PIECEWISE-LINEAR APPROXIMATION METHODS AND PARALLEL ALGORITHMS IN OPTIMIZATION

by

Bernardo Feijoo

Computer Sciences Technical Report #598

May 1985

PIECEWISE-LINEAR APPROXIMATION METHODS AND PARALLEL ALGORITHMS IN OPTIMIZATION

by

BERNARDO FEIJOO

A thesis submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1985

ACKNOWLEDGEMENTS

I am profoundly thankful to my thesis advisor Professor Robert R. Meyer for his encouragement, advice and guidance, not only in the preparation of this thesis, but throughout my graduate studies at the University of Wisconsin-Madison.

I would also like to express my gratitude to Professors Olvi L. Mangasarian and James G. Morris for reading a draft of this thesis, and Professors Stephen M. Robinson and Raphael A. Finkel for serving on the Examination Committee.

I dedicate this thesis to my wife and my family. Their love and support made this work possible.

This research was supported by the Venezuelan Government through "Fundación Gran Mariscal de Ayacucho" and by the National Science Foundation under grant MCS8200632. This thesis was typeset using the computer facilities at the Mathematics Research Center of The University of Wisconsin.

PIECEWISE-LINEAR APPROXIMATION METHODS AND PARALLEL ALGORITHMS IN OPTIMIZATION

Bernardo Feijoo

Under the supervision of Professor Robert R. Meyer

ABSTRACT

In this thesis we develop an algorithm for the solution of nonseparable convex optimization problems. The algorithm is an iterative method based on piecewise-linear approximations of the objective function. A global convergence proof is given under the assumptions that the objective function is convex, Lipschitz continuous and differentiable and that the feasible set is convex and compact. We discuss the implementation of this method in parallel on the CRYSTAL multi-computer to solve multi-commodity traffic assignment problems. Encouraging computational results for the sequential and parallel versions in the case of non-linear network problems are discussed.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

ABSTRACT iii

Chapter 1 INTRODUCTION 1

Chapter 2 REVIEW OF OTHER METHODS 5

2.1 Global Approximation Methods 6

2.2 Local Piecewise-linear Approximations 11

Chapter 3 PIECEWISE-LINEAR APPROXIMATION METHODS 13

3.1 A General Algorithm 13

3.2 Separable Approximations 15

3.3 Piecewise-linear Approximations 18

3.4 Main Convergence Theorem 24

Chapter 4 OPTIMIZATION ON THE CRYSTAL MULTICOMPUTER 31

4.1 Parallel Algorithms for Nonlinear Networks 32

4.2 The CRYSTAL Multicomputer 35

4.3 Implementation on CRYSTAL 40

Chapter 5 COMPUTATIONAL IMPLEMENTATION 55

5.1 Implementation Details 55

5.2 Test Problems 57

5.3 Numerical Results 60

Chapter 6 DIRECTIONS FOR FURTHER RESEARCH 66

APPENDIX 70

REFERENCES 95

CHAPTER 1

INTRODUCTION

Throughout this thesis, we are concerned with the development and implementation of a general algorithm for the solution of convex optimization problems of the general form:

$$\begin{aligned} &\min \ f(\mathbf{x}) \\ &s.t.\ \mathbf{x} \in F \end{aligned} \tag{P}$$

where f is a convex function on the compact convex set $F \subseteq \mathbf{R}^n$. We further assume that a hyper-rectangle $H = \{\mathbf{x} \mid \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$ is given such that $F \subseteq H$ and f is *Lipschitz continuous* (i.e., $\forall\ \mathbf{x},\mathbf{y} \in H,\ |f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|$) and *differentiable* on H. The basic algorithm is an *iterative descent* method based on a procedure $\mathcal{P}$ that for a given non-optimal point $\mathbf{x} \in F$, returns a set $\mathcal{P}(\mathbf{x}) \subseteq F$ with the property that $\forall\ \mathbf{y} \in \mathcal{P}(\mathbf{x})$, $\mathbf{d} = \mathbf{y} - \mathbf{x}$ is a descent direction for f at $\mathbf{x}$. At each iteration an improved feasible solution is produced by first applying $\mathcal{P}$ to the current iterate and then performing a line search along the resulting descent direction.

For notational simplicity in the formulation of the approximating functions involved in our approach, we define the function

$$h_z : \mathbf{R}^n \longrightarrow \mathbf{R}$$

$$h_z(\mathbf{x}) = f(\mathbf{x} + \mathbf{z}) - f(\mathbf{z})$$

whose graph is a translation of the graph of f by the vector $-(\mathbf{z}, f(\mathbf{z}))$. Note that if $\mathbf{z}$ and $\mathbf{y}$ are feasible solutions of (P), then $\mathbf{y} - \mathbf{z}$ is a descent direction for f at $\mathbf{z}$ if and only if $\mathbf{y} - \mathbf{z}$ is a descent direction for h_z at $\mathbf{0}$. Given an iterate $\mathbf{x}^j$, our procedure $\mathcal{P}$ involves approximating $h_{\mathbf{x}^j}$ by a separable function $s_{\mathbf{x}^j}$ that will be called the *underlying separable approximation* to $h_{\mathbf{x}^j}$. A piecewise-linear approximation to $s_{\mathbf{x}^j}$ is then used to obtain $\tilde{\mathbf{x}}^j$ such that $\mathbf{d}^j = \tilde{\mathbf{x}}^j - \mathbf{x}^j$ is a descent direction for $h_{\mathbf{x}^j}$ at $\mathbf{0}$. The procedure returns $\mathcal{P}(\mathbf{x}^j) = \{\tilde{\mathbf{x}}^j\}$. In chapter 3 we prove that this approach yields a sequence of iterates whose objective values converge to the optimal value of (P). It is also shown in that chapter that any other procedure $\mathcal{P}$ that produces a descent direction and defines a closed map gives rise to a convergent algorithm. For example, it's well known that optimization of the linearized objective function will produce a descent direction provided that $\mathbf{x}^j$ is feasible and non-optimal, and this procedure is closed. In the linearly constrained case this is the well-known Frank-Wolfe method. In practice, the choice of $\mathcal{P}$ would largely depend on the particular structure of the constraints defining F.

In chapter 3 we also describe in detail the main properties of the underlying separable approximations, including the piecewise-linear approximation method used to attack the separable subproblems in order to obtain the desired descent properties.

Although the algorithm and approximation techniques are developed for problems of the general form of (P), our aim is to provide an efficient algorithm for the case in which n is large and F is given by a set of network constraints plus upper and lower bounds (i.e., $F = \{\mathbf{x} \mid \mathbf{Ax} = \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$ where $\mathbf{A}$ is a node-arc incidence matrix). For this particular constraint structure the subproblems resulting from the piecewise-linear approximation can be transformed into linear network problems (see [Kamesam and Meyer 82]) that can be solved using the very fast algorithms available for such problems. In other words, we try to take full advantage of the network structure of the problem. Chapter 2 is a brief review of some of the methods readily applicable to problems of this kind.

These problems arise in a great variety of applications. Applications areas include computer network design [Cantor and Gerla 74], [Gavish and Hantler 82], [Magnanti and Wong 84], urban traffic assignment [Bertsekas and Gafni 82], [Dantzig, et al, 79], [Lawphongpanich and Hearn 83], [Pang and Yu 82], hydro-electric power systems [Hanscom, et al, 80], [Rosenthal 81], telecommunication networks [McCallum 76], and water supply systems [Beck, et al, 83]. These problems are among the largest nonlinear programming problems that are currently being studied because large-scale networks of various sorts not only arise naturally in many different areas and represent a type of model that people in a variety of disciplines find intuitively easy to develop and maintain, but also exhibit a mathematical structure that can be exploited to develop algorithms that in many cases can solve at affordable cost problems involving tens of thousands of variables (In the case of linear networks, [Barr and Turner 81] describe the solution of problems with 50,000 constraints and 65 million variables being run on a "production basis" for the U. S. Department of the Treasury).

Of particular interest in terms of widespread applicability are problems of traffic routing, equilibrium, and network design in computer and urban transportation networks. These problems are typically multi-commodity problems, i.e., they involve many different types of *commodities* flowing through the network where, depending on the application, a *commodity* will be associated with the traffic flowing out of a *source* node or between a designated origin-destination pair. Enormous sizes result from the fact that the number of variables and constraints is determined by the number of links and nodes multiplied by the number of commodities.

Some of these problems, in particular traffic assignment problems, have the property that coupling between the commodities occurs only in the objective function. Because of this property, our approach results in a decomposition of the linear network subproblems into separate and smaller linear network problems that can be solved in parallel on distributed computer architectures. The CRYSTAL multicomputer [DeWitt, et al, 84] under development in the Computer Sciences Department at the University of Wisconsin-Madison was used to implement our algorithm in such a way. In chapter 4 we describe this implementation in detail.

In chapter 5 we are concerned with computational considerations with emphasis on 1) generating only the needed segments of the piecewise-linear approximation, 2) the linear programming or network subproblems that result when $\boldsymbol{F}$ has appropriate structure and 3) a comparison of computational results for the original algorithm and its parallel implementation on CRYSTAL. Chapter 6 presents some directions for further research.

CHAPTER 2

REVIEW OF OTHER METHODS

Problem (P), as stated, encompasses a very large class of convex optimization problems and there are many nonlinear programming algorithms that can be used to solve such a problem. The choice of algorithm clearly depends on the particular structure of the constraints defining F and also on the nature of the objective function f. We are mainly interested in the subclass of problems in which F is determined by a set of linear network constraints plus upper and lower bounds on the variables. In this chapter we review some of the algorithms that can be used or specialized to solve problems in this subclass. In section 2.1 we discuss algorithms based on global approximations of the objective function and in section 2.2 we review the use of local approximations in the case in which the objective function is separable.

2.1. GLOBAL APPROXIMATION METHODS

Within this large collection of algorithms we consider the Frank-Wolfe method, piecewise-linear approximation methods, generalized programming, the reduced gradient method, and second order methods such as Newton, quasi-Newton and variable metric methods.

The Frank-Wolfe method [Frank and Wolfe 56] is by far the simplest of these methods in terms of implementation. It is an iterative procedure that uses linearization of the objective function at the current iterate. The resulting linear subproblem is solved and a line search is performed in the direction indicated by the new feasible solution. At each iteration a lower bound on the optimal value is easily obtained and the process is repeated until the current objective value is within a certain tolerance of the current lower bound. In the network case the subproblems generated are all linear network problems and as such can be solved using the very fast codes developed for these problems. Unfortunately, the Frank-Wolfe method has been known to have rather slow convergence and thus is not considered a practical procedure for large-scale optimization.

Global piecewise-linear approximation methods consist of iteratively approximating f on the hyper-rectangle H by a piecewise-linear function $\tilde{f}$, obtaining at each iteration an approximate solution to the original problem. This is achieved by setting up a grid of points in H and using the values of f at these points to generate $\tilde{f}$. The main disadvantage of these methods is the need to generate a rather large number of grid points in order to get a fairly accurate

approximation. This poses problems of storage and efficiency, since the number of function evaluations and the size of the linear subproblems become very large. Furthermore, if the objective function is not separable, the resulting subproblems don't inherit the network structure of the original problem even though they are still linear. Thus the fast network codes cannot be used.

The generalized programming method uses linear approximating subproblems based on function evaluations and solution of these via the simplex method. At each iteration there are two major steps performed, solving a master linear problem and then a Lagrangian relaxation. In general, solving the Lagrangian relaxation requires considerable computational effort, problems of storage and efficiency occur when the number of grid points increases, and once again the network structure of the constraints is lost in the subproblems.

