1	5.75	26	1.90	35	3.023
x2	9.12	32	2.37	52	3.854
x3	14.14	30	4.78	33	2.959
x4	24.55	43	5.22	38	4.703

Table 4.9 *Comparison of DECOMP and BUNDECOMP*
Six subproblems to start with

Problem Name	DECOMP Time in min	DECOMP Number of Cycles	BUNDECOMP As given originally Time in min	BUNDECOMP As given originally Number of Iterations	BUNDECOMP As six subproblems Time in min	BUNDECOMP As six subproblems Number of Iterations
x5	12.20	27	3.03	18	10.79	19
x6	26.07	48	4.49	50	13.58	52
x7	9.45	28	1.01	23	28.13	19
x8	25.97	55	1.86	41	34.09	39
x9	19.94	37	1.31	20	na	na

Table 4.10 *Comparison of DECOMP and BUNDECOMP*
More than six subproblems to start with
na := *not available*

4.6 Conclusion

In this chapter, we presented our experience with the bundle-based algorithm for block-angular linear programming problems and how it performs in regard to other methods. In the first set of experiments, we observed the behavior of BUN-DECOMP as different factors are varied. In the second set of experiments, we compared it with other existing solution methods. From the performance of the algorithm on the test problems, we infer that the bundle-based algorithm runs substantially faster than the other methods tested. We found that with some problems it runs up to about seventy times faster than MINOS. It appears that as the problem size grows, our algorithm does increasingly better. Since it has been reported ([Dir79], [Ho83], [Ho84]) that the Dantzig-Wolfe decomposition method does not run significantly better than good simplex code solving the problem directly, and since from our computational experience we see that our approach does substantially better than both MINOS and DECOMP, the bundle-based decomposition algorithm can be considered as a very promising alternative to other methods for solving block-angular linear programming problems, especially when a sophisticated code for solving the linear programming subroutines is used instead of ZX0LP.

Chapter 5

PARALLEL ALGORITHMS : IMPLEMENTATION AND COMPUTATIONAL EXPERIENCE

Parallel computation offers opportunities to perform computations faster than they can be done with a single processor machine. With the advancement of computer architecture, parallel processors are now available to test parallel algorithms and to observe the speed up that can be achieved. In this chapter, we propose parallel algorithms for problems of type (1.1), and present implementation and computational experience on the CRYSTAL multicomputer [DeW84]. Before, we go into our main discussion, we give below a brief review of parallel processors and a review of work on parallel algorithms done in the field of optimization.

Parallel processors can be divided into two main classes : MIMD and SIMD [Fly66]. In the MIMD (Multiple Instruction stream, Multiple Data stream) model, each processor has a separate flow of control. The processors can compute independently of one another. MIMD can be further divided into two subclasses :

A *multicomputer* consists of several computers connected by a communication medium (e.g., the CRYSTAL multicomputer). A *multiprocessor* consists of several CPU's with shared access to a common memory (e.g., the Sequent BALANCE multiprocessor). In the SIMD (Single Instruction stream, Multiple Data stream) model, all processors execute the same instruction at the same time on their own data (e.g., Illiac IV).

Very often large-scale optimization problems have special structure. Exploiting the special structure, one can hope to propose parallel algorithms and thereby reduce the computational time considerably. Recently, several authors have proposed and tried parallel algorithms for problems arising in the field of optimization. Notables among them are Feijoo [Fei85a], Feijoo and Meyer [Fei85b], Chen [Che87], Chang [Cha86] for large-scale network optimization problems on the CRYSTAL multicomputer, Zenios and Mulvey [Zen86] for network optimization problems on the CRAY X-MP/4, Philips and Rosen [Phi86] for concave minimization and linear complementarity problems on the CRAY X-MP/4, Mangasarian and De Leone ([Man86], [DeL87]) for linear programming problems and linear complementarity problems on the CRYSTAL multicomputer and the Sequent BALANCE multiprocessor, Thompson [Tho87] for linear complementarity problems on the CRYSTAL multicomputer and the BALANCE multiprocessor.

Here we propose parallel algorithms and present computational experience for problems of type (1.1). These parallel algorithms are based on the decomposition technique described in §3.2. We have implemented them on the CRYSTAL

multicomputer for block-angular linear programming problems. This chapter is organized as follows : Section 5.1 gives different variants of parallel algorithms and Section 5.2 consists of an overview of the CRYSTAL multicomputer and the Simple Application package. In Section 5.3, we describe implementation on the CRYSTAL multicomputer, and we report computational experience in Section 5.4.

5.1. Parallel Algorithms

Recall that the problem (1.1) is :

$$\inf_{x_1,\ldots,x_N} \quad \sum_{i=1}^{N} f_i(x_i) \tag{1.1a}$$

subject to

$$\sum_{i=1}^{N} A_i x_i = a, \tag{1.1b}$$

where $a \in \mathbb{R}^m$, and for $i = 1,\ldots,N$, $A_i \in \mathbb{R}^{m\times n_i}, x_i \in \mathbb{R}^{n_i}$, and, f_i's are closed proper convex functions taking values in the extended real line $(-\infty,\infty]$.

Using the decomposition technique described in §3.1, we get the following nonsmooth dual problem :

$$\max_{y} \; g(y)$$

where

$$g(y) := \langle y, a\rangle - \sum_{i=1}^{N} f_i^*(A_i^T y)$$

and where

$$f_i^*(A_i^T y) = \sup_{x_i} \{\langle A_i^T y, x_i\rangle - f_i(x_i)\}.$$

We observe that for given y the subproblems are *independent* of one another. Thus, by exploiting this fact, we can solve the subproblems in parallel, i.e., we can solve the subproblems in parallel to obtain the dual objective and a subgradient, and then update y by the bundle method. Once this is done, we can solve the subproblems in parallel for this new y to update y again, and we can continue until the stopping criterion for the bundle method is met. We propose to have a processor for coordinating the work to solve the subproblems to update y and for doing the work of the bundle method (outside function and gradient evaluation), and the rest of the processors to solve the subproblems.

We consider here two situations : when the subproblems are of even size and when they are of uneven size. By even size, we mean that the expected time to solve each subproblem is almost equal, and by uneven size, we mean that different subproblems take substantially different amounts of time to solve. We explain in the discussion on uneven problems (§ 5.1.2) why we look separately at the case of subproblems of uneven size.

5.1.1 Subproblems of Even Size

First, we consider the situation where we have $(N+1)$ processors at hand. As mentioned before, we are assuming in this section that the expected time to solve each subproblem is almost equal. Now, for problem (1.1), we can assign the work of solving the master dual problem to processor $(N+1)$, and the subproblems $1, 2, \ldots, N$ to processors $1, 2, \ldots, N$. We present below a parallel algorithm for this situation.

Algorithm ALG 5.1

Step 0. Same as the algorithm ALG 3.1.

Step 0.1 Assign the master (dual) problem to processor $(N+1)$, and the subproblems $1, \ldots, N$ to processors $1, \ldots, N$, respectively.

Step 1. Send the value of y^1 to processors $1, \ldots, N$. Keep processor $(N+1)$ idle. Solve the subproblems $1, \ldots, N$ in processors $1, \ldots, N$ in parallel. Send the solution x_i and the objective value z_i to processor $(N+1)$. Keep processors $1, \ldots, N$ idle.

Step 2. Same as the algorithm ALG 3.1.

Step 3. Same as the algorithm ALG 3.1.

Step 4. Same as the algorithm ALG 3.1.

Step 5. Same as the algorithm ALG 3.1, except for computing the dual objective and gradient.

Send u^{k+1} to processors $1, \ldots, N$. Keep processor $(N+1)$ idle.

Solve the subproblems $1, \ldots, N$ in processors $1, \ldots, N$ in parallel. Send the solution x_i and the objective value z_i to processor $(N+1)$. Keep processors $1, \ldots, N$ idle. In processor $(N+1)$, compute the dual objective and a subgradient.

The above situation can be visualized as a *queen-worker* situation. Here, the queen is doing the part of the bundle algorithm, while the workers solve the subproblems. When the workers are busy, the queen is idle. When the workers finish, they send information to the queen. Now, queen starts working again, and the workers are idle, and the process goes on.

Consider the fact that the queen is idle when the workers are busy solving subproblems. There is no reason to keep the queen idle as we can allocate a subproblem to the queen to solve simultaneously with the workers. So, this means that we can use N processors instead of $(N+1)$ processors by letting the queen do the job of one worker. This saves us one processor, and the queen will be 'busy' all the time. So, a variant of of the algorithm ALG 5.1 for the same problem with N processors is presented below. Here, we omit steps 0, 2, 3, 4 as they are the same as in the algorithm ALG 5.1.

Algorithm ALG 5.2

Step 0.1 Assign the master (dual) problem to processor 1 (queen), and the subproblems $1, \ldots, N$ to processors $1, \ldots, N$, respectively.

Step 1. Send the value of y^1 to processors $2, \ldots, N$.

Solve the subproblems $1,\ldots,N$ in processors $1,\ldots,N$ in parallel. Send the solution x_i and the objective value z_i from processors $2,\ldots,N$ to processor 1. Keep processors $2,\ldots,N$ idle.

Step 5. Same as the algorithm ALG 4.1, except for computing the dual objective and gradient.

Send u^{k+1} to processors $2,\ldots,N$.

Solve the subproblems $1,\ldots,N$ in processors $1,\ldots,N$ in parallel. Send the solution x_i and the objective value z_i to processor 1. Keep processors $2,\ldots,N$ idle. In processor 1, compute the dual objective and a subgradient.

Now, suppose we have fewer processors than the number of subproblems. Let us denote the number of processors available by p. Recall that we are assuming that expected time for solving each subproblems is about the same. In this situation, we can try to 'equally' divide the number of subproblems, N, among the processors, p. Suppose that p divides N. Then each processor is allocated

$$\frac{N}{p} := r$$

subproblems. If p does not divide N, then up to q processors, allocate

$$\left\lceil \frac{N}{p} \right\rceil := r_1 \tag{5.1}$$

subproblems, where $q = N - p\lfloor \frac{N}{p} \rfloor$, and, for the rest of the processors (i.e., $p-q$), allocate

$$\left\lfloor \frac{N}{p} \right\rfloor := r_2 \tag{5.2}$$

subproblems. Thus a variant of the above algorithm *ALG 5.2* in steps 0.1, 1 and 5 gives us the following algorithm :

Algorithm ALG 5.3

Step 0.1 Check if p divides N. If it does, then $r_1 = r$, $r_2 = r$, else allocate as defined by (5.1) and (5.2). Assign the master(dual) problem to processor 1, and subproblems 1 to r_1 in processor 1, $r_1 + 1$ to $2r_1$ to processor 2, and so on up to q processors (with r_1 in each); then subproblems $qr_1 + 1$ to $qr_1 + r_2$ to processor $q + 1$, and so on to the last processor p (with r_2 in each).

Step 1. Send the value of y^1 to processors $2, \ldots, p$.

Run processors $1, \ldots, p$ in parallel to solve the N subproblems, (solving the subproblems allocated to each individual processor serially). Send the solution x_i and the objective value z_i from processors $2, \ldots, p$ to processor 1. Keep processors $2, \ldots, p$ idle.

Step 5. Same as the algorithm ALG 4.1, except for computing the dual objective and a subgradient.

Send u^{k+1} to processors $2, \ldots, N$.

Run processors $1, \ldots, p$ in parallel to solve the N subproblems, (solving the subproblems allocated to each individual processor serially). Send the solution x_i and the objective value z_i from processors $2, \ldots, p$ to processor 1. Keep processors $2, \ldots, p$ idle. In processor 1, compute the dual objective and a subgradient.