The reduced gradient method [Wolfe 67] is an iterative procedure originally developed to solve problems of the form

$$\min f(\mathbf{x})$$

$$s.t.\ \mathbf{A}\mathbf{x} = \mathbf{b}$$

$$\mathbf{x} \geq \mathbf{0}$$

where f is continuously differentiable and convex, and $\mathbf{A}$ is an $m \times n$ matrix with full row rank. The nondegeneracy assumption that every basic solution to the constraints has m strictly positive variables is made. A partition of the columns of $\mathbf{A}$ is considered, as follows:

$$\mathbf{A} = (\mathbf{B} \quad \mathbf{S} \quad \mathbf{N})$$

where $\mathbf{B}$ denotes a basis matrix, $\mathbf{S}$ corresponds to *superbasic* variables (i.e., nonbasic variables allowed to vary) and $\mathbf{N}$ denotes nonbasic variables whose values are fixed at 0. This partition induces corresponding partitions on the variables, the gradient of f and the search direction:

$$\mathbf{x} = \begin{pmatrix} \mathbf{x_B} \\ \mathbf{x_S} \\ \mathbf{x_N} \end{pmatrix} \quad \nabla f(\mathbf{x}) = \begin{pmatrix} \nabla_{\mathbf{B}} f(\mathbf{x}) \\ \nabla_{\mathbf{S}} f(\mathbf{x}) \\ \nabla_{\mathbf{N}} f(\mathbf{x}) \end{pmatrix} \quad \mathbf{d} = \begin{pmatrix} \mathbf{d_B} \\ \mathbf{d_S} \\ \mathbf{d_N} \end{pmatrix} .$$

The variables in $\mathbf{x_S}$ and $\mathbf{x_N}$ are regarded as independent variables, whereas the variables in $\mathbf{x_B}$ are considered dependent, since they can be obtained from $\mathbf{x_S}$ and $\mathbf{x_N}$. At each iteration the reduced gradient

$$\begin{pmatrix} \mathbf{r_S} \\ \mathbf{r_N} \end{pmatrix} = \begin{pmatrix} \nabla_{\mathbf{S}} f(\mathbf{x}) \\ \nabla_{\mathbf{N}} f(\mathbf{x}) \end{pmatrix} - \nabla_{\mathbf{B}} f(\mathbf{x})^T \mathbf{B}^{-1} \begin{pmatrix} \mathbf{S} \\ \mathbf{N} \end{pmatrix}$$

is computed and a search direction is obtained by letting $\mathbf{d_S} = -\mathbf{r_S}$, $\mathbf{d_N} = \mathbf{0}$. The direction $\mathbf{d_B}$ is then given by $\mathbf{d_B} = -\mathbf{B}^{-1}\mathbf{S}\mathbf{d_S}$. Searching along the resulting direction $\mathbf{d}$, either a basic variable reaches its lower bound or we find the minimum value of the function along $\mathbf{d}$. In the former case a pivot operation is performed and the partition is changed accordingly. In both cases a new iterate is found and the process repeated.

Note that letting

$$\mathbf{Z} = \begin{pmatrix} -\mathbf{B}^{-1}\mathbf{S} \\ \mathbf{I} \\ \mathbf{0} \end{pmatrix}$$

the search direction is given by

$$\mathbf{d} = \mathbf{Z}\mathbf{v}$$

where

$$\mathbf{v} = -\mathbf{Z}^T \nabla f(\mathbf{x}) .$$

In recent work, the convergence of the reduced gradient method has been improved by introducing the use of second order information and the method has been specialized for nonlinear network problems. [Dembo and Klincewicz 81] discuss a scaled reduced gradient algorithm for nonlinear network flow problems, in which a positive definite scaling matrix $\mathbf{M}$ is used in the direction finding step. The search direction is then given by

$$\mathbf{d} = \mathbf{Z}\mathbf{v}$$

where

$$\mathbf{M}\mathbf{v} = -\mathbf{Z}^T \nabla f(\mathbf{x}) \ .$$

This is motivated by the fact that if we were to use a constrained-Newton algorithm to solve the problem, for a given feasible point $\mathbf{x}$, we would have to solve the following quadratic program to compute a feasible search direction.

$$\min_{\mathbf{p}} \ \frac{1}{2}\mathbf{p}^T\mathbf{H}\mathbf{p} + \nabla f(\mathbf{x})^T\mathbf{p}$$

$$s.t. \ \mathbf{A}\mathbf{p} = \mathbf{0}$$

$$\mathbf{p}_j \geq 0 \text{ if } \mathbf{x}_j = 0$$

where $\mathbf{H}$ is the Hessian matrix of f at $\mathbf{x}$. If we ignore the constraints $\mathbf{p}_j \geq 0$, then letting $\mathbf{p} = \mathbf{Z}\mathbf{y}$, the problem becomes an unconstrained problem in $\mathbf{y}$, since $\mathbf{A}\mathbf{Z} = \mathbf{0}$. The solution to this problem is obtained by solving

$$(\mathbf{Z}^T\mathbf{H}\mathbf{Z})\mathbf{y} = -\mathbf{Z}^T \nabla f(\mathbf{x})$$

and the optimal solution of the constrained problem is recovered from $\mathbf{p} = \mathbf{Z}\mathbf{y}$. In other words, the reduced gradient method can be thought of as being a quasi-Newton method, in which the reduced Hessian $\mathbf{Z}^T\mathbf{H}\mathbf{Z}$ is approximated by the

identity matrix. Thus, by introducing the scaling matrix $\mathbf{M}$ containing some second order information, we hope to get a more accurate approximation to the reduced Hessian. Dembo and Klincewicz prove that the rate of convergence for this algorithm corresponds to the canonical rate given by the largest and smallest eigenvalues of $\mathbf{M}^{-\frac{1}{2}T}\mathbf{Z}^T\mathbf{H}\mathbf{Z}\mathbf{M}^{-\frac{1}{2}}$. That is, if a and A are, respectively, the smallest and largest eigenvalues of that matrix, the sequence of objective values converges linearly with a ratio no greater than $[\frac{(A-a)}{(A+a)}]^2$. Their tests include experiments in which $\mathbf{M} = \mathbf{H_S}$ and $\mathbf{M} = diag(\mathbf{Z}^T\mathbf{H}\mathbf{Z})$ and they obtain encouraging results for separable networks, where these matrices may be efficiently generated. Their algorithm makes efficient use of the network programming data structures and uses a preprocessing device that resolves degeneracy problems.

For nonlinear networks, [Beck, et al 83] propose a reduced gradient algorithm that uses the network data structures efficiently and relies on direction finding procedures based on conjugate gradient methods. They concentrate on the computational implementation and show promising results for separable and nonseparable network problems.

The subproblems generated by these approaches are, in general, more complex than the ones resulting from the approach proposed in this thesis.

For problems with nonlinear objective functions and linear constraints, a version of the reduced gradient method is used in the MINOS optimization package [Murtagh and Saunders 83]. This method uses a stable implementation of the quasi-Newton algorithm to obtain the search direction for the superbasic variables, by solving a system of the form

$$\mathbf{C}^T\mathbf{C}\mathbf{d}_s = -\mathbf{Z}^T\nabla f(\mathbf{x})$$

where $\mathbf{Z}$ is as above and $\mathbf{C}$ is an upper triangular matrix that is updated in various ways in order to make $\mathbf{C}^T\mathbf{C} \approx \mathbf{Z}^T\mathbf{H}\mathbf{Z}$. This implementation makes no explicit use of the possible network structure of the constraints and, for large-scale problems, the solution of the system of equations may pose storage and efficiency problems.

Newton, quasi-Newton and variable metric methods use quadratic approximations of the objective function as an alternative to linear or piecewise-linear approximations. It is possible to ensure global convergence of these methods by including a suitable stepsize procedure. The main disadvantage of these methods, in the case of general large-scale problems, is that solving very large quadratic approximating problems could be computationally very expensive. However, it is possible to solve large positive definite separable quadratic programming problems by highly efficient iterative schemes based on generalizations of SOR methods [Cryer 71] [Mangasarian 77, 81]. In the case of f being separable and having positive definite Hessians, these methods can be used to solve the subproblems generated by an iterative quadratic programming approach. If f is not separable, but its Hessian has positive diagonal, it would be interesting to investigate the use of these methods to solve the underlying separable subproblem at each iteration, instead of generating piecewise-linear approximations.

2.2. LOCAL PIECEWISE-LINEAR APPROXIMATIONS

In the case of f being a separable function (i.e., $f(\mathbf{x}) = \sum f_i(x_i)$) several approximating approaches have been suggested in recent years. These include

the use of local piecewise-linear approximations to each of the f_i [Meyer 79,80], [Kao and Meyer 81] and implicit global approximations [Kamesam and Meyer 82]. The latter form the basis for our direction finding procedure.

Local approximation methods are based on approximating each component function f_i around the i^{th} component of the current iterate by $2k-$segment piecewise-linear functions. A sequence of these approximating problems is considered and the new iterate is chosen to be the "best" feasible solution amongst the solutions of the sequence of subproblems. Computationally, the first solution that effects a "substantial" reduction in the objective function is chosen as the next iterate. These methods have the clear advantage of having linear subproblems of fixed size depending on the number of segments in the approximation and thus do not have the problem of trading off accuracy for storage and efficiency restrictions. The subproblems also maintain the network structure of the original problem and thus can be efficiently solved.

Implicit global approximation methods are based on considering a piecewise-linear approximation of f on all of H but in practice working with only a two-segment approximation per variable at a given time. This is achieved by a column generation-column dropping strategy. The separability of the function implies that the approximation dominates the objective function on H and thus a line search procedure is not needed. This method keeps the size of the linear subproblems fixed and maintains the network structure of the original problem. Computational experience has shown nice convergence properties for large separable nonlinear network problems.

CHAPTER 3

PIECEWISE-LINEAR APPROXIMATION METHODS

In this chapter we discuss the general algorithm, our approach to $\mathcal{P}$ and the convergence properties of this particular choice of procedure. Section 3.1 consists of a description of the main algorithm. In section 3.2 we discuss the properties of the underlying separable approximation. Section 3.3 is devoted to the description of the piecewise-linear approximating problems and the resulting procedure $\mathcal{P}$. In section 3.4 we establish the convergence properties of all approximating functions involved and finally we state and prove the main convergence theorem.

3.1. A GENERAL ALGORITHM

We assume that we have an algorithmic map $\mathcal{P}$ with the property that if $\mathbf{x} \in F$ and $\mathbf{x}$ is not an optimal solution of (P), then $\mathcal{P}(\mathbf{x}) \subseteq F$ and $\forall\, \mathbf{y} \in \mathcal{P}(\mathbf{x})$,

$\mathbf{d} = \mathbf{y} - \mathbf{x}$ is a descent direction for f at $\mathbf{x}$. We also need a line search map. For a feasible point $\mathbf{x}$ and a descent direction $\mathbf{d}$ we consider the problem:

$$\begin{aligned} \min_{\theta} \quad & f(\mathbf{x} + \theta\mathbf{d}) \\ s.t. \quad & \mathbf{x} + \theta\mathbf{d} \in F \\ & \theta \geq 0, \end{aligned} \tag{S}$$

and define a search map S by $S(\mathbf{x}, \mathbf{d}) = \{\mathbf{y} \mid \mathbf{y} = \mathbf{x} + \theta^{\star}\mathbf{d}$, and $\theta^{\star}$ solves (S)$\}$. For simplicity we initially assume exact line searches. The convergence proof is easily extended to any of the inexact line searches based on Armijo-Goldstein rules (see, e.g., [Luenberger 84]) A proof for a modified Goldstein search is given in section 3.4.

In this setting the algorithm is defined as follows:

Algorithm 3.1.1:

Step 0: *Let* $\mathbf{x}^1$ *be a starting feasible solution and* $j \leftarrow 1$.

Step 1: *If* $\mathbf{x}^j$ *satisfies the stopping criterion then stop,*

else

compute $\tilde{\mathbf{x}}^j \in P(\mathbf{x}^j)$.

Step 2: *Let* $\mathbf{d}^j := \tilde{\mathbf{x}}^j - \mathbf{x}^j$. *Obtain* $\mathbf{x}^{j+1} \in S(\mathbf{x}^j, \mathbf{d}^j)$. *Set* $j \leftarrow j + 1$ *and go to Step 1.*

3.2. SEPARABLE APPROXIMATIONS

Let $\mathbf{z}$ be a feasible solution of the problem (P). The *underlying separable approximation* to h_z is defined as follows:

$$s_z(\mathrm{x}) := \sum_{i=1}^{n} s_z^{(i)}(x_i)$$

where $s_z^{(i)}(x_i) = h_z(x_i \mathbf{e}^i)$ and $\mathbf{e}^i$ is the i^{th} canonical unit vector, i.e., all variables except the i^{th} are zero.

Our first lemma establishes some useful properties of the function s_z.

Lemma 3.2.1. *s_z satisfies the following properties:*

(1) s_z is a separable, Lipschitz continuous, differentiable convex function on $H - \mathbf{z}$;

(2) $s_z(\mathbf{0}) = h_z(\mathbf{0}) = 0$;

(3) $\nabla s_z(\mathbf{0}) = \nabla h_z(\mathbf{0}) = \nabla f(\mathbf{z})$.

Proof: The Lipschitz continuity, differentiability and convexity of s_z are inherited from those of h_z. Property (2) is easily checked from the definition of s_z. Property (3) is also straightforward: Let $1 \leq j \leq n$, then

$$\frac{\partial s_z}{\partial x_j}(\mathrm{x}) = \frac{ds_z^{(j)}}{dx_j}(x_j) = \nabla h_z(x_j \mathbf{e}^j) \cdot \mathbf{e}^j = \frac{\partial h_z}{\partial x_j}(x_j \mathbf{e}^j)$$

hence, letting $\mathrm{x} = \mathbf{0}$, we get

$$\frac{\partial s_z}{\partial x_j}(\mathbf{0}) = \frac{\partial h_z}{\partial x_j}(\mathbf{0})$$

and this concludes the proof. ∎

The concepts of feasible direction and descent direction are needed for the discussion of the following results. We now give a formal definition of these terms.

Definition 3.2.2. *Given* $\mathbf{x} \in \boldsymbol{F}$, *we will say* $\mathbf{d}$ *is a feasible direction at* $\mathbf{x}$ *if there is an* $\bar{\alpha} > 0$ *such that* $\mathbf{x} + \alpha\mathbf{d} \in \boldsymbol{F}$ *for all* $\alpha \in [0, \bar{\alpha}]$. *Given* $\mathbf{x} \in \boldsymbol{F}$ *and* $\boldsymbol{f}$ *we say that* $\mathbf{d}$ *is a descent direction for* $\boldsymbol{f}$ *at* $\mathbf{x}$ *if there is an* $\hat{\alpha} > 0$ *such that* $\boldsymbol{f}(\mathbf{x} + \alpha\mathbf{d}) < \boldsymbol{f}(\mathbf{x})$ *for all* $\alpha \in [0, \hat{\alpha}]$.

It's well known that if $\boldsymbol{f}$ is convex and differentiable, $\mathbf{d}$ is a descent direction for $\boldsymbol{f}$ at $\mathbf{x}$ if and only if $\nabla \boldsymbol{f}(\mathbf{x}) \cdot \mathbf{d} < 0$.

The following lemma establishes the relationship between the optimal solution $\mathbf{x}^\star$ of (P) and that of the underlying separable problem centered at $\mathbf{x}^\star$.

Lemma 3.2.3. *The point* $\mathbf{0}$ *is an optimal solution of the separable problem:*

$$\begin{aligned} &\min \ \ s_{\mathbf{x}^\star}(\mathbf{x}) && (\mathrm{SP}(\mathbf{x}^\star)) \\ &\textit{s.t.} \ \ \mathbf{x} \in \boldsymbol{F} - \mathbf{x}^\star \end{aligned}$$

if and only if $\mathbf{x}^\star$ *is an optimal solution of the original problem* (P).

Proof:

[⇒]

Let $\mathbf{d}$ be a feasible direction at $\mathbf{x}^\star$, (if $\mathbf{d}$ doesn't exist, then $\boldsymbol{F} = \{\mathbf{x}^\star\}$ and the lemma is trivial). Since $\mathbf{0}$ is optimal for $(\mathrm{SP}(\mathbf{x}^\star))$ and since $\mathbf{d}$ is a feasible

direction for s_{x^*} at $\mathbf{0}$, we have $\nabla s_{x^*}(\mathbf{0}) \cdot \mathbf{d} \geq 0$. But by lemma 3.2.1(3) we have $\nabla s_{x^*}(\mathbf{0}) = \nabla f(x^*)$ so $\nabla f(x^*) \cdot \mathbf{d} \geq 0$ also, and by the convexity of f, $f(x^* + \alpha \mathbf{d}) \geq f(x^*)$, $\forall \alpha \geq 0$. This is true for any feasible direction at x^*, hence x^* is an optimal solution of (P).

$|\Leftarrow|$

Follows by reversing the roles of s_{x^*} and f. ∎

The next lemma establishes a key descent relationship between f and its separable approximations. It shows that it suffices to find a feasible point that improves the separable approximation in order to get a descent direction for the original objective function.

Lemma 3.2.4. *Let* $y, z \in F$, *if* $s_z(y - z) < 0$, *then* $\mathbf{d} = y - z$ *is a descent direction for* s_z *at* $\mathbf{0}$ *and for* f *at* z.

Proof: Since s_z is convex and differentiable and $s_z(\mathbf{0}) = 0$, then, for every $x \in F$, we have

$$\nabla s_z(\mathbf{0}) \cdot (x - z) \leq s_z(x - z)$$

thus

$$\nabla s_z(\mathbf{0}) \cdot (y - z) \leq s_z(y - z) < 0$$

then, from lemma 3.2.1(3), we have

$$\nabla f(z) \cdot (y - z) < 0$$

so $\mathbf{d}$ is a descent direction for s_z at $\mathbf{0}$ and for f at z. ∎

3.3. PIECEWISE-LINEAR APPROXIMATIONS

So far we have approximated h_z by a separable and in general nonlinear function s_z. We don't attempt to solve the resulting approximating problem but instead we use a piecewise-linear approximation to s_z and, at each iteration, solve the resulting problem. This provides us with the desired descent direction. In this section we describe in detail the piecewise-linear approximating functions to be used and define precisely our procedure $\mathcal{P}$.

Given $z \in F$ and a vector $\Lambda = (\lambda_1, \ldots, \lambda_n) > 0$ of gridsizes, a piecewise-linear approximation $\tilde{f}$ to s_z is defined as follows:

$$\tilde{f}(z, \Lambda, x) = \sum_{i=1}^{n} \tilde{f}_i(z, \Lambda, x_i)$$

where

$$\tilde{f}_i(z, \Lambda, x_i) = \begin{cases} s_z^{(i)}((k_i - 1)\lambda_i) + c_i^{+k_i}\{x_i - (k_i - 1)\lambda_i\} \\ \quad \text{for } (k_i - 1)\lambda_i \le x_i \le k_i\lambda_i \\ \quad k_i = 1, 2, \ldots, s_i \\ s_z^{(i)}(-(k_i - 1)\lambda_i) + c_i^{-k_i}\{x_i + (k_i - 1)\lambda_i\} \\ \quad \text{for } -k_i\lambda_i \le x_i \le -(k_i - 1)\lambda_i \\ \quad k_i = 1, 2, \ldots, s_i' \end{cases}$$

and

$$c_i^{+k_i} = \frac{\{s_z^{(i)}(k_i\lambda_i) - s_z^{(i)}((k_i - 1)\lambda_i)\}}{\lambda_i}$$

$$-c_i^{-k_i} = \frac{\{s_z^{(i)}(-k_i\lambda_i) - s_z^{(i)}(-(k_i - 1)\lambda_i)\}}{\lambda_i}$$

and s_i and s_i' are chosen so that $s_i\lambda_i = u_i - z_i$ and $-s_i'\lambda_i = l_i - z_i$. (For notational simplicity, we assume that at any iteration all segments used to construct an approximation $\tilde{f}_i$ are *equal*. In practice the segments near the boundaries defined by l_i and u_i are generally smaller than λ_i. This poses no problems in utilizing the theory to be developed since the proofs below merely utilize the fact that λ_i is an *upper bound* on segment size. Moreover, the segments of $\tilde{f}_i$ may be generated *as needed* starting at 0, and it is seldom necessary to generate more than a few such segments. In a sense, $\tilde{f}_i$ is implicitly rather than explicitly generated). Note that the convexity of $s_z^{(i)}$ implies that $s_z(x_i) \leq \tilde{f}_i(z, \Lambda, x_i)$ for all $x_i \in [l_i - z_i, u_i - z_i]$ and hence $\tilde{f}$ dominates s_z on $H - z$. This property plays an important role in the convergence arguments.

Given the iterate x^j constructed in the j^{th} iteration, we define an approximating problem

$$\min_{x} \sum_{i=1}^{n} \tilde{f}_i(x^j, \Lambda^j, x_i) \qquad (AP(x^j, \Lambda^j))$$
$$s.t.\ x \in F - x^j$$

that will be the basis for our procedure $\mathcal{P}$ for finding the search direction.

Procedure $\mathcal{P}(x^j)$:

Step 1: *If* $j = 1$, *then choose* Λ^1 *such that* $0 \leq \Lambda^1 \leq u - l$,

else

for $\alpha \in (0,1)$ *let* $\Lambda^j \leftarrow \alpha\Lambda^{j-1}$.

Step 2: *Obtain an optimal solution* $\tilde{y}$ *of* $(AP(x^j, \Lambda^j))$.

If $\tilde{f}(x^j, \Lambda^j, \tilde{y}) = 0$, *then* $\Lambda^j \leftarrow \alpha\Lambda^j$ *and repeat Step 2,*

else

$\tilde{x}^j \leftarrow \tilde{y} + x^j$ *and stop, returning* $\{\tilde{x}^j\}$.

It should be noted that $\tilde{f}$ is defined in such a way that $\tilde{f}(z, \Lambda, 0) = 0$ and thus the test in Step 2 of $\mathcal{P}$ determines if the mesh is fine enough to produce a descent direction for $\tilde{f}$ (and hence for s_z and f). In each call to $\mathcal{P}$ the mesh size is reduced by a factor of α. If this reduction is not enough to produce an improvement in the piecewise-linear approximation we repeatedly cut down the mesh size until such an improvement is achieved. We know from earlier results for piecewise-linear approximations of separable functions (see, e.g., [Kamesam and Meyer, 82]) that if 0 is not an optimal solution for $(SP(x^j))$, then for a small enough Λ^j a better feasible solution will be found, i.e., $\tilde{f}(x^j, \Lambda^j, \tilde{x}^j - x^j) < 0$. In most cases in practice, however, only the initial optimal solution is needed.

The next three lemmas establish the uniform convergence of the approximating functions considered.

Lemma 3.3.1. *If* K *is an index set such that* $x^j \xrightarrow{j \in K} \bar{x}$, *then*

$$s_{x^j}(x - x^j) \xrightarrow{j \in K} s_{\bar{x}}(x - \bar{x})$$

uniformly in x.

Proof: Given $\epsilon > 0$, $\exists\, N$ such that if $j \in K$ and $j \geq N$ then $||\mathrm{x}^j - \bar{\mathrm{x}}||_1 < \frac{\epsilon}{2nL}$.

Then for such ϵ we have

$$|s_{\mathrm{x}^j}(\mathrm{x} - \mathrm{x}^j) - s_{\bar{\mathrm{x}}}(\mathrm{x} - \bar{\mathrm{x}})| \leq \sum_{i=1}^{n} |s^{(i)}_{\mathrm{x}^j}(x_i - x_i^j) - s^{(i)}_{\bar{\mathrm{x}}}(x_i - \bar{x}_i)|$$

$$= \sum_{i=1}^{n} |h_{\mathrm{x}^j}((x_i - x_i^j)\mathrm{e}^i) - h_{\bar{\mathrm{x}}}((x_i - \bar{x}_i)\mathrm{e}^i)|$$

$$\leq \sum_{i=1}^{n} |f(\mathrm{x}^j + (x_i - x_i^j)\mathrm{e}^i) - f(\bar{\mathrm{x}} + (x_i - \bar{x}_i)\mathrm{e}^i)| + |f(\mathrm{x}^j) - f(\bar{\mathrm{x}})|$$

$$\leq \sum_{i=1}^{n} L(||(\mathrm{x}^j + (x_i - x_i^j)\mathrm{e}^i) - (\bar{\mathrm{x}} + (x_i - \bar{x}_i)\mathrm{e}^i)||_1 + ||\mathrm{x}^j - \bar{\mathrm{x}}||_1)$$

$$\leq \sum_{i=1}^{n} 2L||\mathrm{x}^j - \bar{\mathrm{x}}||_1 = 2nL||\mathrm{x}^j - \bar{\mathrm{x}}||_1 < \epsilon$$

Since ϵ is arbitrary this concludes the proof. ∎

Lemma 3.3.2. *If* $\Lambda^j \xrightarrow{j \in K} 0$ *for* $K \subseteq \mathbf{N}$. *Then* $\tilde{f}(\mathrm{z}, \Lambda^j, \mathrm{x}) \xrightarrow{j \in K} s_{\mathrm{z}}(\mathrm{x})$ *uniformly in* x *and* z.