5.1.2 Uneven subproblems

Now, suppose the solution time required for each subproblem is not expected to be equal. Then the 'equal' allocation as done in ALG 5.3 may not be a good idea. To illustrate this point, consider the following example. Suppose we have two processors, and four subproblems A, B, C, D, and each requiring 3, 1, 10 and 5 minutes, respectively, to solve. If we use the algorithm ALG 5.3, then the queen is in processor 1 with subproblems A and B, and a worker is in processor 2 with subproblems C and D. So, whenever the subproblems needed to be solved, the ones in processor 1 get done in 4 minutes whereas the ones in processor 2 take 15 minutes. So, the queen is *idle* for 11 minutes and it is 15 minutes since start of the subproblems before she can start working on the master (dual) problems again. In a parallel computing environment, it is always desirable to reduce both idle time and total computing time, i.e. to do 'load balancing'. Clearly, in this example, we can do better. If we allocated problems A, B and D to processor 1 and only problem C to processor 2, then we find that the processor 1 gets done with its subproblems in 9 minutes and processor 2 in 10 minutes. Thus, the queen needs to wait only 1 minute, and it has been 10 minutes since the start of work on all subproblems, before she can do the work of the master (dual) problem again. So, we can probably do better than equal allocation by using some other allocation scheme.

So, now the question is : 'Is there a good allocation scheme that can be used in our situation with minimum overall solution time ?' Recall that our problem is to allocate N subproblems to p processors efficiently.

In the literature, this problem is known as the *independent task-scheduling problem* [Gar78]. It is known to be NP-complete for $p \geq 2$, and except for small problems, a practical algorithm is not known for $p \geq 3$.

Before, we go into some approximation algorithms, we need to define a few terms. Note that these algorithms require knowledge of solution times of each subproblems beforehand.

Let the processing time for N subproblems be, $t(i), i = 1, \ldots, N$, respectively. We denote the processors by $1, 2, \ldots, p$. Note that in our situation, a processor can solve one subproblem at a time, and it does not work on another problem until the present one is done.

A *feasible schedule* $\mathcal{F}$ is to assign each problem $i, i = 1, \ldots, N$, to a processor, $p(i), i = 1, \ldots, N$ ($1 \leq p(i) \leq p$), and a starting time $s(i) \geq 0$ such that if $p(i) = p(j)$ and $i \neq j$, then the two exclusive intervals $(s(i), s(i) + t(i))$ and $(s(j), s(j) + t(j))$ are disjoint.

We use I to denote a specific instance of our scheduling problem, and denote $A(I)$ to be the value of the schedule obtained by using a scheduling algorithm A producing a feasible schedule for I. We denote the optimal value for I by $OPT(I)$.

We present now the *largest processing time* algorithm due to Graham [Gra69]. Let $f(j)$ denote the finish time in processor j.

Algorithm LPT 5.4

Step 0. Reorder the subproblems in decreasing order of execution time, and then reindex the subproblems.

Step 1. Initially, set $f(j) = 0 \; \forall \; j$. Start with $i \leftarrow 1$.

Step 2. $j \leftarrow \min \{l \geq 1 : f(l) \leq f(k), \quad 1 \leq k \leq p\}$.

Step 3. $p(i) \leftarrow j, \quad s(i) \leftarrow f(j)$.

Step 4. $f(j) \leftarrow f(j) + t(i)$.

Step 5. If $i = n$, terminate with an overall finish time = $\max \{f(j) : 1 \leq j \leq n\}$;

Otherwise, $i \leftarrow i + 1$, and go to step 2.

Here in step 2, we find the processor with the earliest finish time, and then, in step 3, we assign subproblem i to that processor and start this problem when the problems assigned earlier to this processor are done. Step 4 is for updating, and step 5 is to check the completion of the process. Let $LPT(I)$ be the value of the schedule obtained by the algorithm LPT 5.4. This algorithm can be implemented to run in $O(N \log Np)$ time, and one gets the following performance-guarantee theorem.

Theorem 5.1. *[Gra69]*

$$LPT(I) \leq (\frac{4}{3} - \frac{1}{3p}) OPT(I).$$

Proof : See [Gra69]. ∎

Another approximation algorithm, known as the MULTIFIT algorithm, based on the *first fit decreasing* algorithm for the bin-packing problem, has been proposed by Coffman *et al* [Cof78]. The bin-packing problem may be stated as follows : Given a number of subproblems, N, and their solution time, and a bound B, a *packing* is a distribution of the N subproblems in p bins such that time taken

to solve subproblems in each bin is less than or equal to B. The goal is to minimize number, p, of bins used in packing. Next we present the *first fit decreasing* algorithm.

Let us assume for now that we have N bins (or processors). Let $f(j), j = 1, \ldots, N$ denote the finish time in each bin. Let $ffd(B)$ denote the number of non-empty bins, and T denote the total time required to solve all the subproblems serially.

Algorithm FFD 5.5

Step 0. Reorder the subproblems in decreasing order of solution time, and reindex the problems.

Step 1. $f(j) \leftarrow 0, 1 \leq j \leq N$.

Set $i \leftarrow 1$.

Step 2. $j \leftarrow \min \{l \geq 1 : f(l) + t(i) \leq B\}$

Step 3. $p(i) \leftarrow j$, $s(i) \leftarrow f(j)$.

Step 4. $f(j) \leftarrow f(j) + t(i)$.

Step 5. if $i = n$, $ffd(B) \leftarrow$ number of bins for which $f(j) > 0, 1 \leq j \leq N$; stop.

Else, $i \leftarrow i + 1$, and go to step 2.

In the above algorithm, after reordering and reindexing, a packing is done by adding each subproblem in succession to the lowest indexed bin into which it fits without violating the capacity constraints, and finally, $ffd(B)$ gives the number of nonempty bins.

In our problem, B is unknown and the number of processors, p, is given. In the MULTIFIT algorithm, B is guessed by using binary search, and repeatedly calling the FFD algorithm. The initial upper and lower bounds on B are :

$$B_{lower} := \max \{\frac{T}{p}, \max_{1 \leq i \leq N} \{t(i)\}\}, \tag{5.3}$$

$$B_{upper} := \max \{\frac{2T}{p}, \max_{1 \leq i \leq N} \{t(i)\}\}. \tag{5.4}$$

[Cof78]. Let k be the bound on the number of iterations of the algorithm FFD 5.5. Initially, the subproblems are reordered and reindexed in decreasing order of execution time so that in subsequent calls to algorithm FFD 5.5, one need not reorder again. Now, we present the MULTIFIT algorithm in a precise manner [Cof78] :

Algorithm MF 5.6

Step 1. Initialize B_{lower} and B_{upper} as given in (5.3) and (5.4). Set $i \leftarrow 1$.

Step 2. If $i > k$, halt.

Else, set $B \leftarrow (B_{lower} + B_{upper})/2$.

Step 3. Apply the algorithm FFD 5.5.

Step 4. If $ffd(B) \leq p$, then $B_{upper} \leftarrow B$;

Else, $B_{lower} \leftarrow B$.

$i \leftarrow i + 1$, and go to step 2.

The above algorithm, together with sorting, can be implemented in $O(N \log N + kN \log p)$ time. Coffman *et al* [Cof78] suggested that $k = 7$ is a reasonable choice for the number of outer iterations. Let $MF(I)$ denote the valued of the schedule obtained by the algorithm MF 5.6. Then, one gets the following performance-guarantee theorem.

Theorem 5.2. *[Cof78]*

$$MF(I) \leq 1.22\,OPT(I).$$

Proof : See [Cof78]. ∎

Thus, in the worst case, the algorithm MF 5.6 gives a better performance than the algorithm LPT 5.4. However, the algorithm LPT 5.4 takes slightly less time to run than the algorithm MF 5.6 for $k > 1$.

Essentially, the algorithm LPT 5.4 and the algorithm MF 5.6 are the two well-known algorithms for scheduling N independent subproblems in p processors. However, we should mention here that Friesen and Langston [Fri86] have improved the MULTIFIT algorithm to give a better performance guarantee. Also, Leung [Leu82] proposed another algorithm based on a dynamic programming approach, but he did not give any performance guarantee result.

From the above discussion, we can see that that we can either apply the algorithm LPT 5.4 or the algorithm MF 5.6 to schedule the N independent subproblems (uneven) in p processors. Up to this point, we have assumed that we know the solution time of each subproblems. But, in practice, we do not have *a priori* knowledge of the solution time. Instead, we estimate the solution time. We will discuss estimation procedures in a later section. Next we present the following variant of the parallel algorithm ALG 5.3 for uneven problems. Since the step 1 and the step 5 are the same as before, we mention only the step 0.1.

Algorithm ALG 5.7

Step 0.1 Estimate the solution time of each subproblems.

Rearrange the subproblems in decreasing order of execution time and reindex the problems.

Apply either the algorithm LPT 5.4 or the algorithm MF 5.6 to allocate the subproblems in p processors.

Send the data of the subproblems to the appropriate processor.

5.2 The CRYSTAL multicomputer and SAP

We have implemented algorithms ALG 5.3 and ALG 5.7 on the CRYSTAL multicomputer [DeW84] for block-angular linear programming problems (4.1). Before going into the implementation on the CRYSTAL multicomputer, we give a brief description of CRYSTAL and the Simple Application Package (SAP) here.

The CRYSTAL multicomputer is a collection of 20 DEC VAX 11/750's connected by an 80 megabit/sec Proteon Pronet token ring [DeW84]. Each of the 750's has 2 megabytes of local memory except for one which has 3 megabytes. We shall refer to each 750's as a *node* or a *processor*. Some of the nodes have disks attached to them. One can access CRYSTAL via 'host' machines. These host machines are either VAX 11/750 or 11/780 computers running the bsd 4.3 Unix operating system.

Multiple users can use the multicomputer by requesting for a subset of the collection, known as a *partition*. A partition can be obtained via the nugget command interpreter (NCI). NCI is also used for linking and loading a program on the nodes, for pausing or halting a program, to check the status of the nodes,

to free a partition when done with computation and for various other purposes. When a partition is acquired, the nodes in that partition are given a virtual node number from 1 to the number of nodes in that partition. The host is always given the virtual node number 0.

The CRYSTAL software is written in a local extension of the language, Modula. Communication between nodes and the host is accomplished by means of messages. Software for doing inter-process communication in a high level language, called the *Simple Application Package* (SAP), is available to the user. It is written in Fortran, Modula, Pascal and C. We used the Fortran version. Though a distributed operating system, Charlotte, is available on CRYSTAL, we did not use it as 1) SAP was sufficient for our purpose, and 2) Fortran is not implemented on Charlotte.

As we mentioned before, communication among the nodes is done by means of messages. Each message (or packet) can consist of up to 2048 bytes of information. The Simple Application Package supports two queues for communication buffering in each node and the host, one for incoming packets and the other for outgoing packets. The user can decide the number of buffers needed for the intended tasks. If the user wants to send an array of length needing more than 2048 bytes, the array must be split in order to send it in two or more packets. Before loading information to a packet that is going to be sent, the program busy-waits until a sending buffer is available. Once the buffer is available, outgoing data are placed in it, the destination is indicated and finally it is sent to that destination by

calling a "send" routine. The packet is also identified by the virtual node number of the sending node. When the sending is done, the buffer is freed for further use. On the receiving side, it busy-waits to check if anything is coming from another node. The virtual node number of the arriving packet can be checked to see if it is coming from the node from which it is supposed to come. Once the checking is over, it accepts the message and then frees the input buffer for later use. Communication between a host machine and a node is the same as that among the nodes.

The three main routines of the Simple Application Package used for initializing queues for communication buffering, and for sending and receiving information are : `mkbufr`, `sendbf`, and `frbufr`. Five integer variables are needed for message communication. The variables `numsen` and `numrec` are used for total number of output and input buffers, `buflth` is used for length of each buffer, `id` is used for name of the partition and `size` is used for number of nodes in that partition. Two dimensional real arrays, `bufsen` and `bufrec`, are used for buffering outgoing and incoming packets, each of size `buflth x numsen` and `buflth x numrec`, respectively. For real arrays, `buflth` could be a maximum of 512 ($=2^9$). Whenever we want to send a real vector of length more than 512, we split it into more than one packet of length 512. To identify them correctly, we use the first four bytes of each packet with a signal number. Then when the packets are received at the other end coming from the same node, the receiving node checks both the source it is coming from and the signal number it is expecting for proper identification.

The logical array, `iflag` (incoming) of length `numrec` is used for checking if a message has arrived in `bufrec`. For example, for some buffer counter, `in`, `iflag (in) = .true.` indicates that a message has arrived in `bufrec ( . ,in)`. The entry, `source (in)`, in the integer array, `source` (of length `numrec`), indicates the source the packet is coming from. Once the receiving is over, the buffer counter, `in`, is incremented by 1 *modulo* `numrec`. Similarly, there are the logical array, `oflag`, and the integer array, `dest`, both of length `numsen` on the sending side. For a buffer counter `out`, `oflag (out) = .true.` means that a message in `bufsen ( . , out )` is still being sent, and in `dest (out)` the user indicates the destination of the packet. As in for receiving, when sending is over, the buffer counter, `out`, is incremented by 1 *modulo* `numsen`. Initially, both the variables, `in` and `out`, are set to 1.

In the node machines, the integer variable `id` is replaced by the variable `me`, where `me` is the virtual node number of that node. Since the buffer arrays are declared real, double precision numbers can be sent by declaring two double precision arrays `dbfsen` of size `buflth/2 x numsen` and `dbfrec` of size `buflth/2 x numrec`, and then "equivalencing" them with the corresponding real arrays.

The routine, `mkbufr`, is called only once at the beginning of the host and the node programs for initialization. The routines, `sendbf` and `frbufr`, are used for sending and receiving messages. We give examples of sending and receiving messages in Fortran in Figures 5.1 and 5.2. Here the packet is sent from node 0

(i.e., the host) and is received at node 1. In Figure 5.1, the data are read from the array outdata, and in Figure 5.2, the data are placed in the array indata.

There are two other routines available in SAP. The routine run is used in the host program for synchronizing execution of programs running on different nodes. The routine pause is called to put a node in pause state, and it will start running when the host program calls run.

```
10 if ( oflag(out) ) go to 10
   bufsen(1,out) = signal
   do 20 i=1, k
       bufsen(i+1,out) = outdata(i)
20 continue
   dest(out) = 1
   call sendbf (out)
   out = mod(out,numsen) + 1
```

Figure 5.1 *Sending a message from node 0 to node 1 (in Fortran)*

```
10 if ( .not.  iflag(in) ) go to 10
   if ( ( source(in) .ne.  0 ) .or.
       ( bufrec(1,in) .ne.  signal ) ) then
       print *, ' Not the right buffer '
   endif
   do 20 i=1, k
       indata(i,out) = bufrec(i+1,in)
20 continue
   call frbufr (out)
   in = mod(in,numrec) + 1
```

Figure 5.2 *Receiving a message at node 1 from node 0 (in Fortran)*

5.3 Implementation on CRYSTAL

In the previous section we described the Simple Application Package. In this section, we discuss the implementation on the CRYSTAL multicomputer.

We wrote three programs : one for the host machine for initialization, the second one for doing the bundle algorithm and solving some of the subproblems, and the third one for solving subproblems and sending solutions to the second one. We shall call them the *host* program, the *queen* program and the *worker* program, respectively. The queen program always resides on the virtual node 1. The host program does initialization and generates data. The availability of a locally developed command *ct* (based on the *telnet* command to login remotely to a remote host) helped us obtain output information directly from the nodes. So, we did not send the final information back to the host when everything is done. We give a sketch of our host program in Figure 5.3.

Program Host
    Initialize buffers.
    Read input data. Let the number of subproblems be N.
    Generate Data.
    Decide where to send which subproblems.
    Send data to Queen processor (Node 1) first.
    Send data to rest of the nodes (i.e. to workers).
End

Figure 5.3 *Host Program*

The queen program in node 1 solves the main part of the bundle method and some of the subproblems. She sends y to the workers, and receives the solutions back from them. An outline of the queen program is given in Figure 5.4.

Program Queen

Initialize buffers.

Receive problem data from host.

Receive signal from the last node that it has received data from host.

Set done = false ; Start solving.

While { not done } do

Begin

Send y to the nodes $2, \ldots, p$.

Solve her share of subproblems.

Receive solutions from node $2, \ldots, p$.

Do updating for the bundle method.

If optimality reached, set done = true.

End while.

Send signal to nodes $2, \ldots, p$ to stop worker programs.

Receive timing information from nodes $2, \ldots, p$.

Print solution and timing information.

End

Figure 5.4 *Queen Program*

The worker program runs on nodes $2, \ldots, p$. It solves its share of the subproblems. It receives the dual variable y from the queen processor and sends back the solutions of the subproblems to the queen processor. We give an outline of the worker program in Figure 5.5. Here, the worker is expecting to receive y or a signal from queen to inform him that she is done. However, in the queen program, we have two separate statements, one with 'send y to the nodes $2, \ldots, p$', and the other with 'send signal to nodes $2, \ldots, p$ to stop worker programs'. Actually, in practice we also send a signal with the first statement. So, the worker, on receiving

a packet, checks which signal it is. If it is the stopping signal, he sets *done = true*; else, he solve his share of the subproblems.

```
Program Worker I
    Initialize buffers.
    Receive problem data from host.
    If I am last node, send signal to queen to start solving.
    Set done = false
    While { not done } do
      Receive y or signal of queen done from Queen.
      if { signal of queen done } then
        done = true
      else
        Solve its share of subproblems
        Send solution to queen
      endif.
    End while.
    Send timing information to Queen.
End.
```

Figure 5.5 *I^{th} Worker Program*

Once the codes for the queen and the workers are ready, they are compiled using the Fortran compiler f77 linking the library for SAP and Fortran and C library. Then, the queen and the worker programs are loaded to the CRYSTAL nodes using the *link* command in NCI environment. The whole program is started by executing the host executable and giving the partition number.

5.4 Computational Experience

We have implemented the algorithms ALG 5.3 and ALG 5.7 on the CRYSTAL multicomputer for block-angular linear programming problems. We extended the code, BUNDECOMP, to do parallel computation on the multicomputer. In order to compare the parallel algorithms to their serial version (single processor), we need a few definitions here.

Let T_p be the elapsed time required to run the parallel algorithm in p nodes (processors). Then the *speedup*, S_p, is defined as

$$S_p := \frac{T_1}{T_p}.$$

The *efficiency*, E_p, is defined as

$$E_p := \frac{S_p}{p}.$$

Theoretically, with our algorithm,

$$T_p \geq \frac{T_1}{p}.$$

So, E_p satisfies

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \leq 1.$$

Thus, one hopes for E_p to be as close to one as possible.

We first discuss our computational experience with subproblems of *even size*.

5.4.1 Case I

For block-angular linear programming problems, we assume that subproblems of almost similar dimension takes almost equal amount of time to solve them, and we call them subproblems of even size. For these we used the scheme described in the algorithm ALG 5.3 to allocate the LP subproblems. The test problems we tried are 2a1, 3a1, 4a1, 2b1, 3b1 and 4b1 from Table 4.1. The problems 2a1, 3a1 and 4a1 have 100 subproblems and 10 coupling constraints, but the sizes of the whole problems are 850 × 1500, 1250 × 2800 and 1500 × 4000, respectively. Since the subproblems are of (almost) even size, each subproblem in 2a1, 3a1 and 4a1 is of dimension (about) 8 × 15, 12 × 28 and 15 × 40, respectively. The problems 2b1, 3b1 and 4b1 are similar to 2a1, 3a1 and 4a1 except that the b's have 20 coupling constraints.

Theoretically, whether we utilize one processor or ten processors the total number of dual iterations (and calls to SIMUL) and the final primal and dual objective value should be about the same. Recall that for given y, after solving the subproblems, the dual function, $g(y)$, and the subgradient, π, are computed by using :

$$g(y) = \langle y, a \rangle - \sum_{i=1}^{N} \{ \langle A_i^T y, \hat{x}_i \rangle - f_i(\hat{x}_i) \},$$

and

$$\pi = a - \sum_{i=1}^{N} A_i \hat{x}_i,$$

where $\hat{x}_i$ is the computed solution of the i^{th} subproblem, $i = 1, \ldots, N$. Suppose we have two processors and N is odd, then if we compute

$$\sum_{i=1}^{\lfloor N/2 \rfloor} A_i \hat{x}_i$$

in one processor and compute

$$\sum_{i=\lceil N/2 \rceil}^{N} A_i \hat{x}_i$$

in another processor and then sum the results together, the resulting vector should come out to be the same as when we compute

$$\sum_{i=1}^{N} A_i \hat{x}_i$$

serially in one processor. Similar results hold for the calculation of $g(y)$. Thus, the dual function and the subgradient should come out to be the same no matter whether we used one processor or several processors. But in practice, they do not come out to be the same due to round off in numerical calculations.

After doing a preliminary testing and realizing that we did not get the same number of iterations (and other information) with different numbers of processors, we decided to send the solution $\hat{x}_i$ from worker processors to the queen processor and then have the matrix multiplications performed serially (going from $i = 1$ to N) at the queen processor to obtain the dual function value and the subgradient. When we did it this way, the number of iterations and other information were the same using different numbers of processors. In Figure 5.6, we present the speedup against the number of processors when the matrix multiplications are

done in serial for the problems 2a1, 3a1 and 4a1 and when tried on 1, 2, 3, 5, 7, 9 and 11 processors. Although the speedup (and so the efficiency) is getting better as the problem size is getting bigger, we see that speedup seems to have an asymptotic behavior as the number of processor increases. The high cost of matrix multiplication operations being done in serial is the major factor in observing an asymptotic behavior in speedup while the number of processors is still in single digits.

We wished to do better (in terms of speedup and efficiency) than what we achieved with matrix multiplication being done in serial, especially, when we realized that the matrix multiplication is a major factor for not achieving better efficiency. Thus we tried again the version of the parallel code where the matrix multiplication is being done in parallel. For the same problems, this resulted in obtaining the number of iterations and other information to be different with different numbers of processors, as we expected from our earlier observation. But at the same time, it resulted in obtaining considerably better speedup even though the number of calls to SIMUL were different. The computational results with the six test problems tried on 1, 2, 3, 5, 7, 9 and 11 processors are presented in Tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6. In Figures 5.7 and 5.8, we plotted speedup against the number of processors, the first figure with test problems 2a1, 3a1 and 4a1, and the second one with test problems 2b1, 3b1 and 4b1. The irregular behavior of some of the graphs can be attributed to

Nodes (Processor)	ITER/NSIM	Time (sec)	Speedup	Efficiency
1	20/56	475.86	1	1
2	22/71	293.33	1.622	0.811
3	20/56	181.03	2.628	0.876
5	21/68	127.00	3.746	0.749
7	19/47	79.92	5.954	0.851
9	23/66	132.30	3.596	0.399
11	19/48	94.26	5.048	0.459

Table 5.1 *Time, Speedup and Efficiency for* 2a1 (850 × 1500)

Nodes (Processor)	ITER/NSIM	Time (sec)	Speedup	Efficiency
1	19/49	1261.79	1	1
2	20/41	618.36	2.041	1.021
3	15/49	457.16	2.760	0.920
5	20/41	265.53	4.752	0.950
7	19/37	192.80	6.545	0.935
9	15/50	202.24	6.239	0.693
11	20/41	161.96	7.791	0.708

Table 5.2 *Time, Speedup and Efficiency for* 3a1 (1250 × 2800)