Proof: Let $j \in K$, $\mathrm{z} \in F$ and $\mathrm{x} \in F - \mathrm{z}$. Then if we define

$$E_i = |\tilde{f}_i(\mathrm{z}, \Lambda^j, x_i) - s^{(i)}_{\mathrm{z}}(x_i)|$$

we have

$$|\tilde{f}(\mathrm{z}, \Lambda^j, \mathrm{x}) - s_{\mathrm{z}}(\mathrm{x})| \leq \sum_{i=1}^{n} E_i$$

An estimate for E_i in the presence of a Lipschitz condition for $s^{(i)}_{\mathrm{z}}$ is well known (see, e.g., [Thakur 78]):

$$E_i \leq \frac{L\lambda_i^j}{2}$$

So now we have that

$$|\tilde{f}(z, \Lambda^j, x) - s_z(x)| \leq \frac{L}{2} \sum_{i=1}^{n} \lambda_i^j = \frac{L||\Lambda^j||_1}{2}$$

and since we have $\Lambda^j \xrightarrow{j \in K} 0$ the convergence is uniform. ∎

Lemma 3.3.3. *Let $K \subseteq \mathbf{N}$ be an index set. If $\Lambda^j \xrightarrow{j \in K} 0$ and $x^j \xrightarrow{j \in K} \bar{x}$, then*

$$\tilde{f}(x^j, \Lambda^j, x - x^j) \xrightarrow{j \in K} s_{\bar{x}}(x - \bar{x})$$

uniformly in x.

Proof: Since

$$|\tilde{f}(x^j, \Lambda^j, x - x^j) - s_{\bar{x}}(x - \bar{x})| \leq$$

$$|\tilde{f}(x^j, \Lambda^j, x - x^j) - s_{x^j}(x - x^j)| + |s_{x^j}(x - x^j) - s_{\bar{x}}(x - \bar{x})|$$

the result follows from lemmas 3.3.1 and 3.3.2. ∎

The following theorem uses the continuity properties of the approximating functions to show that the optimal solutions of the approximating problems have the appropriate continuity property.

Theorem 3.3.4. *Let $x^\star$, $\tilde{x} \in F$ and $K \subseteq \mathbf{N}$ be an index set such that*

(1) $x^j \xrightarrow{j \in K} x^\star$.

(2) $\tilde{x}^j - x^j$ *is an optimal sol. of* $(\mathrm{AP}(x^j, \Lambda^j))$, *for* $j \in K$ *with* $\tilde{x}^j \xrightarrow{j \in K} \tilde{x}$.

(3) $\Lambda^j \xrightarrow{j \in K} 0$.

Then $\tilde{x} - x^*$ *is an optimal solution of* $(SP(x^*))$.

Proof: Let $x^{**} - x^*$ be an optimal solution of $(SP(x^*))$ and choose $\epsilon > 0$. By lemma 3.3.3 and the continuity of s_{x^*}, $\exists\, N$ such that for $j \in K$, $j \geq N$ we have $|\tilde{f}(x^j, \Lambda^j, x - x^j) - s_{x^*}(x - x^*)| < \epsilon$, $\forall\, x \in H$ and $|s_{x^*}(\tilde{x}^j - x^*) - s_{x^*}(\tilde{x} - x^*)| < \epsilon$. In particular

$$\tilde{f}(x^j, \Lambda^j, x^{**} - x^j) < s_{x^*}(x^{**} - x^*) + \epsilon \tag{1}$$

$$s_{x^*}(\tilde{x} - x^*) < s_{x^*}(\tilde{x}^j - x^*) + \epsilon \tag{2}$$

and similarly

$$s_{x^*}(\tilde{x}^j - x^*) < \tilde{f}(x^j, \Lambda^j, \tilde{x}^j - x^j) + \epsilon \tag{3}$$

but from (2) and (3) we get

$$s_{x^*}(\tilde{x} - x^*) < \tilde{f}(x^j, \Lambda^j, \tilde{x}^j - x^j) + 2\epsilon \tag{4}$$

On the other hand, since $x^{**} - x^j \in F - x^j$ and $\tilde{x}^j - x^j$ is optimal for $(AP(x^j, \Lambda^j))$, we have

$$\tilde{f}(x^j, \Lambda^j, \tilde{x}^j - x^j) \leq \tilde{f}(x^j, \Lambda^j, x^{**} - x^j) \tag{5}$$

so combining inequalities (1), (4) and (5) yields

$$s_{x^*}(\tilde{x} - x^*) < s_{x^*}(x^{**} - x^*) + 3\epsilon$$

Since ϵ is arbitrary, $s_{x^*}(\tilde{x} - x^*) \leq s_{x^*}(x^{**} - x^*)$ and the optimality of $\tilde{x} - x^*$ is established. ∎

3.4. MAIN CONVERGENCE THEOREM

In this section we prove that algorithm 3.1.1, with $\mathcal{P}$ as described in the previous section, generates a sequence of iterates whose objective values converge to the optimal value of (P). We also extend the proof to handle an inexact line search procedure based on a Goldstein-type test.

Our first theorem shows that the sequence of objective values of the iterates is a monotone strictly decreasing sequence.

Theorem 3.4.1. *Let* x^j *and* x^{j+1} *be iterates generated by algorithm 3.1.1 with* $\mathcal{P}$ *defined as in section 3.3. If* x^j *is not an optimal solution of (P), then* $f(x^{j+1}) < f(x^j)$.

Proof: Since x^j is not optimal for problem (P) we have, by lemma 3.2.3, that $\mathbf{0}$ is *not* optimal for $(SP(x^j))$. As mentioned in the previous section , we know that $\mathcal{P}$ will produce $\tilde{x}^j$ such that $\tilde{f}(x^j, \Lambda^j, \tilde{x}^j - x^j) < 0$. The approximation $\tilde{f}$ is defined so that $\tilde{f}(x^j, \Lambda^j, x) \geq s_{x^j}(x), \ \forall\, x \in H - x^j$. These inequalities imply

$$s_{x^j}(\tilde{x}^j - x^j) \leq \tilde{f}(x^j, \Lambda^j, \tilde{x}^j - x^j) < 0$$

so by lemma 3.2.4 $d^j = \tilde{x}^j - x^j$ is a *descent* direction for f at x^j, and since $x^{j+1} \in \mathcal{S}(x^j, d^j)$ we have the desired result. ∎

From the preceding theorem it follows that $\{f(x^j)\}$, being a bounded sequence, converges. We now show that the limit of this sequence is the optimal value of (P) (we will assume that a full sequence $\{x^j\}$ is generated, otherwise the method terminates at an optimal solution).

Theorem 3.4.2 (Main Convergence Theorem). *The iterates* $\mathbf{x}^j$ *generated by algorithm 3.1.1, with* $\mathcal{P}$ *defined as in section 3.3, have the property that* $\{f(\mathbf{x}^j)\}_{j=1}^{\infty}$ *converges to the optimal value of the original problem (P) and every accumulation point of* $\{\mathbf{x}^j\}_{j=1}^{\infty}$ *is an optimal solution of (P).*

Proof: Let $\{\mathbf{x}^j\}_{j=1}^{\infty}$ be a sequence of iterates generated by algorithm 3.1.1, let $\{\tilde{\mathbf{x}}^j\}_{j=1}^{\infty}$ be the sequence produced by step 1 of the algorithm. The sequences $\{\mathbf{x}^j\}_{j=1}^{\infty}$, $\{\mathbf{x}^{j+1}\}_{j=1}^{\infty}$ and $\{\tilde{\mathbf{x}}^j\}_{j=1}^{\infty}$ are contained in $\boldsymbol{F}$, a compact set, so they have accumulation points in $\boldsymbol{F}$, say $\mathbf{x}^\star$, $\hat{\mathbf{x}}$ and $\tilde{\mathbf{x}}$ respectively. Without loss of generality there exists an index set $\boldsymbol{K} \subseteq \mathbf{N}$, such that

$$\mathbf{x}^j \xrightarrow{j\in K} \mathbf{x}^\star$$

$$\mathbf{x}^{j+1} \xrightarrow{j\in K} \hat{\mathbf{x}}$$

$$\tilde{\mathbf{x}}^j \xrightarrow{j\in K} \tilde{\mathbf{x}}$$

We claim that $\mathbf{x}^\star$ is an optimal solution of (P). Assuming $\mathbf{x}^\star$ is not an optimal solution of (P) we will get a contradiction. By lemma 3.2.3 $\mathbf{0}$ is *not* an optimal solution of $(\mathrm{SP}(\mathbf{x}^\star))$. Since $\Lambda^j \xrightarrow{j\in K} \mathbf{0}$, by theorem 3.3.4 we have that $\tilde{\mathbf{x}} - \mathbf{x}^\star$ is an optimal solution of $(\mathrm{SP}(\mathbf{x}^\star))$, hence $s_{\mathbf{x}^\star}(\tilde{\mathbf{x}} - \mathbf{x}^\star) < 0$. By lemma 3.2.4 $\mathbf{d}^\star := \tilde{\mathbf{x}} - \mathbf{x}^\star$ is a descent direction for f at $\mathbf{x}^\star$.

Now consider the problem

$$\begin{aligned} \min_{\theta} \quad & f(\mathbf{x}^\star + \theta \mathbf{d}^\star) \\ s.t. \quad & \mathbf{x}^\star + \theta \mathbf{d}^\star \in \boldsymbol{F} \\ & 0 \le \theta \le 1 \end{aligned} \tag{S'}$$

and let θ^* be an optimal solution of (S'). Defining $x_s := x^* + \theta^* d^*$ we have $f(x_s) < f(x^*)$. Now let $y^{j+1} = x^j + \theta^* d^j$ for d^j as defined in algorithm 3.1.1. Since $0 < \theta^* \leq 1$ we have $y^j \in F$ and $y^j \xrightarrow{j \in K} x_s$. Moreover $f(x^{j+1}) \leq f(y^j)$ because x^{j+1} minimizes f over the set $\{x \mid x = x^j + \theta d^j,\ x \in F,\ \theta \geq 0\}$. In the limit we then have $f(\hat{x}) \leq f(x_s) < f(x^*)$, contradicting $f(\hat{x}) = f(x^*)$.

So x^* is an optimal solution of (P). ∎

Note that the proof of the theorem goes through if $\mathcal{P}$ is any closed mapping with the property that, if x is not optimal for (P) and $y \in \mathcal{P}(x)$, then $y - x$ is a descent direction for f at x.

We now show how the proof of the main convergence theorem can be extended to the case in which an inexact line search is used. We chose a modified Goldstein test because of its analytical simplicity. The proof requires the additional assumption that f is continuously differentiable. Let us first define a modified Goldstein test which performs a line search within the feasible compact set F. We denote by $\mathcal{I}_G$ the set of acceptable stepsizes.