Nodes (Processor)	ITER/NSIM	Time (sec)	Speedup	Efficiency
1	21/48	2439.81	1	1
2	16/56	1311.32	1.861	0.931
3	21/60	929.91	2.624	0.875
5	15/46	532.72	4.580	0.916
7	19/44	396.24	6.157	0.880
9	20/44	320.77	7.606	0.845
11	19/41	263.97	9.243	0.840

Table 5.3 *Time, Speedup and Efficiency for* 4a1 (1500 × 4000)

Nodes (Processor)	ITER/NSIM	Time (sec)	Speedup	Efficiency
1	28/77	752.01	1	1
2	40/90	456.55	1.647	0.824
3	38/77	297.20	2.530	0.843
5	38/98	226.25	3.324	0.665
7	49/96	189.42	3.970	0.567
9	46/111	174.08	4.320	0.480
11	31/69	99.61	7.549	0.686

Table 5.4 *Time, Speedup and Efficiency for* 2b1 (850 × 1500)

Nodes (Processor)	ITER/NSIM	Time (sec)	Speedup	Efficiency
1	28/68	1777.49	1	1
2	28/78	1009.17	1.761	0.881
3	43/93	809.42	2.196	0.732
5	29/77	441.77	4.024	0.805
7	34/92	382.63	4.645	0.664
9	28/75	272.31	6.527	0.725
11	38/74	253.72	7.006	0.637

Table 5.5 *Time, Speedup and Efficiency for* 3b1 (1250 × 2800)

Nodes (Processor)	ITER/NSIM	Time (sec)	Speedup	Efficiency
1	29/72	3220.31	1	1
2	22/59	1501.35	2.145	1.073
3	28/71	1125.79	2.860	0.953
5	39/73	733.06	4.393	0.879
7	32/88	572.49	5.625	0.804
9	na	na	na	na
11	35/88	410.17	7.851	0.714

Table 5.6 *Time, Speedup and Efficiency for* 4b1 (1500 × 4000)
na := *not available*

i) round off errors and thus resulting in increase or decrease in calls to SIMUL (NSIM) and number of iterations, and so the solution time, and

ii) the effect of communication and idle time.

Both (i) and (ii) together can make a significant difference in speedup and efficiency for smaller problems (e.g. 2a1). But as problem size increases, the speedup graphs appear to be growing more stably even though there are some differences in the number of SIMUL calls (e.g. 4a1, 4b1). Thus communication and idle time is not a significant factor for larger problems. In case of two problems (3a1 and 4b1), we obtained efficiency greater than one using two processors. This is due to the fact that the number of SIMUL calls is more with one processor than with two processors for both these cases. Finally, comparing Figures 5.6 and 5.7, we see that a smaller number of coupling constraints means better speedup and efficiency. This is not surprising, as the number of coupling constraints determines the dimension of the dual space. So as suggested by Dirickx and Jennergren [Dir79], we also think that it is a good idea to keep down the number of coupling constraints in the modeling stage itself.

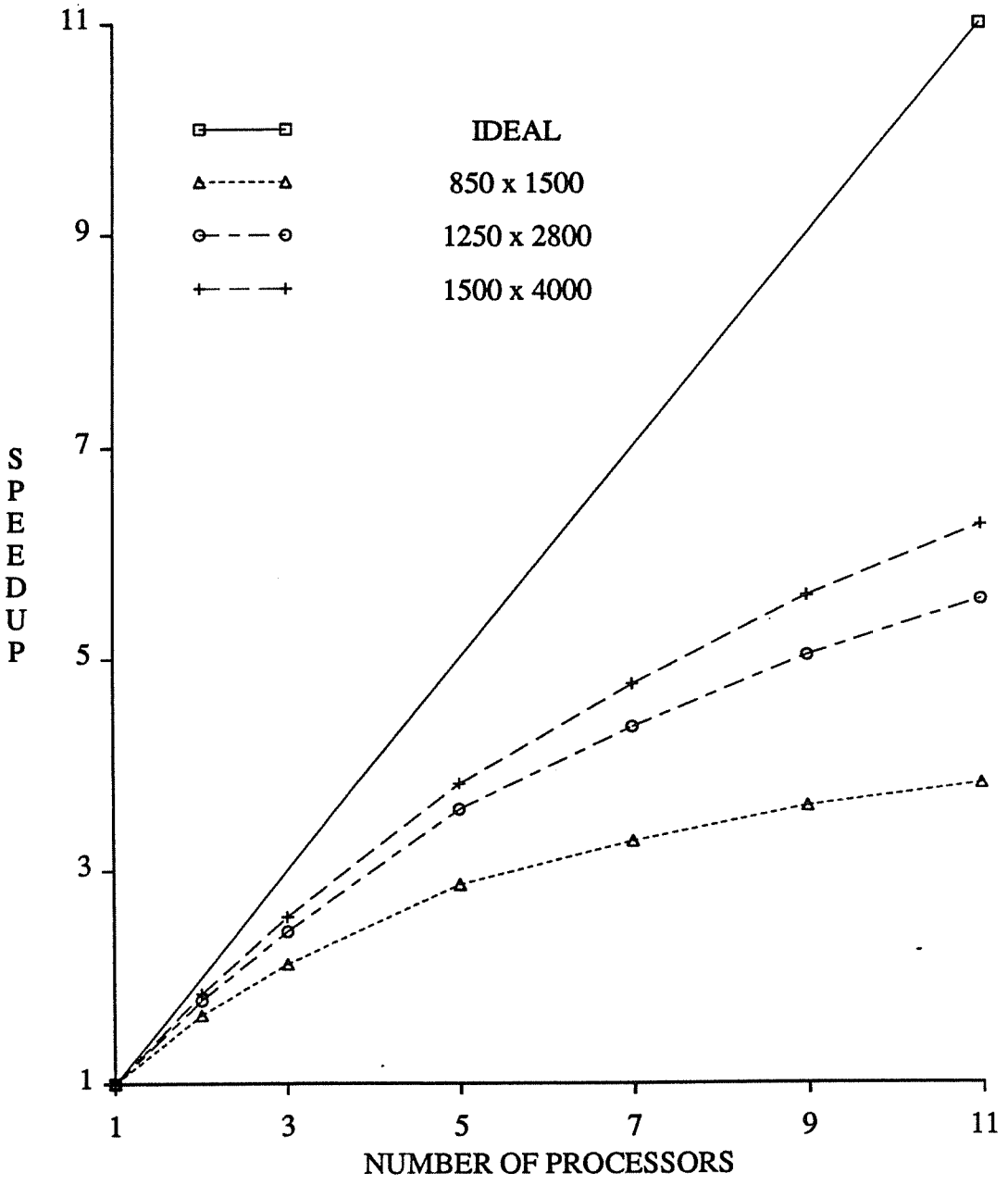

Figure 5.6 Speedup vs. Number of Processors (10 coupling constraints, 100 subproblems) matrix multiplication in serial

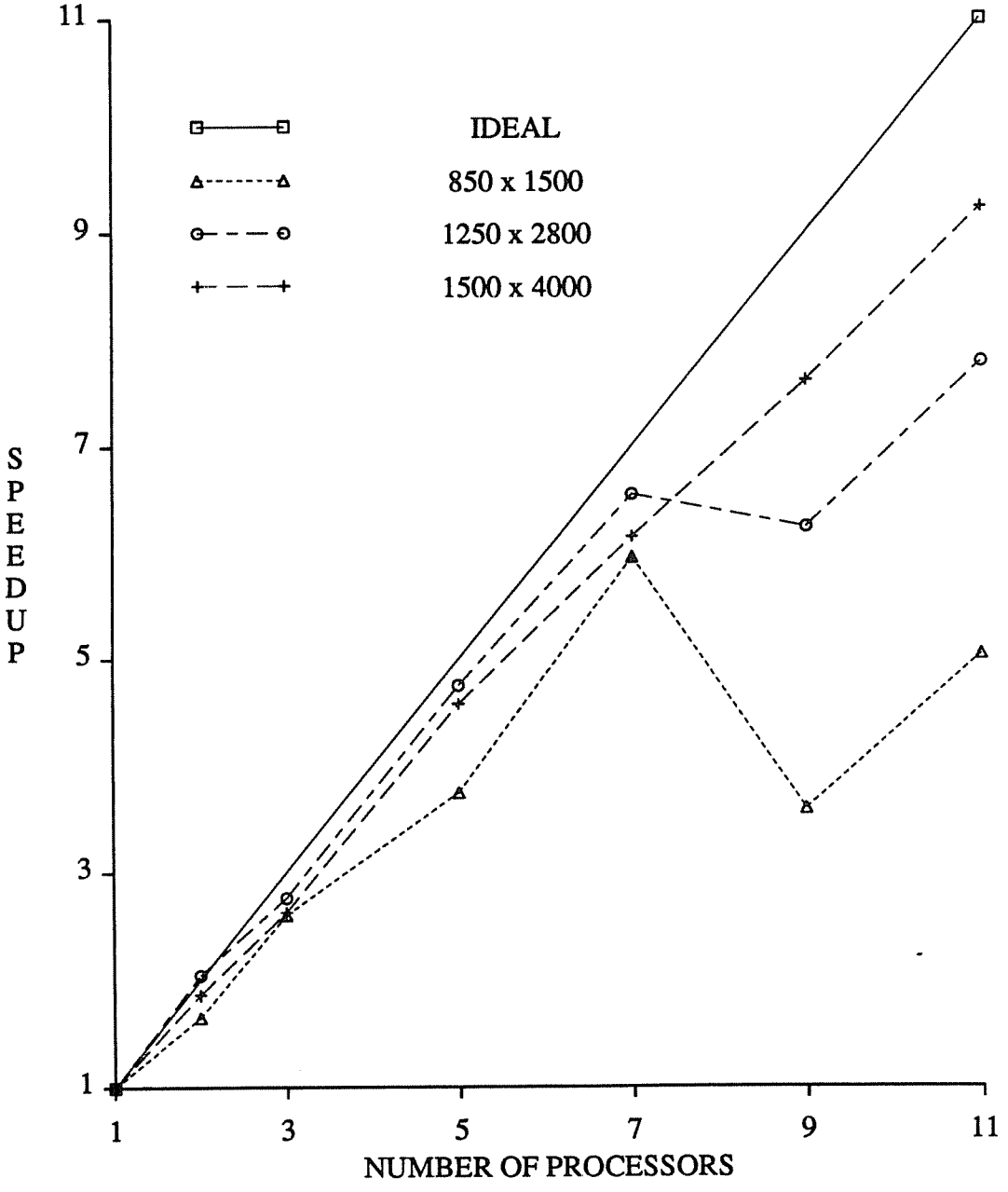

Figure 5.7 Speedup vs. Number of processors (10 coupling constraints, 100 subproblems) matrix multiplication in parallel