Modified Goldstein Test. *Let $x \in F$ and let d be a descent direction for f at x. For a given fixed ϵ, with $0 < \epsilon < \frac{1}{2}$, define*

$$\mathcal{A}(x,d) = \{\theta \mid \theta \geq 0,\ x + \theta d \in F \text{ and } \epsilon \leq \mathcal{Q}(\theta,x,d) \leq 1 - \epsilon\}$$

where

$$\mathcal{Q}(\theta,x,d) := \frac{f(x + \theta d) - f(x)}{\theta \nabla f(x) \cdot d}\ .$$

If $\mathcal{A}(x,d) \neq \emptyset$, then $\mathcal{I}_G := \mathcal{A}(x,d)$, else $\mathcal{I}_G := \{\theta^*\}$, where

$$\theta^* = \max\{\theta \mid x + \theta d \in F\}\ .$$

The corresponding search map is

$$S_G(x,d) = \{y \mid y = x + \theta d, \quad \theta \in I_G\}$$

If we use this test in algorithm 3.1.1, let us consider how the proof of theorem 3.4.2 may be extended. The first part is independent of the line search and thus the conclusion that $d^* = \tilde{x} - x^*$ is a descent direction is still true. It suffices to show that $f(\hat{x}) < f(x^*)$.

Let $j \in K$ and $x^{j+1} = x^j + \theta^j d^j$, and suppose $A(x^j, d^j) = \emptyset$. Then $\theta^j \geq 1$ since $x^j + d^j \in F$, and $Q(\theta^j, x^j, d^j) > 1 - \epsilon$, since $Q(\theta^j, x^j, d^j) < \epsilon$ would imply $A(x^j, d^j) \neq \emptyset$ because $Q(\theta, x^j, d^j) \geq \epsilon$ for θ sufficiently small. Thus, $Q(\theta^j, x^j, d^j) \geq \epsilon$ and

$$f(x^j + \theta^j d^j) - f(x^j) \leq \theta^j \epsilon \nabla f(x^j) \cdot d^j \tag{1}$$

On the other hand, if $A(x^j, d^j) \neq \emptyset$ then θ^j clearly satisfies (1). We thus conclude that (1) is satisfied for all $j \in K$. Now since $x^{j+1} = x^j + \theta^j d^j$ we can write

$$\theta^j = \frac{||x^{j+1} - x^j||}{||d^j||} .$$

If we define

$$\bar{\theta} := \frac{||\hat{x} - x^*||}{||d^*||}$$

we have $\theta^j \xrightarrow{j \in K} \bar{\theta}$. Taking the limit in (1), we get

$$f(\hat{x}) \leq f(x^*) + \epsilon \bar{\theta} \nabla f(x^*) \cdot d^* \tag{2}$$

Note that $\bar{\theta} > 0$, since $\theta^j \xrightarrow{j\in K} 0$ implies $\mathcal{A}(\mathrm{x}^j, \mathrm{d}^j) \neq \emptyset$ for j large enough and $\mathcal{Q}(\theta^j, \mathrm{x}^j, \mathrm{d}^j) \xrightarrow{j\in K} 1$, contradicting $\mathcal{Q}(\theta^j, \mathrm{x}^j, \mathrm{d}^j) \leq 1 - \epsilon$. Then from (2), since $\nabla f(\mathrm{x}^\star) \cdot \mathrm{d}^\star < 0$, we have $f(\hat{\mathrm{x}}) < f(\mathrm{x}^\star)$. ∎

To obtain some insights with respect to the convergence rate of the algorithm, we regard the problem as having only equality constraints, since near the solution, the set of active inequality constraints will hopefully remain the same, in which case we can consider them as being equality constraints. Let us also assume that f is a quadratic function. The problem (P) then becomes

$$\begin{aligned} \min \ & \frac{1}{2}\mathrm{x}^T\mathrm{Q}\mathrm{x} + \mathrm{c}^T\mathrm{x} \\ s.t. \ & \mathrm{A}'\mathrm{x} = \mathrm{b}' \end{aligned} \tag{P}$$

where Q is a symmetric positive definite matrix. Let us consider an idealized algorithm consisting of solving the underlying separable problem *exactly* at each iteration and using the direction obtained as a search direction for finding an improved solution for (P). The stepsize is then obtained by minimizing the objective function of (P) along the search direction. Given an iterate x^j, letting $\mathrm{D} = diag(\mathrm{Q})$, the underlying separable function centered at x^j is given by

$$\begin{aligned} s_{\mathrm{x}^j}(\mathrm{z}) &= \sum_{i=1}^{n} h_{\mathrm{x}^j}(z_i\mathrm{e}^i) \\ &= \sum_{i=1}^{n} \frac{1}{2}(z_i\mathrm{e}^i + \mathrm{x}^j)^T\mathrm{Q}(z_i\mathrm{e}^i + \mathrm{x}^j) + \mathrm{c}^T(z_i\mathrm{e}^i + \mathrm{x}^j) - \frac{1}{2}\mathrm{x}^{jT}\mathrm{Q}\mathrm{x}^j - \mathrm{c}^T\mathrm{x}^j \\ &= \sum_{i=1}^{n} \frac{1}{2}(z_i\mathrm{e}^i)^T\mathrm{Q}(z_i\mathrm{e}^i) + (\mathrm{x}^{jT}\mathrm{Q} + \mathrm{c}^T)(z_i\mathrm{e}^i) \\ &= \frac{1}{2}\mathrm{z}^T\mathrm{D}\mathrm{z} + (\mathrm{x}^{jT}\mathrm{Q} + \mathrm{c}^T)\mathrm{z} \end{aligned}$$

where $\mathbf{z} = \mathbf{x} - \mathbf{x}^j$. The underlying separable problem then becomes

$$\begin{aligned} \min \ & \frac{1}{2}\mathbf{z}^T\mathbf{D}\mathbf{z} + (\mathbf{Q}\mathbf{x}^j + \mathbf{c})^T\mathbf{z} \\ s.t. \ & \mathbf{A}'\mathbf{z} = \mathbf{0} \ . \end{aligned}$$

On the other hand, (P) is equivalent to the problem

$$\begin{aligned} \min \ & \frac{1}{2}\mathbf{z}^T\mathbf{Q}\mathbf{z} + (\mathbf{Q}\mathbf{x}^j + \mathbf{c})^T\mathbf{z} \\ s.t. \ & \mathbf{A}'\mathbf{z} = \mathbf{0} \ . \end{aligned}$$

Now let $\mathbf{Z}$ be the matrix whose columns form a basis for the null space of $\mathbf{A}'$. Then if we let $\mathbf{z} = \mathbf{Z}\mathbf{y}$, the latter problem becomes the unconstrained problem

$$\min \ \frac{1}{2}\mathbf{y}^T(\mathbf{Z}^T\mathbf{Q}\mathbf{Z})\mathbf{y} + (\mathbf{Q}\mathbf{x}^j + \mathbf{c})^T\mathbf{Z}\mathbf{y} \qquad \text{(QP)}$$

and the underlying separable problem is transformed into

$$\min \ \frac{1}{2}\mathbf{y}^T(\mathbf{Z}^T\mathbf{D}\mathbf{Z})\mathbf{y} + (\mathbf{Q}\mathbf{x}^j + \mathbf{c})^T\mathbf{Z}\mathbf{y} \qquad \text{(SQP)}$$

The solution to (SQP) is obtained by solving the system

$$(\mathbf{Z}^T\mathbf{D}\mathbf{Z})\mathbf{y} = -\mathbf{Z}^T(\mathbf{Q}\mathbf{x}^j + \mathbf{c})$$

and since the gradient of the objective function in (QP) evaluated at $\mathbf{y} = \mathbf{0}$ is exactly the negative of the right hand side, the procedure would amount to an approximate Newton method using the matrix $\mathbf{Z}^T\mathbf{D}\mathbf{Z}$ as an approximation to the true Hessian $\mathbf{Z}^T\mathbf{Q}\mathbf{Z}$. The rate of convergence of such a method is given by

the canonical rate of the eigenvalues of the matrix $(\mathbf{Z}^T\mathbf{D}\mathbf{Z})^{-1}\mathbf{Z}^T\mathbf{Q}\mathbf{Z}$, namely the objective values of the iterates will converge linearly with a ratio of $[\frac{(A-a)}{(A+a)}]^2$, where A and a are, respectively, the largest and smallest eigenvalues of such matrix (see, e.g., [Luenberger 84]).

Now, since for sufficiently small gridsizes the piecewise-linear approximation is close to the underlying separable approximation, the solutions of these problems are also close and the convergence rate of our method should be similar to the one for the idealized procedure described above.

CHAPTER 4

OPTIMIZATION ON THE CRYSTAL MULTICOMPUTER

In this chapter we discuss in detail the implementation of algorithm 3.1.1 in parallel on the CRYSTAL multicomputer, in the case of (P) being a nonlinear multicommodity network problem with coupling between commodities only in the objective function. In section 4.1 we describe how the algorithm can be specialized when (P) has this particular structure. Section 4.2 deals with details of the CRYSTAL multicomputer and the particular software used. Finally, in section 4.3 we briefly describe how the algorithm is implemented on CRYSTAL.

4.1. PARALLEL ALGORITHMS FOR NONLINEAR NETWORKS

As previously mentioned, we are concerned with the particular class of problems in which the feasible set F is given by network constraints plus upper and lower bounds on the variables. Problem (P) then becomes:

$$\begin{aligned} &\min f(\mathbf{x}) \\ &s.t.\ \mathbf{A}\mathbf{x} = \mathbf{b} \\ &\mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned} \qquad \text{(NNP)}$$

where $\mathbf{x}$ is an $n-$ vector of flows and $\mathbf{A}$ is a node-arc incidence matrix.

Within this large class of problems we find the collection of problems of traffic routing, equilibrium, and network design in computer and urban transportation networks. These problems are typically multi-commodity problems. There are two fundamental types of traffic assignment problems: symmetric problems in which the congestion on a given link is determined by the total flow summed over all commodities in that link only, giving rise to a symmetric Jacobian matrix in the corresponding variational problem, and asymmetric problems in which the congestion on a link depends on the total flows in several links. It is well known [Steenbrink 74] that the former problem is equivalent to a convex optimization problem under relatively weak assumptions, whereas the latter gives rise to a variational inequality that has a more complex optimization formulation involving a non-differentiable objective function. In this chapter we will concentrate on the former problem, although some of the decomposition ideas

considered apply equally well to both problem classes. This problem is of the form (NNP), where the objective function represents total congestion over all of the links of the network, and the constraints represent the supply, demand, and conservation constraints for each of the individual commodities flowing through the network. The number of commodities may be quite large because, depending on the formulation, there can be a commodity corresponding to each node or to each origin-destination pair. However, in the urban traffic equilibrium problem it is typically the case that the only coupling between different commodities occurs in the objective function. Because of this property, the approximation of the objective function by a separable function leads to a decomposition of the problem into separate optimization problems, one for each commodity. These single commodity problems may then be optimized in parallel (many algorithms for the asymmetric problem also contain a phase in which the objective function is replaced by a linear approximation so that the same decomposition may be used).

While the traffic assignment problem has been the subject of investigation by numerous researchers in recent years, and many algorithms have been proposed and tested for its solution, in almost all cases these algorithms lead to a decomposition of the problem into smaller subproblems, and thus are suitable candidates for research on the use of distributed systems.

We consider now the application of algorithm 3.1.1 to the problem (NNP). Given the iterate $\mathbf{x}^j$ obtained in iteration j the piecewise-linear approximating problem becomes:

$$\min_{\mathbf{x}} \sum_{i=1}^{n} \tilde{f}_i(\mathbf{x}^j, \Lambda^j, x_i)$$
$$s.t.\ \mathbf{A}\mathbf{x} = \mathbf{0} \qquad (\mathrm{AP}(\mathbf{x}^j, \Lambda^j))$$
$$\mathbf{l} \leq \mathbf{x} + \mathbf{x}^j \leq \mathbf{u}\ .$$

The problem $(\mathrm{AP}(\mathbf{x}^j, \Lambda^j))$ has two key features: 1) because the objective function is now separable, the problem may be decomposed into K separate optimization problems, where K is the number of commodities in the original problem; and 2) because of the convexity of the original objective function, each of these new problems is equivalent to a linear network optimization problem.