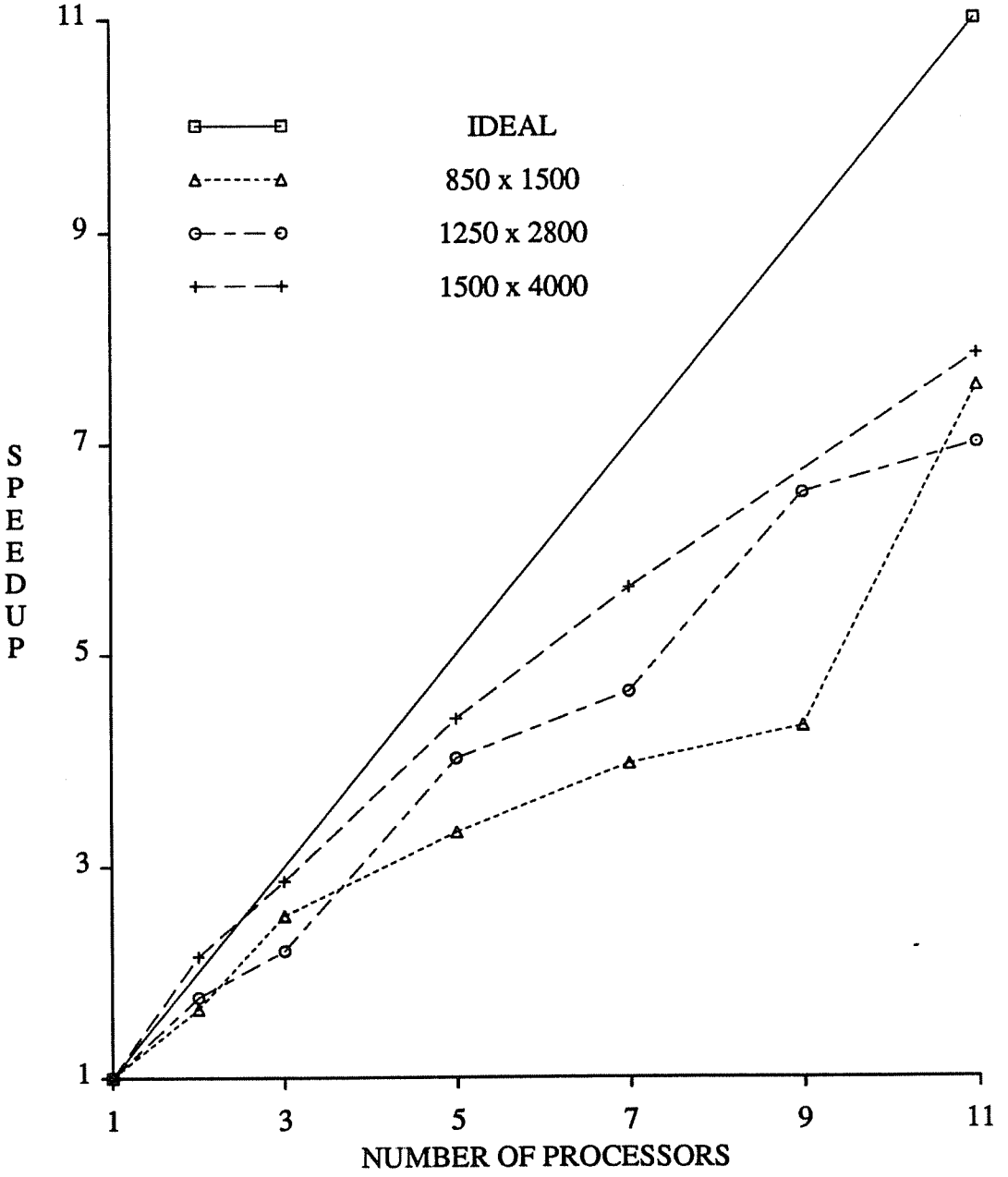

Figure 5.8 Speedup vs. Number of processors (20 coupling constraints, 100 subproblems) matrix multiplication in parallel

5.4.2 Case II

We have implemented the algorithm ALG 5.7 for uneven subproblems using both the algorithm LPT 5.4 and the algorithm MF 5.6 for scheduling subproblems. We did our tests for block-angular linear programming problems. Hereafter, we will refer to the two versions of the algorithm ALG 5.7 as the algorithm LPT and the algorithm MF, respectively.

Both the algorithms LPT 5.4 and MF 5.6 require knowledge of the solution time beforehand for assigning the subproblems to the different processors. Since we do not have any *a priori* knowledge of the solution time we decided to estimate it. To accomplish this we generated linear programming problems of different sizes, with five problems, in each size, to be solved by the routine ZX0LP. We first took the average of the time of the five random problems in each size. Then we did a regression fit for different problem sizes. For solution time, we obtained the following relation :

$$c\,m^{2.17}n^{0.89}$$

where m is the number of rows and n is the number of variables, and c is some constant. This gives us an estimate for the solution time for the subproblems. The reader should note that we solve the subproblems from scratch only the first time, and after that, we use parametric programming. So, the behavior of the solution time at subsequent dual point may be somewhat different, but we use the same estimate at the subsequent dual points also. Thus, we do a static scheduling of

the subproblems, i.e., we do not move the subproblems around to other processors at subsequent dual points, once they are allocated in the beginning.

The implemented algorithms were tried on two test problems. The first test problem one is of dimension 475×1000 with 10 coupling constraints and 30 subproblems. The second test problem is of dimension 400×1100 with 10 coupling constraints and 34 subproblems. The smallest numbers of constraints for the subproblems for both test problems one and two are 5, whereas the largest numbers of constraints are 40 and 39, respectively. The smallest numbers of variables for the subproblems for test problems one and two are 15 and 8, respectively, and the largest numbers of variables are 80 and 68, respectively. We tested the algorithms using 1, 2, 3 and 5 processors. We present the information about iterations (ITER), number of calls to SIMUL (NSIM) and time in Tables 5.7 and 5.9. The numbers in parenthesis indicate the time taken for the first call to SIMUL and the average of time taken for the subsequent calls to SIMUL. The difference in the number of iterations (and number of calls to SIMUL) between the methods (LPT and MF) is due to round off error; this happens as after the scheduling, the ordering of the subproblems in both the methods may be different and that may make a difference in the numerical calculation of the dual function value and the subgradient. This observation is similar to the observation we made in the previous section.

Speedup and efficiency are reported in Tables 5.8 and 5.10. The speedup vs. number of processors is plotted in Figures 5.9 and 5.10 for problems one and two,

respectively. From the results, we observe that the algorithm LPT gives better speedup (and efficiency) than the algorithm MF for these two test problems on the number of processors tried. This is clear for smaller number of processors, when we can clearly see the difference in total time, time for first call to SIMUL and the average of time taken for the subsequent calls to SIMUL. For five processors, the gaps seem to be narrowing. Under any circumstances, we have more than 50 % efficiency with both the methods for the problems and cases we tried. With larger test problems, we hope to obtain similar or better behavior.

	Using Algorithm LPT		Using Algorithm MF	
Number of processors	ITER/ NSIM	Time in sec	ITER/ NSIM	Time in sec
1	30/57	1367.86 (221.12, 20.30)	30/57	1367.86 (221.12, 20.30)
2	24/65	774.27 (115.58, 10.16)	24/62	830.99 (138.08, 11.23)
3	24/70	611.69 (88.47, 7.47)	23/67	613.49 (101.62, 7.64)
5	24/63	418.84 (66.70, 5.56)	28/72	452.46 (66.72, 5.31)

Table 5.7 *Test problem 1 : Output information*
Times in parenthesis are the time for the first call to SIMUL and the average of the subsequent calls to SIMUL

	Using Algorithm LPT		Using Algorithm MF	
Number of processors	Speedup	Efficiency	Speedup	Efficiency
2	1.767	0.884	1.646	0.823
3	2.236	0.745	2.230	0.743
5	3.265	0.653	3.023	0.605

Table 5.8 *Test problem 1 : Speedup and Efficiency*

Number of processors	Using Algorithm LPT ITER/ NSIM	Using Algorithm LPT Time in sec	Using Algorithm MF ITER/ NSIM	Using Algorithm MF Time in sec
1	26/81	889.76 (134.40, 9.33)	26/81	889.76 (134.40, 9.33)
2	23/71	489.78 (80.17, 5.74)	29/72	557.63 (94.16, 6.40)
3	28/68	365.59 (61.89, 4.40)	30/82	424.27 (66.35, 4.27)
5	30/74	292.64 (52.71, 3.13)	24/77	314.44 (53.35, 3.32)

Table 5.9 *Test problem 2 : Output information*
Times in parenthesis are the time for the first call to SIMUL and the average of the subsequent calls to SIMUL

Number of processors	Using Algorithm LPT Speedup	Using Algorithm LPT Efficiency	Using Algorithm MF Speedup	Using Algorithm MF Efficiency
2	1.767	0.884	1.646	0.823
3	2.236	0.745	2.230	0.743
5	3.265	0.653	3.023	0.605

Table 5.10 *Test problem 2 : Speedup and Efficiency*

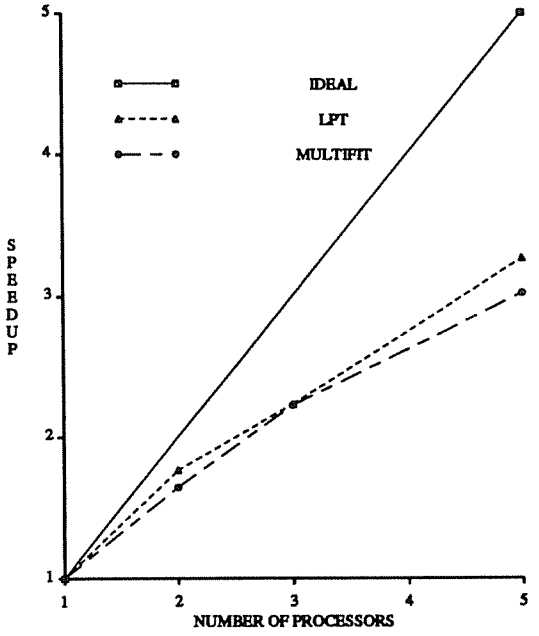

Figure 5.9 Speedup vs. Number of processors with scheduling by the algorithm LPT and the algorithm MF : test problem 1

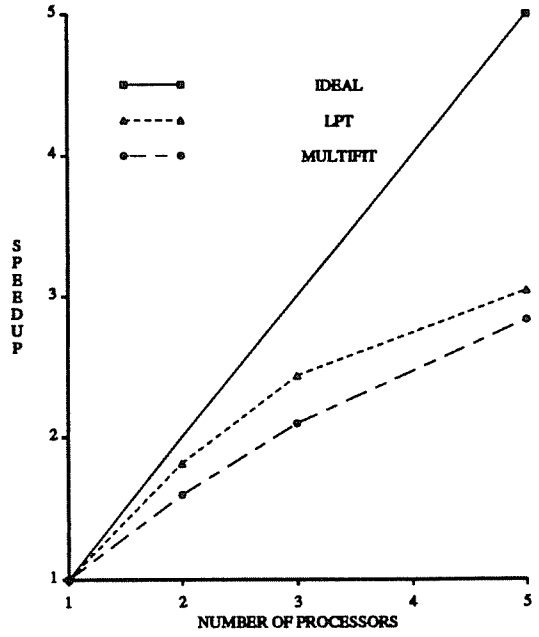

Figure 5.10 Speedup vs. Number of processors with scheduling by the algorithm LPT and the algorithm MF : test problem 2

5.5 Summary

In this chapter, we presented parallel algorithms for problem type (1.1) and their implementation and computational experience on the CRYSTAL multicomputer. For problems with subproblems of even size, we allocated subproblems equally among the processors, and observed that we can attain efficiency in the range of 70% - 80%. For problems with uneven subproblems, we identified the problem of allocation of the subproblems to different processors with the *independent task-scheduling* problem [Gar78]. We implemented the least processing time algorithm and the MULTIFIT algorithm for scheduling the subproblems. These algorithms require knowledge of solution time of subproblems beforehand. In our case, this is not available. Instead, we estimate the solution time. The implemented algorithms, then, give an efficiency of more than 60% on the test problems we tried, with the least processing time algorithm giving slightly better efficiency than the MULTIFIT algorithm.

Chapter 6

A DOUBLY-COUPLED LINEAR PROGRAM

In this chapter we propose two-stage decomposition methods for the doubly-coupled linear program (2.6). The problem (2.6) cannot be directly attacked by the decomposition technique described in Chapter 3 for the type of problem (1.1). Here, we propose two approaches where we apply the decomposition technique in two stages to the problem (2.6). For convenience, we display here the doubly-coupled linear program :

$$\min_{x_0,\ldots,x_N} \quad \langle c_0, x_0\rangle + \langle c_1, x_1\rangle + \cdots + \langle c_N, x_N\rangle$$

subject to

$$\begin{array}{lllll} D_1\, x_0 + B_1\, x_1 & & & & = b_1 \\ D_2\, x_0 + & B_2\, x_2 & & & = b_2 \\ \vdots & & \ddots & & \vdots \\ D_N\, x_0 + & & & B_N\, x_N & = b_N \\ A_0\, x_0 + A_1\, x_1 + & \cdots & & + A_N\, x_N & = a \end{array} \tag{6.1}$$

$$x_i \geq 0, i = 0, 1, \ldots, N,$$

where $B_i \in \mathbb{R}^{m_i \times n_i}$, $D_i \in \mathbb{R}^{m_i \times n_0}$, $A_i \in \mathbb{R}^{m \times n_i}$, for $i =, 0, 1, \ldots, N$, and $a \in \mathbb{R}^m$, $b_i \in \mathbb{R}^{m_i}$, $i = 1, \ldots, N$ with $x_i \in \mathbb{R}^{n_i}, i = 0, 1, \ldots, N$.

We found only two references which deal with solution methods for the problem (6.1). They are Ritter's algorithm [Rit67], and the algorithm by Hartman and Lasdon [Har70].

We present here the two approaches.

6.1 The first approach

First we define the perturbation function, $F(x, p)$, as

$$F(x,p) = \begin{cases} \sum_{i=0}^{N} \langle c_i, x_i \rangle, & \text{if } \begin{cases} a - \sum_{i=0}^{N} A_i\, x_i = p, \\ D_i x_0 + B_i x_i = b_i,\ i = 1, \ldots, N, \\ x_i \geq 0,\ i = 0, 1, \ldots, N; \end{cases} \\ \infty, & \text{otherwise,} \end{cases}$$

where $x = (x_0, \ldots, x_N)$. The Lagrangian is

$$\begin{aligned} L(x,y) &= \inf_p \{\langle y, p \rangle + F(x,p)\} \\ &= \begin{cases} \sum_{i=0}^{N} \langle c_i, x_i \rangle + \langle y, a - \sum_{i=0}^{N} A_i x_i \rangle, & \text{if } \begin{cases} D_i x_0 + B_i x_i = b_i, \\ \quad i = 1, \ldots, N \\ x_i \geq 0,\ i = 0, 1, \ldots, N; \end{cases} \\ \infty, & \text{otherwise.} \end{cases} \end{aligned}$$

Then, the dual problem is

$$\max_y \ g(y) \tag{6.2}$$

where

$$\begin{aligned} g(y) &= \inf_x \ L(x,y) \\ &= \langle y, a \rangle + \inf_x \left\{ \sum_{i=0}^{N} \langle c_i - y A_i, x_i \rangle \;\middle|\; \begin{array}{r} D_i x_0 + B_i x_i = b_i, i = 1, \ldots, N \\ x_i \geq 0, i = 0, 1, \ldots, N \end{array} \right\}. \end{aligned}$$

Note that g is nonsmooth concave. We will denote the second part by $\tilde{g}(y)$, i.e.,

$$\tilde{g}(y) := \inf \left\{ \sum_{i=0}^{N} \langle c_i - yA_i, x_i \rangle \;\middle|\; \begin{array}{c} D_i x_0 + B_i x_i = b_i, i = 1, \ldots, N \\ x_i \geq 0, i = 0, 1, \ldots, N \end{array} \right\}. \tag{6.3}$$

For given y, the problem (6.3) is the following structured linear program :

$$\inf_x \langle c_0 - yA_0, x_0 \rangle + \cdots + \langle c_N - yA_N, x_N \rangle$$

subject to

$$\begin{array}{lll} D_1\, x_0 + B_1\, x_1 & & = b_1 \\ D_2\, x_0 + & B_2\, x_2 & = b_1 \\ \vdots & \ddots & \vdots \\ D_N\, x_0 + & & B_N\, x_N = b_1 \end{array}$$

$$x_i \geq 0, i = 0, 1, \ldots, N.$$

Its usual linear programming dual is

$$\max_{v_1, \ldots, v_N} \quad \langle b_1, v_1 \rangle + \cdots + \langle b_N, v_N \rangle$$

subject to

$$\begin{array}{lll} v_1\, B_1 & & \leq c_1 - yA_1 \\ & v_2\, B_2 & \leq c_2 - yA_2 \\ & \ddots & \vdots \\ & & v_N\, B_N \leq c_N - yA_N \\ v_1\, D_1 + & \cdots & + v_N\, D_N \leq c_0 - yA_0 \end{array} \tag{6.4}$$

$$v_i\, (i = 1, \ldots, N) \quad \text{unrestricted.}$$

For now, we consider y to be temporarily fixed at $\hat{y}$. Then the above LP (6.4) looks very close to the problem (4.1), and so, we should be able to apply the decomposition technique to this problem. First, we rewrite the above problem as a minimization problem :

$$\min_{v_1,\ldots,v_N} \quad -\langle b_1, v_1\rangle - \cdots - \langle b_N, v_N\rangle$$

subject to

$$\begin{aligned}
v_1\ B_1 &\leq c_1 - \hat{y}A_1 \\
v_2\ B_2 &\leq c_2 - \hat{y}A_2 \\
\ddots &\quad \vdots \\
v_N\ B_N &\leq c_N - \hat{y}A_N \\
v_1\ D_1 + \quad \cdots \quad + v_N\, D_N &\leq c_0 - \hat{y}A_0 \\
u_i\,(i = 1,\ldots,N) &\quad \text{unrestricted.}
\end{aligned} \tag{6.5}$$

For simplicity, let

$$\hat{c}_i := c_i - \hat{y}A_i, \quad i = 0, 1, \ldots, N.$$

Now we define the perturbation function for this problem as

$$\mathcal{F}(v, q, \hat{y}) := \begin{cases} -\sum_{i=1}^{N} \langle b_i, v_i\rangle, & \text{if } \begin{cases} B_i^T v_i \leq \hat{c}_i, i = 1,\ldots,N \\ -\hat{c}_0 + \sum_{i=1}^{N} D_i^T v_i \leq q \end{cases} \\ +\infty, & \text{else.} \end{cases}$$

So the Lagrangian for this problem is

$$\begin{aligned}
\mathcal{L}(v, u, \hat{y}) &= \inf_q \{\langle u, q\rangle + \mathcal{F}(v, q, \hat{y})\} \\
&= \begin{cases} -\sum_{i=1}^{N} \langle b_i, v_i\rangle + \langle u, -\hat{c}_0 + \sum_{i=1}^{N} D_i^T v_i\rangle, & \text{if } \left| \begin{array}{r} u \geq 0 \\ B_i^T v_i \leq \hat{c}_i, i = 1,\ldots,N \end{array} \right. \\ -\infty, & \text{if } \left| \begin{array}{r} u \not\geq 0 \\ B_i^T v_i \leq \hat{c}_i, i = 1,\ldots,N \end{array} \right. \\ +\infty, & \text{if } B_i^T v_i \not\leq \hat{c}_i. \end{cases}
\end{aligned}$$

So, for this fixed $\hat{y}$, we get the following dual problem of the problem (6.5) :

$$\max_{u \geq 0} \ \tilde{t}(u, \hat{y})$$

where

$$\tilde{t}(u, \hat{y}) = \begin{cases} -\langle \hat{c}_0, u \rangle + \sum_{i=1}^{N} \inf_{v_i} \ \{ \langle D_i u - b_i, v_i \rangle \mid B_i^T v_i \leq \hat{c}_i \} & \text{if } u \geq 0; \\ -\infty, & \text{else.} \end{cases}$$

Observe that u is the original variable x_0. So, writing the above problem as the dual problem of the problem (6.4) and with x_0 substituted in place of u, we have

$$-\max_{x_0 \geq 0} \ \tilde{t}(x_0, \hat{y}) = \min_{x_0 \geq 0} \ \{ -\tilde{t}(x_0, \hat{y}) \} := \min_{x_0 \geq 0} \ t(x_0, \hat{y})$$

and

$$t(x_0, \hat{y}) = \begin{cases} \langle \hat{c}_0, x_0 \rangle + \sum_{i=1}^{N} \sup_{v_i} \ \{ \langle b_i - D_i x_0, v_i \rangle \mid B_i^T v_i \leq \hat{c}_i \}, & \text{if } x_0 \geq 0; \\ +\infty, & \text{else.} \end{cases} \tag{6.6}$$

The function $t(\cdot, \hat{y})$ is a nonsmooth convex function in x_0, and to solve the nonsmooth problem

$$\min_{x_0 \geq 0} \ t(x_0, \hat{y}) \tag{6.7}$$

one can take the nonsmooth exact penalty function approach [Pie69] to transform (6.7) to the unconstrained problem

$$\min_{x_0} \ \{ \ t(x_0, \hat{y}) + \alpha \parallel (-x_0)_+ \parallel_1 \ \}, \tag{6.8}$$

where α is a large positive number. To compute the function value and a subdifferential for the objective function in the problem (6.8), one needs to solve the LP subproblems :

$$\begin{aligned} & \sup_{v_i} \ \langle b_i - D_i x_0, v_i \rangle \\ \text{subject to} \quad & B_i^T \, v_i \leq c_i - \hat{y} A_i, \end{aligned} \tag{6.9}$$

for $i = 1, \ldots, N$. Note that the dual variables of the problem $(6.9)_i$ correspond to the variables $x_1, \ldots, x_N$ of the problem (6.1). Under a certain assumption (similar to the one given in § 3.1), a subgradient for the objective function in (6.8) at x_0 is given by

$$c_0 - \hat{y}A_0 - \sum_{i=1}^{N} v_i D_i + \alpha(-x_0)_+,$$

where v_i is the computed solution of the LP subproblem $(6.9)_i$ for given x_0 and $\hat{y}$.

Suppose we are mainly interested in obtaining a primal optimal solution of the problem (6.1). In that case, we do not need the 'primal' optimal solution $(v_1, \ldots, v_N)$ of the problem (6.5). Thus, we can directly apply the bundle method of Lemaréchal *et al* [Lem81] to the problem (6.8), instead of using the bundle-based decomposition algorithm of Chapter 3. On the other hand if we also want the 'primal' optimal solution $(v_1, \ldots, v_N)$ of the problem (6.5), we need to apply the bundle-based decomposition algorithm of Chapter 3.

Once the problem (6.8), or equivalently (6.7), is solved for fixed y, we have to solve the outer problem (6.2), where

$$g(y) = \langle y, a \rangle + \min_{x_0 \geq 0} \; t(x_0, y).$$

As noted before, $g(\cdot)$ is nonsmooth concave. A subgradient for $g(\cdot)$ at y can be given as

$$a - A_0\, x_0 - \sum_{i=1}^{N} A_i\, x_i,$$

where x_0 is the computed solution of the problem (6.7), and $x_1, \ldots, x_N$ are the dual solutions of the problems $(6.9)_i$ for this x_0.

Finally, we apply the bundle-based decomposition algorithm of Chapter 3 to the dual problem (6.2). We present the approach described above for solving the problem (6.1) in a compact way as the following algorithm :

Algorithm ALG 6.1

Step 1. *Outer loop problem*

Start with y^1. Set $j \leftarrow 1$.

Step 2. *Inner loop problem*

For given y^j, solve the problem (6.7) by the bundle method.

Step 3.

Check if y^j is optimal. If so, stop.

If not, update y^j to y^{j+1} by the bundle-based decomposition algorithm ALG 3.1.

$j \leftarrow j + 1$, and go to Step 2.

Remarks

(1) In Step 2, one could solve the problem (6.7) by the bundle-based decomposition method of Chapter 3 if one is interested in obtaining the solutions $(v_1, \ldots, v_N)$ of (6.4) also.

(2) Note that for given y and x_0, one needs to solve the LP subproblems $(6.9)_i$ from scratch the very first time. After that for the inner loop problem, the objective changes as it is dependent on x_0. Then, when we go to the outer loop, and y is updated, the right hand side of the problems $(6.9)_i$ changes. So, it is desirable to have an LP code that can solve the problems $(6.9)_i$ by

parametric programming both for the objective function and for the right hand side.