Specifically, assume that $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_K)$, where $\mathbf{x}_k$ is the vector of arcs corresponding to commodity k, and similarly $\mathbf{x}^j = (\mathbf{x}_1^j, \ldots, \mathbf{x}_K^j)$, $\mathbf{l} = (\mathbf{l}_1, \ldots, \mathbf{l}_K)$ and $\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_K)$. We assume there is no coupling between commodities in the constraints so we can write

$$A = \begin{pmatrix} A_1 & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & A_2 & \ldots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \ldots & A_K \end{pmatrix}$$

where $\mathbf{A}_k$ is the node-arc incidence matrix corresponding to commodity k for each k. The problem $(\mathrm{AP}(\mathbf{x}^j, \Lambda^j))$ may then be decomposed into a set of subproblems, one for each commodity, as follows. Let S_k be the subset of $\{1, \ldots, n\}$ corresponding to commodity k, then the corresponding subproblem is:

$$\min \sum_{i \in S_k} \check{f}_i(\mathbf{x}^j, \Lambda^j, x_i)$$

$$s.t.\ \mathbf{A}_k \mathbf{x}_k = \mathbf{0} \qquad (\mathrm{AP}_k(\mathbf{x}^j, \Lambda^j))$$

$$\mathbf{l}_k \leq \mathbf{x}_k + \mathbf{x}_k^j \leq \mathbf{u}_k \ .$$

Now step 2 of the procedure $\mathcal{P}$ described in 3.3 consists of solving *in parallel* the set of subproblems $(\mathrm{AP}_k(\mathbf{x}^j, \Lambda^j))$ for $k = 1, \ldots, K$, and since the objective function in each of the subproblems is piecewise-linear and convex, equivalent linear network subproblems can be solved via very fast linear network codes. For our implementation, a modification of the RNET package [Grigoriardis and Hsu 79] was used.

Letting $\tilde{\mathbf{y}}_1, \ldots, \tilde{\mathbf{y}}_K$ be the solutions of these subproblems, we set $\tilde{\mathbf{y}} = (\tilde{\mathbf{y}}_1, \ldots, \tilde{\mathbf{y}}_K)$ and $\mathcal{P}$ returns $\tilde{\mathbf{x}}^j = \tilde{\mathbf{y}} + \mathbf{x}^j$. So at each iteration the bulk of the work in $\mathcal{P}$ is done in parallel and a reduction in total real time is achieved as described in chapter 5.

4.2. THE CRYSTAL MULTICOMPUTER

CRYSTAL is a set of VAX-11/750 computers (currently there are 20) with 2 megabytes of memory each, connected by a 10 megabit/sec Proteon ProNet token ring. It can be used simultaneously by multiple research projects through partitioning the available processors according to the requirements of each project.

This partitioning is done via the software, and, once a user has acquired a "partition" or subset of processors, the user then has exclusive access to the node machines of that partition. CRYSTAL software is written in a local extension to Modula. Researchers can employ the CRYSTAL multicomputer in a number of ways. Projects that need direct control of processor resources can be implemented using a reliable communication service (the "nugget") [Cook, et al, 83] that resides on each node processor. Projects that prefer a higher-level interface can be implemented using the Charlotte distributed operating system. The Charlotte kernel provides multiprocessing, inter-process communication, and mechanisms for scheduling, store allocation, and migration. There is also a package called the **simple-application package**, which is a set of routines allowing application programmers to use the nugget for communication at a high level. Versions of this package are available for projects using Fortran, Modula, Pascal and C. Our implementation of the algorithm makes use of this set of routines. Development, debugging, and execution of projects takes place remotely through any of several VAX-11/780 hosts running Berkeley Unix 4.2. Acquiring a partition of node machines, resetting each node of the partition, and then loading an application onto each node may be performed interactively from any host machine. CRYSTAL has been used for research in a variety of areas, including distributed operating systems, programming languages for distributed systems, tools for debugging distributed systems, multiprocessor database machines, protocols for high performance local network communications, and numerical methods. Future plans for CRYSTAL include an increase in the number of node machines to approximately 40 and an upgrade of the communications medium to an 80 megabit/sec token ring.

The first step for running an application program on CRYSTAL is to obtain a partition of node machines. This is done via the **Nuggetmaster Command Interpreter** (NCI) which allows the user to send commands to the nuggetmaster. These commands include `show`, `new`, `link`, `load`, `state` and `free`. The `show` command shows the status of the network of node machines including which nodes are free for use, current existing partitions and information about the condition of each machine. The `new` command is used to obtain or reserve partitions according to availability. The `link` command links the specified code to the nugget in each specified node while the `load` command actually downloads the linked code to the specified node machines. The `state` command gives information about the status of each node in the partition for purposes of debugging. Finally, the `free` command deallocates the specified partitions. In the next section we give a sample run of NCI.

We now describe the use of the Fortran version of the simple-application package. There are two sets of routines in the package, the routines used for communication by the host machine and the ones used by the node machines. The main difference between the two is that, in the host machine, receiving and sending of messages is a *blocking* procedure (i.e., the program has to wait until the message is read or delivered to proceed with execution).

For the program to be used in the host machine, the user should provide some extra variables as described below. Five integer variables `numrecbufs`, `numsendbufs`, `buflength`, `parid` and `parsiz`, representing respectively the number of input and output buffers, the length of these buffers, the name of the partition and the number of node machines in it. A two-dimensional real array `recbfs`

of size buflength × numrecbufs is used as a pool of buffers to store incoming data. A two-dimensional real array senbfs of size buflength × numsendbufs is used as the corresponding pool of buffers for outgoing data. A one-dimensional boolean array iflags of size numrecbufs which specifies if messages have arrived in recbfs (iflags(i)=true means a message has arrived in recbfs(*,i)). A one-dimensional boolean array oflags of size numsendbufs specifying if messages in senbfs have been sent (oflags(i)=true means a message in senbfs(*,i) is still being sent). A one-dimensional integer array source of size numrecbufs which denotes the sources (machine numbers) of the corresponding messages in recbfs. Finally, a one-dimensional integer array dest of size numsendbufs indicating the destinations for messages in senbfs. For the programs used by the node machines the same variables are needed with the exception of parid which is replaced by mchnum, representing the number of the node machine the program is in. Note that the buffer arrays are of type real. In order for users to work with double precision data, a viable alternative is to define two double precision arrays decbfs and denbfs of sizes buflength/2 × numrecbufs and buflength/2 × numsendbufs, respectively, and "equivalence" them with the corresponding real arrays. We used this approach to handle our double precision data. It is important to note that the simple-application package allows a maximum of 2000 bytes per buffer, this implies that the maximum value that buflength can take is 500. Some of our double precision arrays have dimensions greater than 250, so we have to send them partitioned into different buffers (so far just two).

The first routine that should be called is mkbufr. This routine should be called only once. It has as arguments all the variables mentioned in the previous

paragraph, and informs the library routines these variables are the ones to be used for communication purposes. In the host machine, it returns the partition size in `parsiz`, whereas in the node machines it returns the machine number in `mchnum`.

For the process of sending information the simple-application package provides the routine `sendbf` which has as argument an integer `nextob` representing the number of the buffer to be sent. This routine sends the data contained in `senbfs(*,nextob)` to the destination specified in `dest(nextob)`. A side effect is to set `oflags(nextob) = true` until the message is safely delivered. In the host machine, this procedure blocks processing until the send is performed. A user first loads the data into the first available output buffer, sets the destination of the message by modifying the array `dest` accordingly, and finally calls `sendbf`. In the host library, the routine `recbf` is used to receive data from the node machines. This routine puts arriving messages into `recbfs(*,nextib)`, where `nextib` is the number of the next available input buffer. Note that `recbf` is not required in the node library, since the communication is non-blocking on the node machines. After the data is cleared the user should call `frbufr(nextib)` which frees buffer number `nextib` so it can be used by latter receives. This is done by setting `iflags(nextib)=false`.

The routine `run` is also provided in the host library so execution can be synchronized. This routine works in conjunction with the `pause` routine of the node interface library. Nodes can be put in pausing state, by calling `pause`, until the host program calls `run` so they can start processing.

4.3. IMPLEMENTATION ON CRYSTAL

We now discuss the details of the parallel implementation of our algorithm on CRYSTAL. We consider in this section only the case in which the number of commodities in the network problem is not greater than the number of available processors (the case in which the number of commodities is greater is discussed in chapter 6). The general structure of the program is the following:

[1] Input program data.

[2] Until optimality repeat:

  [a] Solve network subproblems (in parallel).

  [b] Perform line search to obtain new iterate.

[3] Output final solution data.

Since I/O processes on the node machines were not fully implemented at the time we performed this research, we use a **host program**, which resides in the host machine (a VAX 11/780), for stages [1] and [3]. Our approach for the node machines involves two other programs, a **master program** which resides on node machine 1 and a **slave program** which is used by node machines 2 through K, K being the number of commodities. The master program, apart from handling communication with the host and slaves and performing [2][a] for

its particular subproblem, performs the line search at each iteration. The slave program, working in each of the slave nodes, just receives data from the master node and solves its particular subproblem, sending the solution data back to the master.

Note that other approaches can also be implemented. For example, if availability of processors is not a restriction, one could work with $K+1$ node machines and use node $K+1$ exclusively for step [2][b], leaving the optimization of the subproblems to nodes 1 through K.

The host program is then structured as follows:

```
1   Program Host
2       Input problem data.
3       Call run to start node programs.
4       Send data to master node.
5       Receive final solution from master.
6       Print solution and timing data.
7   end.
```

where between steps 5 and 6 the host program remains idle until the process of optimization at the nodes is finished and then receives the solution and timing data from the master.

The master program has the following structure:

```
1   Program Master
2       Pause until host calls run.
3       Receive data from host.
4       for i := 2 until K do
5           Send parameter data to node i.
6       while true do
7           for i := 2 until K do
8               Send new data to node i.
9           Optimize subproblem 1.
10          for i := 2 until K do
11              Receive new partial solutions.
12          Perform line search.
13          if optimal or maxiter then
14              Send stopsignal to slave nodes.
15              go to 21.
16          end if
17      end while
18      Send final solution to host.
19  end
```

In step 6, the master node sends to each slave node the variables defining the structure of the network and the parameters to be used throughout the optimization process, whereas in step 10, it sends just the segment of the stepsize and iterate vectors to be used in the next iteration for the corresponding sub-

problem. The test performed in step 15 determines if the solution is optimal or if the maximum number of major iterations has been reached, in either case the slaves are advised to stop processing and the current solution is sent to the host.

For nodes $k = 2$ through $k = K$ each slave program looks as follows:

```
1   Program Slave k
2       Pause until host calls run.
3       Receive parameter data from master.
4       while true do
5           Receive new data from master.
6           if stopsignal then go to 12.
7           Optimize subproblem k.
8           Send new partial solution to master.
9       end while
10      Send timings to host.
11  end
```

Figure 4.3.1 shows an schematic description of how this programs are synchronized.

After these programs are compiled on the host machine using the f77 Unix Fortran compiler, they should be linked to the simple-application library and any other required libraries (e.g., Fortran and C libraries). The resulting executable files corresponding to the node programs are then ready to be linked and loaded onto the respective node machines. This, as discussed in the previous section, is done via NCI. Figures 4.3.2 and 4.3.3 show a sample run of NCI.

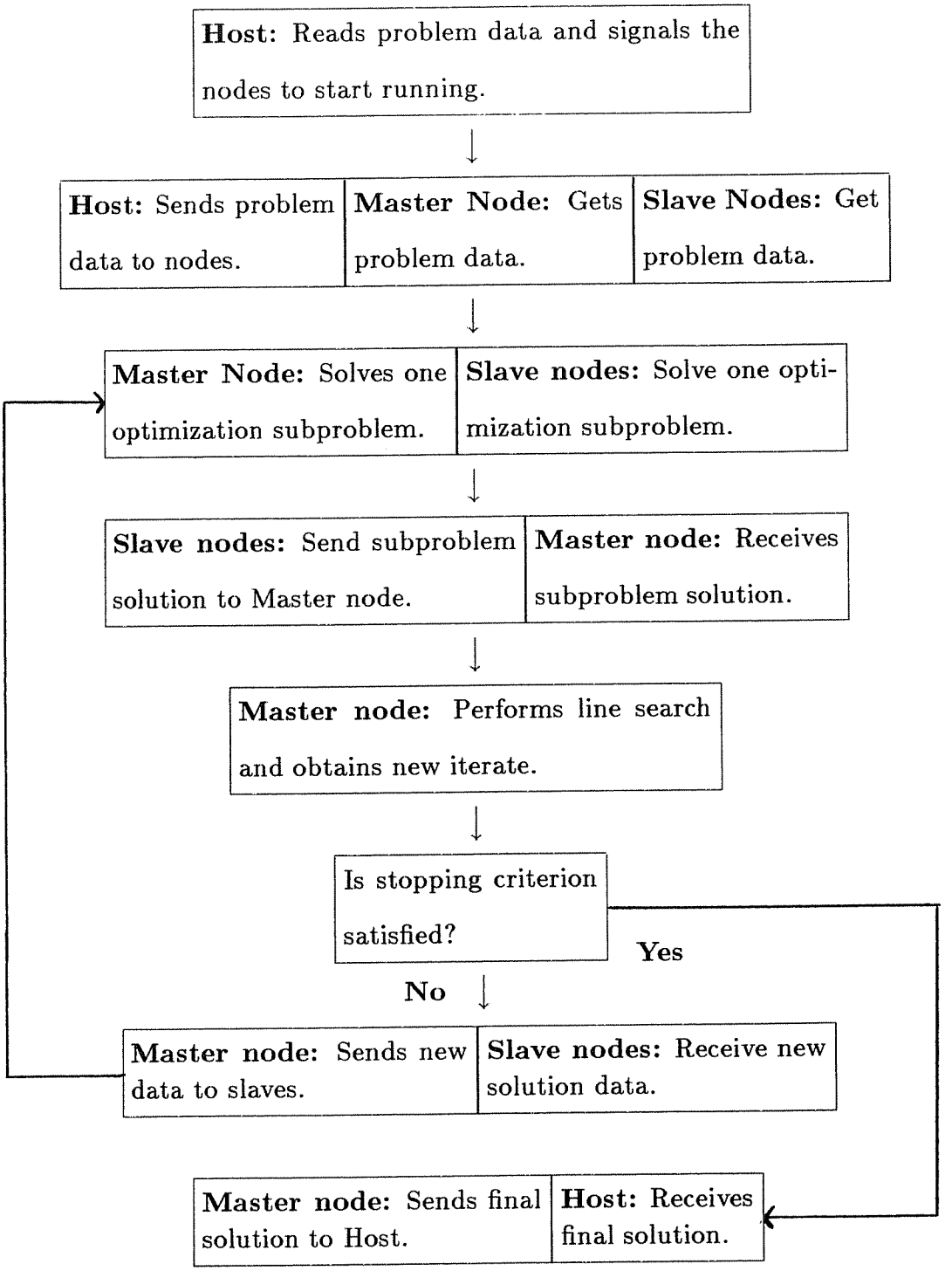

Fig. 4.3.1: Scheme for the distributed algorithm

```
% nci

Initializing NCI version 4.1  .  .  .

   Nuggetmaster Command Interpreter

NCI: new 4

Partition 1 has been created and targeted.
Targets:    partition 1 nodes 1,2,3,4

NCI: show

adr vax Mb ttys class state   nug pri  part owner
-------------------------------------------------
62  750 2  1    user  accept  reg 5    1.1  bernar
61  750 2  1    user  accept  reg 5    1.2  bernar
60  750 2  1    user  accept  reg 5    1.3  bernar
59  750 2  1    user  accept  reg 5    1.4  bernar

NCI: link ../dirmas/probo3.o -l -n 1

partition 1:
node 1 ...  linked OK ...  loaded OK

NCI: link ../dirsla/probo3.o -l -n 2 3 4

partition 1:
node 2 ...  linked OK ...  loaded OK
node 3 ...  linked OK ...  loaded OK
node 4 ...  linked OK ...  loaded OK
```

Fig. 4.3.2: Sample run of NCI (1)

```
NCI: state

partition 1:
node 1 ...  PAUSING
node 2 ...  PAUSING
node 3 ...  PAUSING
node 4 ...  PAUSING

NCI: bye

warning:    a retained partition exists

% probo3.o <  datp3 >  outfile3

        .
        .       (* Programs running, see Figs.  4.3.4 - 4.3.7 *)
        .

% nci

Initializing NCI version 4.1  .  .  .
A preexisting partition has been targeted.
Targets:    partition 1 nodes 1,2,3,4

   Nuggetmaster Command Interpreter

NCI: free

partition 1 has been deallocated

NCI: bye

%
```

Fig. 4.3.3: Sample run of NCI (2)

The sample run starts by calling NCI and using the new command to get a new partition of 4 node machines. The show command then allows us to check on the status of the partition. In the case shown, we were given node machines with logical addresses 59, 60, 61 and 62. Each of them has 2 Megabytes of memory and 1 terminal. The nugget installed is the regular nugget (as opposed to an experimental one) and it is in `accept` state, that is waiting for an application to be loaded. The partition has a default priority of 5 and the owner of the partition is also shown.

Afterwards, we link and load the executable file corresponding to the master node program into node 1. Instead of using the `load` command to load it, we use the `-l` flag of the `link` command which allows us to link and load the executable file in one shot. The nodes where the load will be done are indicated after the `-n` flag. Then we carry out a similar process for the slave node programs for nodes 2, 3 and 4. A check on the state of the partition reveals the nodes are in pausing state (i.e., waiting for the host to send a signal, so they can start work simultaneously). The next step is to exit NCI temporarily and, back in the system, start running the host program. After the program finishes, we go back to NCI, free the partition and exit.

In figures 4.3.4 through 4.3.7 we give samples of communication code for different cases.

For legibility reasons, we have omitted the segments of code that perform the timing of each process. Examples of complete communication routines are given in the appendix. In figure 4.3.4 we have an example of how the host sends

```
      .
      .
C**** Wait for the next output buffer to become available.
C
 200  IF(OFLAGS(NEXTOB)) GOTO 200
C
C**** Load data on array DENBFS(*,NEXTOB)
C**** corresponding to output buffer number NEXTOB.
C
      DO 1 I=1,NARCS
           DENBFS(I,NEXTOB)=FROM(I)
 1    CONTINUE
C
C**** Set destination, in this case node 1 (master node).
C
      DEST(NEXTOB)=1
C
C**** SENDBF sends buffer NEXTOB.
C
      CALL SENDBF(NEXTOB)
C
C**** Increment the output buffer counter by 1.
C
      NEXTOB=NEXTOB+1
      IF(NEXTOB.GT.NUMSENDBUFS) NEXTOB=1
      .
      .
```

Fig. 4.3.4: Sending data from the host

```
      .
      .
C**** Receive message into next available buffer.
C
      CALL RECBF
C
C**** Send message to master node, host is ready.
C
 300  IF(OFLAGS(NEXTOB)) GOTO 300
      DEST(NEXTOB)=1
      CALL SENDBF(NEXTOB)
C
C**** Receive message into next available buffer.
C
      CALL RECBF
      .
      .
C**** Read information from array DECBFS(*,NEXTIB).
C
      DO 1 I=1,NUMNET*NARCS/2
            XNEW(I)=DECBFS(I,NEXTIB)
 1    CONTINUE
C
C**** Free buffer NEXTIB for subsequent use.
C
      CALL FRBUFR(NEXTIB)
      NEXTIB=NEXTIB+1
      IF(NEXTIB.GT.NUMRECBUFS) NEXTIB=1
      .
      .
```

Fig. 4.3.5: Receiving data at the host

```
      .
      .
C
C**** Check that data has been received.
C
 300  IF(.NOT.IFLAGS(NEXTIB)) GOTO 300
C
C**** Read data from array DECBFS(*,NEXTIB)
C**** corresponding to input buffer NEXTIB.
C
      DO 2 I=1,NARCS
            LB(I)=DECBFS(I,NEXTIB)
 2    CONTINUE
C
C**** Free buffer for further use and increment input
C**** buffer count.
C
      CALL FRBUFR(NEXTIB)
      NEXTIB=NEXTIB+1
      IF(NEXTIB.GT.NUMRECBUFS) NEXTIB=1
      .
      .
```

Fig. 4.3.6: Receiving data at the nodes

```
      .
      .
C**** Wait for next output buffer to become ready.
C
 100  IF(OFLAGS(NEXTOB)) GOTO 100
C
C**** Put data in array DENBFS(*,NEXTOB) corresponding
C**** to next output buffer
C
      DENBFS(1,NEXTOB)=RTN(4)
      DENBFS(2,NEXTOB)=RTN(5)
      DO 1 I=1,NARCS
            DENBFS(I+2,NEXTOB)=XNEW(I)
 1    CONTINUE
C
C**** Set destination for the buffer, send it calling
C**** SENDBF and increment output buffer counter.
C
      DEST(NEXTOB)=1
      CALL SENDBF(NEXTOB)
      NEXTOB=NEXTOB+1
      IF(NEXTOB.GT.NUMSENDBUFS) NEXTOB=1
      .
      .
```

Fig. 4.3.7: Sending data from the nodes

data to the master node. The first step is to check if the next available output buffer nextob is free to use by succesively checking the busy flag on it. Once we have an output buffer for use, we load some data into it. In this case the array from, which defines the tails of the arcs in the network, is loaded into the output buffer denbfs(*,nextob). The destination is set and sendbf is called to send the information in denbfs(*,nextob) to node 1. After the message is safely delivered execution of the program resumes and the counter of available buffers nextob is incremented with the provision that if we exceed the number of available buffers, it should start again from 1.

In figure 4.3.5 we show how the host receives data from the nodes. First we should call recbf to perform a receive of the incoming message into the next available input buffer. Since this procedure is blocking, the master node has to wait for the host to acknowledge the arrival of the previous message. The host does this by sending an empty message to the master node. Then the next message is received and the process is repeated until all messages are received by the host. Finally, the information on the input buffers is processed. In the portion of code shown, the contents of the first input buffer decbfs(*,nextib) are the components of the first half of the solution vector xnew (storage restrictions force sending this vector in two parts). The input buffers are freed after the data is processed and the input buffer counter nextib is incremented. One could read the data from each input buffer right after its arrival but we tried to minimize the waiting time for the master node, so we first get all the buffers and then we read the data from them.

Figure 4.3.6 shows how the nodes receive data. Initially the busy flag on the next available input buffer is checked repeatedly until it indicates the arrival of a message. The data is read from the corresponding input buffer right after it arrives. In this case we are receiving the array of lower bounds on the flows of the network subproblem. The buffer is then freed for use and the counter incremented as before.

Finally, in figure 4.3.7 we describe how the nodes send data to each other. As usual we wait for the next available output buffer. Then we load the data to be sent in the buffer. In this example the first 2 entries of the buffer will contain information given by RNET, that is number of pivots and number of degenerate pivots. The rest is the segment of the new feasible solution corresponding to the subproblem that this particular node was handling. The destination is set to be the master node and `sendbf` is called. Once again the counter for output buffers is incremented.

In our tests we have used `numsendbufs` = `numrecbufs` = 13, since in the largest problem, the maximum number of buffers that can arrive at approximately the same time is 11 (coming from the slave nodes). It is always best to have some extra buffers, so we add two more for insurance. This setting avoids the possible delay in communication caused by trying to send or receive a message, but having no buffers available for the task. In the case of a message arriving at a node machine and not finding a suitable buffer, the delay could be up to a full second before a second try is made.

Regarding the timing of processes, we decided to divide the timing intervals into three types: 1) **Wait time**, time spent waiting for a message to arrive, which

is measured by making observations before and after a wait on either `iflags` or `oflags`, 2) **Communication time**, time spent processing incoming or outgoing data (i.e., loading or reading buffers), and 3) **Work time**, time spent actually doing algorithmic work. In the next chapter we discuss the timings obtained.

CHAPTER 5

COMPUTATIONAL IMPLEMENTATION

This chapter contains computational results, for the algorithm in its original form and also for the parallel implementation on CRYSTAL. In section 5.1 we discuss details of the implementation of the algorithm. Section 5.2 describes the problems tested. In section 5.3 we show the numerical results for the original algorithm and for the parallel implementation.