6.2 The second approach

Here, we give another approach to solve the problem (6.1). We can rewrite the problem (6.1) as

$$\min_{x_0 \geq 0} \{ \langle c_0, x_0 \rangle + r(x_0) \}, \tag{6.10}$$

where $r(x_0)$ is defined by the value of the following linear programming problem :

$$\min_{x_1,\ldots,x_N} \quad \langle c_1, x_1 \rangle + \cdots + \langle c_N, x_N \rangle$$

subject to

$$\begin{aligned}
B_1\, x_1 \qquad\qquad\qquad\qquad &= b_1 - D_1 x_0 \\
B_2\, x_2 \qquad\qquad\qquad\qquad &= b_2 - D_2 x_0 \\
\ddots \qquad\qquad\qquad\quad &\vdots \qquad\qquad\qquad (6.11) \\
B_N\, x_N &= b_N - D_N x_0 \\
A_1\, x_1 + \quad \cdots \quad + A_N\, x_N &= a - A_0 x_0 \\
x_i \geq 0, i = 1, \ldots, N.
\end{aligned}$$

For given x_0, the problem (6.11) is exactly the same as the problem (4.1). Thus, we can apply the bundle-based decomposition algorithm ALG 4.1 to the problem

(6.11). To solve the outer loop problem (6.10), we can again take the nonsmooth exact penalty function approach [Pie69] and solve the unconstrained problem

$$\min_{x_0} \{ \langle c_0, x_0 \rangle + r(x_0) + \alpha \parallel (-x_0)_+ \parallel_1 \} \tag{6.12}$$

for some large positive number α. The problem (6.12) can be solved by the usual bundle method [Lem81]. A subgradient of the objective function in the problem (6.12) is given by

$$c_0 - y\, A_0 - \sum_{i=1}^{N} v_i\, D_i + \alpha(-x_0)_+,$$

where y is the dual solution of the dual problem, similar to (3.2), of the problem (6.11), and v_i is a dual solution of the LP subproblem

$$\min_{x_i} \langle c_i - yA_i, x_i \rangle$$

$$\text{subject to} \quad B_i\, x_i = b_i - D_i x_0, \quad x_i \geq 0.$$

Here the outer loop problem is to solve the problem (6.12), or equivalently, to solve the problem (6.10). We state the above approach in the following algorithmic form :

Algorithm ALG 6.2

Step 1. *Outer loop problem*

Start with x_0^1. Let $j \leftarrow 1$.

Step 2. *Inner loop problem*

For given x_o^j, solve the problem (6.11) by the bundle-based decomposition algorithm ALG 4.1.

Step 3.

Check if x_0^j is optimal. If so, stop.

If not, update x_0^j to x_0^{j+1} by the bundle method [Lem81].

$j \leftarrow j+1$, and go to Step 2.

Here, we have discussed alternative two-stage decomposition methods to solve the problem (6.1). The essential difference between the above two approaches is the order of the outer loop and inner loop problems. In the first approach, the outer variable is y and the inner variable is x_0, whereas in the second approach the outer variable is x_0 and the inner variable is y (the dual variables of the problem (6.11)). In this sense, the two approaches are symmetric and these algorithms can be considered as dual to each other.

Appendix

Problem Name	Size of whole problem	EPS/ DF1	Objective value : primal/dual	ITER/ EVAL	Gradient Accuracy/ Relative	Time in Min.	*Stopping Rule*
1a3	350 × 500	0.1	-1729.0941	31	0.22×10^{-11}	3.19	*Normal*
		1.5	-1729.1547	75	na	0.47	*End*
						0.03	
1a3	350 × 500	0.5	-1728.8947	24	0.38×10^{-12}	2.62	*Normal*
		1.5	-1729.3947	60	na	0.44	*End*
						0.03	
1a3	350 × 500	0.75	-1728.7209	22	0.31×10^{-12}	2.59	*Normal*
		1.5	-1729.4708	59	na	0.44	*End*
						0.03	
1b1	350 × 500*	0.1	-1248.5427	60	0.363×10^{1}	2.92	*Normal*
		100	-1248.0388	136	0.335×10^{-1}	0.07	*End*
						0.02	
1b1	350 × 500*	1.0	-1247.5926	47	0.302×10^{-13}	2.15	*Normal*
		100	-1248.5925	105	0.985×10^{-15}	0.07	*End*
						0.02	
1b2	350 × 500*	1.0	-1640.7158	62	0.109×10^{1}	4.13	*Max15*
		1000	-1641.5603	157	0.289×10^{-1}	0.20	
						0.02	
1b2	350 × 500*	5.0	-1638.8857	63	0.171×10^{-13}	3.83	*Normal*
		1000	-1643.8857	125	0.795×10^{-15}	0.19	*End*
						0.02	

Table A.1 *Output information from BUNDECOMP*

* – *DX is set at* 10^{-6}*; rest are set at* 10^{-7}.

(Continued on the next page)

Problem Name	Size of whole problem	EPS/ DF1	Objective value : primal/dual	ITER/ EVAL	Gradient Accuracy/ Relative	Time in Min.	*Stopping Rule*
1b3	350 × 500*	1.0	-769.4664	69	0.686	6.24	*Max15*
		100	-770.2497	167	0.455×10^{-1}	0.40	
						0.03	
1b3	350 × 500*	5.0	-768.0163	54	0.114×10^{1}	5.15	*Max15*
		100	-772.4489	134	0.137	0.40	
						0.03	
1b3	350 × 500	10.0	-764.3880	57	0.215×10^{-13}	5.24	*Normal*
		10	-774.0837	133	0.116×10^{-14}	0.40	*End*
						0.03	
1b4	350 × 500	10.0	-1513.8084	48	0.139×10^{1}	17.13	*Max15*
		10	-1521.7147	114	0.661×10^{-1}	2.43	
						0.13	
1b4	350 × 500	20.0	-1505.9762	39	0.399×10^{-2}	14.73	*Normal*
		1000	-1525.9759	76	0.573×10^{-3}	2.43	*End*
						0.16	
1c2	350 × 500*	1.0	-1693.9730	66	0.414×10^{1}	5.44	*Max15*
		10^5	-1694.4614	155	0.121	0.19	
						0.03	
1c2	350 × 500*	20.0	-1679.6585	103	0.744×10^{-2}	6.85	*Normal*
		10^5	-1699.6589	182	0.987×10^{-4}	0.19	*End*
						0.03	
1d1	350 × 500	1.0	-1204.3195	129	0.368×10^{1}	9.53	*Max15*
		10^5	-1205.8113	265	0.582	0.08	
						0.03	
1d1	350 × 500	5.0	-1200.6983	179	0.129×10^{-12}	12.09	*Normal*
		10^5	-1205.6259	307	0.520×10^{-13}	0.08	
						0.03	

Table A.1 *Output information from BUNDECOMP*

* – *DX is set at* 10^{-6}*; rest are set at* 10^{-7}.

(Continued on the next page)

Problem Name	Size of whole problem	EPS/ DF1	Objective value : primal/dual	ITER/ EVAL	Gradient Accuracy/ Relative	Time in Min.	*Stopping Rule*
1d2	350 × 500	1.0	-1320.2651	129	0.160×10^{1}	11.08	*Max15*
		1000	-1320.7020	262	0.146	0.19	
						0.03	
1d2	350 × 500	20.0	-1313.8677	207	0.179×10^{-1}	14.89	*Normal*
		1000	-1332.8708	309	0.982×10^{-3}	0.19	*End*
						0.03	
1e1	350 × 500	50.0	-1244.0764	185	0.136×10^{-1}	15.02	*Normal*
		1000	-1293.9884	270	0.870×10^{-3}	0.07	*End*
						0.03	
1e2	350 × 500	50.0	-1364.2005	131	0.115×10^{-1}	11.07	*Normal*
		1000	-1413.7078	190	0.581×10^{-3}	0.18	*End*
						0.03	
1e3	350 × 500	20.0	-1554.1824	147	0.213×10^{1}	15.51	*Max15*
		1000	-1573.3340	257	0.137	0.37	
						0.04	
2a3	850 × 1500	1.5	-7244.7490	25	$0.2.600 \times 10^{-6}$	40.47	*Normal*
		2.0	-7246.2646	68	na	7.87	
						0.49	
2a3	850 × 1500	1.5	-7244.7490	25	0.260×10^{-5}	40.47	*Normal*
		2.0	-7246.2646	68	na	7.87	*End*
						0.49	
2a4	850 × 1500*	1.0	-6950.5272	24	0.705×10^{-14}	362.46	*Normal*
		100	-6951.5269	50	0.106×10^{-15}	66.95	*End*
						6.03	
2a4	850 × 1500*	1.0	-6947.6271	20	0.126×10^{-13}	355.87	*Normal*
		100	-6952.6718	43	0.251×10^{-15}	66.83	*End*
						6.88	
2b1	850 × 1500	1.0	-6768.7397	57	0.300×10^{-5}	8.39	*Normal*
		1.5	-6769.7480	107	na	0.67	*End*
						0.07	

Table A.1 *Output information from BUNDECOMP*

* – *DX is set at* 10^{-6}*; rest are set at* 10^{-7}.

(Continued on the next page)

Problem Name	Size of whole problem	EPS/ DF1	Objective value : primal/dual	ITER/ EVAL	Gradient Accuracy/ Relative	Time in Min.	*Stopping Rule*
2b2	850 × 1500*	1.0	-7039.9266	46	0.552×10^{-14}	27.15	*Normal*
		1000	-7040.9263	101	0.604×10^{-16}	3.47	*End*
						0.23	
2b2	850 × 1500*	5.0	-7036.8860	39	0.703	24.27	*Dxmin*
		1000	-7042.0010	78	0.681×10^{-2}	3.39	
						0.27	
2b3	850 × 1500	1.0	-6862.8493	31	0.675×10^{1}	59.86	*Max15*
		10^5	-6864.4961	86	0.180×10^{1}	8.17	
						0.62	
2b3	850 × 1500	5.0	-6860.7105	48	0.123×10^{1}	62.98	*Max15*
		10^5	-6865.7148	106	0.302	8.13	
						0.52	
2b4	850 × 1500	1.0	-5769.4902	37	0.129×10^{1}	407.93	*Max15*
		100	-5770.4287	84	0.797×10^{-1}	66.96	
						4.11	
2b4	850 × 1500	5.0	-5765.1557	24	0.703×10^{1}	392.93	*Max15*
		100	-5771.1528	61	0.435	66.52	
						5.44	
6a1	4000 × 10000	0.5	-68400.6086	22	0.105	401.12	*Max15*
		10^4	-68401.1094	66	0.801×10^{-4}	80.57	
						4.93	

Table A.1 *Output information from BUNDECOMP*
* – *DX is set at* 10^{-6}*; rest are set at* 10^{-7}.

Bibliography

[Adl73] Adler, I. and A. Ülkücü, On the number of iterations in Dantzig-Wolfe decomposition, in *Decomposition of Large Scale System*, ed. D.M. Himmelblau, North-Holland, Amsterdam, 1973.

[Bar69] Bartels, R. H., and G. H. Golub, The simplex method of linear programming using LU decomposition, *Comm. ACM*, **12**, 266-268 & 275-278 (1969) .

[Bea65] Beale, E. M. L., P. A. B. Hughes, and R. E. Small, Experiences in using a Decomposition Program, *Computer Journal*, **8**, 13-18 (1965).

[Cha86] Chang, M. D., *A parallel primal simplex variant for generalized networks*, Ph. D. dissertation, Department of General Business, University of Texas at Austin, Austin, August 1986.

[Che87] Chen, R.-J., *Parallel algorithms for a class of convex optimization problems*, Ph. D. dissertation, Computer Sciences Dept, University of Wisconsin–Madison, 1987.

[Cof78] Coffman, E. G., M. R. Garey, and D. S. Johnson, An application of bin packing to multiprocessor scheduling, *SIAM J. Computing*, **7**, 1-17 (1978).

[Dan61b] Dantzig, G. B. and A. Madansky, On the solution of two-stage linear programs under uncertainty, *Proc. Fourth Berkeley Symposium on Mathematical Statistics and Probability, Vol. 1*, University of California Press, Berkeley, 165-176 (1961).

[Dan67] Dantzig, G. B., and R. M. Van Slyke, Generalized upper bounded techniques for linear programming, *J. Comp. System Sci.*, **1**, 213-226 (1967).

[Dan60] Dantzig, G. B., and P. Wolfe, Decomposition principle for linear programs, *Operations Research*, **8**, 101-111 (1960).

[Dan61a] Dantzig, G. B., and P. Wolfe, The decomposition algorithm for linear programs, *Econometrica*, **29**, 767-778 (1961).

[DeL85] De Leone, R., Sequential Overrelaxation for large sparse linear programs, Internal Report, CRAI, Italy, June 1985.

[DeL87] De Leone, R. and O. L. Mangasarian, Serial and parallel solution of large-scale linear programs by augmented Lagrangian successive overrelaxation, Techincal Report # 701, Computer Sciences Department, University of Wisconsin–Madison, June 1987.

[DeW84] DeWitt, D. J., R. Finkel and M. Solomon, The CRYSTAL multicomputer: design and implementation experience, Technical Report # 553, Computer Sciences Department, University of Wisconsin–Madison, September, 1984.

[Dir79] Dirickx, Y. M. I., and L. P. Jennergren, *Systems Analysis by Multilevel Methods : With Applications to Economics and Management*, Wiley, Chichester, England, 1979.

[Erm83] Ermoliev, Y., Stochastic quasigradient methods and their application to systems optimization, *Stochastics*, **9**, 1-36 (1983).

[Fei85a] Feijoo, B., *Piecewise-linear approximation methods and parallel algorithms in optimization*, Ph. D. disseration, Technical Report # 598, Computer Sciences Department, University of Wisconsin–Madison, May 1985.

[Fei85b] Feijoo, B and R. R. Meyer, Optimization on the Crystal multicomputer, in *Computing 85*, eds. G. Bucci and G. Valle, Elsevier Science Publishers, 1985.

[Fly66] Flynn, M. J., Very high-speedy computing systems, in *Proceedings of the IEEE*, **54**, # 12, 1901-1909 (1966).

[Fri86] Friesen, D. K. and M. A. Langston, Evaluation of a MULTIFIT-based scheduling algorithm, *Journal of Algorithms*, **7**, 35-59 (1986).

[For72] Forrest, J. J. H. and J. A. Tomlin, Updating triangular factors of the basis to maintain sparsity in the product-form simplex method, *Mathematical Programming*, **2**, 263-278 (1972).

[Gar78] Garey, M. R., R. L. Graham, and D. S. Johnson, Performance guarantees for scheduling algorithms, *Operations Research*, **26**, 3-21 (1978).

[Gil81] Gill, P. E., W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, New York, 1981.

[Gra69] Graham, R. L., Bounds on Multiprocessing timing Anomalies, *SIAM J. Appl. Math.*, **17**, 416-429 (1969).

[Ha80] Ha, C. D., *Decomposition methods for structured convex programming*, Ph. D. dissertation, Department of Industrial Engineering, University of Wisconsin–Madison, 1980.

[Har70] Hartman, J. K. and L. S. Lasdon, A generalized upper bounding method for doubly coupled linear programs, *Naval Res. Log. Quart.*, **17**, 411-429 (1970).

[Hel74] Held, M., P. Wolfe, and H. Crowder, Validation of subgradient optimization, *Mathematical Programming*, **6**, 62-88 (1974).

[Ho] Ho, J. K., and É. Loute, DECOMP User's Guide, unpublished manuscript.

[Ho81] Ho, J., and É. Loute, An advanced implementation of the Dantzig-Wolfe decomposition algorithm for linear programming, *Mathematical Programming*, **20**, 303-326 (1981).

[Ho83] Ho, J., and É. Loute, Computational experience with advanced implementation of decomposition algorithms for linear programming, *Mathematical Programming*, **27**, 283-290 (1983).

[Ho84] Ho, J., and É. Loute, Computational aspects of dynamico : a model of trade and development in the world economy, *Revue Française d'Automatique, Informatique et Recherche opérationnelle*, **18**, 403-414 (1984).

[Imsl84] IMSL User's manual, Edition 9.2, International Mathematical and Statistical Library, Houston, Texas, November 1984.

[Kal79] Kall, P., Computational methods for solving two-stage stochastic linear programming problems, *Z. Angew. Math. Phys.*, **30**, 261-271 (1979).

[Ken80] Kennington, J. L. and R. V. Helgason, *Algorithms for Network Programming*, John Wiley & Sons, New York, 1980.

[Kut73] Kutcher, G. P. , On the decomposition price endogenous models, in *Multi-level Planning : Case Studies in Mexico*, eds. L. M. Goreux, A. S. Manne, North-Holland, Amsterdam, 1973.

[Las78] Lasdon, L. S., Large-Scale programming, in *Handbook of Operations Research*, eds. Moder and Elmagharby, Van-Nostrand, 266-294, 1978.

[Lem75] Lemaréchal, C., An extension of Davidon's methods to nondifferentiable problems, *Mathematical Progamming 3*, North Holand, 95-109, 1975.

[Lem78a] Lemaréchal, C., Bundle methods in nonsmooth optimization, in *Nonsmooth Optimization*, eds. C. Lemaréchal and R. Mifflin, Pergamon Press, Oxford, 79-102, 1978.

[Lem78b] Lemaréchal, C., A view of Line-searches, in *Nonsmooth Optimization*, eds. C. Lemaréchal and R. Mifflin, Pergamon Press, Oxford, 59-78, 1978.

[Lem86a] Lemaréchal, C., Constructing bundle methods for convex optimization, in *FERMAT Days '85 : Mathematics for Optimization*, ed. J. B. Hiriart-Urruty, Elsevier Science Publishers B. V. (North-Holland), 1986.

[Lem86b] Lemaréchal, C., Private communication, 1986.

[Lem86c] Lemaréchal, C., Private communication, 1986.

[Lem85] Lemaréchal, C., and M.-C. Bancora Imbert, Le module M1FC1, preprint: INRIA, B.P. 105, Le Chesnay, France, March 1985.

[Lem81] Lemaréchal, C., J. J. Strodiot and A. Bihain, On a bundle algorithm for Nonsmooth Optimization, in *Nonlinear Programming 4*, eds. O.

L. Mangasarian, R. R. Meyer and S. M. Robinson, Academic Press, 245-282, 1981.

[Leu82] Leung, J. Y.-T., On scheduling independent tasks with restricted execution time, *Operations Research*, **30**, 163-171 (1982).

[Man84] Mangasarian, O. L., Sparsity-preserving SOR algorithms for separable quadratic and linear programming, *Comput.& Ops. Res.*, **11**, 105-112, 1984.

[Man86] Mangasarian, O. L. and R. De Leone, Parallel successive overrelaxation method for symmetric linear complementarity problems and linear programs, Technical Report # , Computer Sciences Department, University of Wisconsin–Madison, 1986.

[Man79] Mangasarian, O. L., and R. R. Meyer, Nonlinear perturbation of linear programs, *SIAM J. Control Opt.*, **17**, 745-752, 1979.

[Mar75] Marsten, R. E., W. W. Hogan and J. W. Blankenship, The boxstep method for large-scale optimization, *Operations Research*, **23**, 389-405 (1975).

[Mif79] Mifflin, R., A stable method for solving certain constrained least-squares problems, *Mathematical Programming*, **16**, 141-158 (1979).

[Mur83] Murtagh, B. A. and M. A. Sauders, MINOS 5.0 user's guide, Technical Report # SOL 83-20, System Optimization Laboratory, Department of Operations Research, Stanford University, December 1983.

[Phi86] Phillips, A. T. and J. B. Rosen, Multitasking mathematical programming algorithms, Technical Report # 86-10, Computer Science Department, University of Minnesota, Minneapolis, June 1986.

[Pie69] Pietrzykowski, T., An exact potential method for constrained maxima, *SIAM J. Numer. Anal.*, **6**, 299-304 (1969).

[Pol78] Poljak, B. T., Subgradient method : a survey of Soviet research, in *Nonsmooth Optimization*, eds. C. Lemaréchal and R. Mifflin, Pergamon Press, Oxford, 5-28 (1978).

[Rit67] Ritter, K., A decomposition method for linear programming problems with coupling constraints and variables, Tech. Rep. 739, Mathematics Research Center, University of Wisconsin-Madison, 1967.

[Rob78] Robinson, S. M., *Lectures on Convex Analysis*, unpublished notes, 1978.

[Rob86] Robinson, S. M. Bundle-based decomposition: Description and preliminary results, in *System Modelling and Optimization*, eds. A. Prekopa, J. Szelezsan ahd B. Strazicky, (Lecture Notes in Control and Information Sciences, Vol. 84), Springer-Verlag, Berlin, 1986.

[Roc70] Rockafellar, R. T., *Convex Analysis*, Princeton University Press, 1970.

[Roc76] Rockafellar, R. T., Monotone operators and the proximal point algorithm, *SIAM J Control Opt.*, **14**, (1976).

[Ros64] Rosen, J. B., Primal partitioning for block diagonal matrices, *Numerische Math.*, **6** , 250-260 (1964).

[Rus86] Ruszczyński, A., A regularized decomposition method for minimizing a sum of polyhedral functions, *Mathematical Programming*, **35**, 309-333 (1986).

[Sau76] Saunders, M. A., A fast, stable implementation of the simplex method using Bartels-Golub updating, in *Sparse Matrix Computations*, eds. J. R. Bunch and D. J. Rose, Academic Press, New York, 213-226 (1976).

[Str74] Strazicky, B., On an algorithm for solution of the two-stage stochastic programming problem, *Methods of Operations Research*, XIX, 142-156 (1974).

[Str80] Strazicky, B, Some results concerning an algorithm for the discrete recourse problem, in *Stochastic Programming*, ed. M, Dempster, Academic Press, London, 263-274 (1980).

[Tho87] Thompson, K. M., *Parallel and serial solution of large-scale linear complementarity problems*, Ph. D. dissertation, Department of Industrial Engineering, University of Wisconsin–Madison, August 1987.

[Van69] Van Slyke, R. and R. Wets, *L*-shaped linear programs with applications to optimal control and stochastic programming, *SIAM J. Appl. Math.*, **17**, 638-663 (1969).

[Wet83] Wets, R., Stochastic programming : solution techniques and approximation schemes, in *Mathematical Programming : The State of the Art, Bonn 1982*, eds. A. Bachem, M. Grötschel and B. Korte, Springer-verlag, Berlin, 566-603 (1983).

[Win74] Winkler, C., Basis factorization for block angular linear programs : unified theory of partitioning and decomposition using the simplex method, Tech. Rep. SOL 74-19, Systems Optimization Lab, Stanford University (1974).

[Wol75] Wolfe, P., A method of conjugate subgradients for minimizing nondifferentiable functions, *Mathematical Programming 3*, eds. Balanski and Wolfe, North Holand, 143-173, 1975.

[Zen86] Zenios, S. A. and J. M. Mulvey, Nonlinear Network Programming on Vector Supercomputers : a study on the CRAY X-MP, *Operations Research*, **34**, 667-682 (1986).

[Zow84] Zowe, J., Nondifferentiable Optimization—A motivation and a short Introduction into the subgradient–and the bundle concept, presented at the NATO Advanced Study Institute on Computational Mathematical Programming, Bad Windsheim, West-Germany, July 1984.