5.1. IMPLEMENTATION DETAILS

As previously mentioned, the algorithm is particularly useful in the case of the problem being a nonlinear network problem. In this case the resulting approximating problems can be transformed into linear network problems. This is done by introducing several new variables in place of the original variables without altering the network structure of the problem. In practice however, only two segments of the approximation are stored at a given time, and extra segments

are added as needed (for details see [Kamesam and Meyer 82]). It is worth noting that in order to maintain this two-segment approximation when a particular variable reaches an absolute bound, an imaginary arc is generated, with a very high linear cost, so the structure of the approximation is maintained.

After some computational experimentation with a uniform gridsize in which all segments, except the ones close to the bounds, were equal, we decided to implement a variable gridsize approach. For a given gridsize λ_i^j, this approach consists of having the first segment to the right of the current iterate being of size λ_i^j, the second of size $2\lambda_i^j$, the third of size $4\lambda_i^j$, and so on. The same is done for the segments to the left of the current iterate. This approach has resulted in a decrease in the amount of work needed for the optimization of the subproblems, and a corresponding worsening of the resulting descent direction. Nevertheless, the results indicate that this approach is of advantage when dealing with nonseparable problems.

Another issue of computational relevance is the factor α by which the gridsize is reduced at each iteration. In [Kamesam and Meyer 82] it is reported that for separable problems, their algorithm working with a value of $\alpha = 0.25$ yielded the best results. Our experience has shown that the best choice of α for nonseparable problems is more dependent on the problem being solved. For some problems a value of $\alpha = 0.25$ would cause large amounts of work and somewhat slow convergence. In section 5.3 we show the values of α used for the different test problems.

Since in general an initial feasible solution is not available, the first iterate

x^1 is generated by solving a problem of the form $AP(\bar{x}, \Lambda^0)$, where $\bar{x}$ is an arbitrary element of H and λ_i^0 is typically $\frac{(u_i - l_i)}{4}$.

As a stopping criterion we have used previously obtained solutions to each of the problems, stopping our algorithm whenever our solution value matched the best known solution value to the particular problem. The technique of producing a lower bound at each iteration via solving a Lagrangian relaxation problem and stopping when the lower bound and the current solution value are close enough is not computationally practical for our kind of test problems. Another alternative is solving a linearized problem at each iteration obtaining a lower bound. Our experience with this approach is that the resulting bound provides a reasonable good idea of how close we are to the optimal solution, but is not very accurate.

5.2. TEST PROBLEMS

Thus far, three test problems have been used. All are nonlinear multi-commodity network flow problems of the form of traffic equilibrium problems. The problems are of a relatively small size in which the number of commodities is not greater than the number of processors available on CRYSTAL.

Problem 1 was provided to us by Professor J. S. Pang [Pang 83]. The network corresponding to this problem consists of 20 nodes, 28 links and 2 commodities with corresponding origin-destination pairs, (1,15) and (1,20) both having a demand of 100 units. The equivalent optimization problem then has 56 variables

Origin	Dest.	Demand	Origin	Dest.	Demand
2	3	2000	3	2	200
2	4	2000	4	2	200
2	5	1000	5	2	100
3	4	1000	4	3	100
3	5	2000	5	3	200
4	5	1000	5	4	100

Table 5.1.1: Origin-Destination pairs for problem 3

and 40 constraints. The objective function for this problem is given by:

$$\sum_{k=1}^{28} c_k^1 f_k^5 + c_k^2 f_k$$

where f_k represents *total* flow on link k (i.e., the sum of the flows of commodities using that link), and $\mathbf{c}^1$ and $\mathbf{c}^2$ are given vectors of coefficients.

Problem 2 was taken from [Bertsekas and Gafni 82]. The network for this problem consists of 25 nodes, 40 links and 5 commodities. The origin-destination pairs are (1,4), (2,5), (3,1), (4,2), and (5,3) with demands 0.1, 0.2, 0.3, 0.4 and 0.5, respectively. After some cleaning up of the resulting optimization problem (i.e., removing arcs that cannot be used for a particular commodity) the problem is converted to one having 60 variables and 60 constraints. The objective function for the problem is:

$$\sum_{k=1}^{40} \frac{f_k^3}{3} + \frac{f_k^2}{2} + f_k$$

where f_k is as before.

Problem 3 comes from [Steenbrink 74]. The network for this problem has 9 nodes, 36 links and 12 commodities. Table 5.1.1 shows the origin-destination pairs and corresponding demands.

The resulting optimization problem has 432 variables and 108 constraints. The objective function is given by:

$$\sum_{k=1}^{36} c_k^1 f_k^2 + c_k^2 f_k \ .$$

5.3. NUMERICAL RESULTS

As previously mentioned, we use a modification of the RNET package [Grigoriardis and Hsu 79], for solving the linear network subproblems. The original RNET package was designed to have all its data of type integer. This version was modified by [Kamesam and Meyer 82] to allow the costs and dual variables to be double precision real variables and to implicitly handle the generation and dropping of segments of the approximation. We further modified it to allow for the vectors of variables, right hand sides and lower and upper bounds to be also double precision variables. This was necessary since the line search procedure in general yields real data, even having integer base point and direction vectors. We also implemented the variable gridsize strategy described in 5.1 by modifying the implicit generation of segments. The algorithms are coded entirely in Fortran and they were run using the f77 compiler with optimization running under the UNIX Operating System. The numerical results reported for the sequential version were obtained on a VAX 11/780. For comparison purposes, we also used the MINOS package to solve the test problems. In table 5.3.1 we show a restatement of the size and structure of the problems.

	Links	Nodes	Cmdts.	Cnstr.	Vars.
Problem 1	28	20	2	40	56
Problem 2	40	5	5	60	60
Problem 3	36	9	12	108	432

Table 5.3.1: Test Problems

Numerical results for our algorithm are shown on table 5.3.2. We give the number of major iterations of our algorithm, the total number of RNET pivots, CPU time for the algorithm and the objective function value obtained. The factor α by which the gridsize is reduced at each iteration is $\alpha = 0.70$ for problem 1, $\alpha = 0.25$ for problem 2 and $\alpha = 0.42$ for problem 3.

	Iter.	Pivots	CPU time	Obj. value
Problem 1	30	429	7.27 s.	1010540.6
Problem 2	11	346	4.81 s.	47.858538
Problem 3	11	2266	44.61 s.	16957.674

Table 5.3.2: Computational Results on VAX 11/780

The performance of MINOS is shown on table 5.3.3. We include total number of major iterations, CPU time and the objective function value obtained.

	Iter.	CPU time	Obj. value
Problem 1	32	7.71 s.	1010540.6
Problem 2	6	6.15 s.	47.858538
Problem 3	106	39.01 s.	16957.674

Table 5.3.3: MINOS results on VAX 11/780

We now discuss the numerical results obtained for the parallel implementation of our algorithm on CRYSTAL. Since the VAX 11/780 is a faster machine than the VAX 11/750's used as node machines, we implemented the sequential version of the algorithm so it can be run on CRYSTAL (i.e., using just one

node machine), and in this way we can get a realistic view of the advantages, in terms of real-time savings, of using a parallel approach. In table 5.3.4 we display the results obtained using both the parallel and the sequential versions of the algorithm.

Problem 1

	Parallel	Sequential
CPU time	13.0 s./3.6 s.	16.8 s.
Real time	15.2s.	17.2 s.

Problem 2

	Parallel	Sequential
CPU time	4.8 s./1.0 s.	8.5 s.
Real time	7.0 s.	8.9 s.

Problem 3

	Parallel	Sequential
CPU time	13.7./8.5 s.	111.0 s.
Real time	23.5 s.	111.8 s.

Table 5.3.4: Computing times on CRYSTAL

In the parallel case, two CPU time figures appear, the first represents the CPU time for the master node and the second an average of the CPU times for the slave nodes. The figure for the master node is higher than the one for

the slave nodes due to the fact that the master node performs the line search. We should also remark that, for these test problems, the CPU time for each individual slave node does not differ much from the average.

The differences between Real and CPU time for the master node are due to waiting and communication times. We performed timing tests to determine how these times were distributed. Table 5.3.5 shows the timing results for the master node and for each problem. The times shown represent, respectively, time working on subproblems, time performing search, waiting time and communication time as defined in 4.3 and the time given is the total over all iterations.

	Optim.	Search	Idle	Comm.
Problem 1	5.21 s.	7.84 s.	1.82 s.	0.39 s.
Problem 2	1.07 s.	3.74 s.	1.58 s.	0.59 s.
Problem 3	9.49 s.	4.18 s.	3.68 s.	6.31 s.

Table 5.3.5: Timings at the master node

Table 5.3.6 shows timing data for the slave nodes. The times shown are averages of the times for each individual slave node over all iterations.

	Optim.	Idle	Comm.
Problem 1	3.64 s.	11.09 s.	0.19 s.
Problem 2	1.00 s.	5.45 s.	0.08 s.
Problem 3	8.50 s.	14.31 s.	0.32 s.

Table 5.3.6: Timings at the slave nodes

Note that in some cases the time corresponding to the line search is greater than the optimization time. This fact indicates that improving the efficiency and parallelism of the line search is a subject which needs further work. The idle times for the master node are due to the fact that in some iterations the master node finishes its optimization before some slave nodes have done so, and hence it has to wait for them. The master node also has to wait for the host to be ready to receive messages, but our tests show that this time is small compared to time waiting for the slave nodes. Communication times for the master, as mentioned before, represent the time spent loading and unloading buffers and, as such, increase with the size of the problem.

The idle times for the slave nodes are due not only to the time they wait for the master node to perform the line search, but also to the same phenomenon of different optimization times, since the faster slave nodes also have to wait while other slave nodes are finishing their optimization procedures. Communication times for the slave nodes are determined by subproblem size rather than by overall problem size, as confirmed by our figures.

It is reasonable to expect that the communication times for the master node would be roughly $K - 1$ times the ones for the slave nodes, but this is not the case. We believe this is so because communication time includes basically loading or unloading of buffers and calls to the interface routines, and a call to `sendbf` is more time consuming that one to `frbufr`, plus the fact that the amount of data going from the master node to the slave nodes is larger than the amount going the other way. Also the simultaneous arrival of messages to the master node in-

terrupts execution of communication code, increasing communication time. This is of more relevance in the larger problems.

CHAPTER 6

DIRECTIONS FOR FURTHER RESEARCH

In this chapter we discuss directions for further research both in the theoretical and computational areas.

The discussion in chapter 3 for the linearly constrained quadratic case leads us to believe that near the optimal solution, the convergence rate of our piecewise-linear algorithm should be linear with a rate of approximately $[\frac{(A-a)}{(A+a)}]^2$, where A and a are, respectively, the largest and smallest eigenvalues of the matrix $(\mathbf{Z}^T\mathbf{D}\mathbf{Z})^{-1}\mathbf{Z}^T\mathbf{H}\mathbf{Z}$, where $\mathbf{H}$ represents the Hessian of the function at the current iterate, $\mathbf{D} = diag(\mathbf{H})$ and $\mathbf{Z}$ is the matrix whose columns form a basis for the null space of the matrix $\mathbf{A}'$. This would imply that our algorithm would behave better for problems in which the objective function is near separable, in the sense of having a Hessian with small off-diagonal entries relative to the diagonal elements. The work of Dembo on truncated Newton methods [Dembo 81] may be useful in this area.

Another interesting topic for further investigation is the use of Newton, quasi-Newton and variable metric methods to iteratively solve large-scale non-linear network problems, by using generalizations of SOR methods such as the ones proposed by [Cryer 71] and [Mangasarian 77, 81] for solving the quadratic subproblems. In the separable case, the Hessian of the objective function is diagonal, so these methods can be used to solve such subproblems. In the non-separable case, we could use an underlying separable function approach as the one described here and solve an approximating quadratic separable problem to obtain a descent search direction. In either case, some work is needed to establish the convergence properties of such a method and its computational effectiveness. On the other hand, it would be interesting to investigate the use of some other algorithms for solving the underlying separable network subproblems such as the one proposed by [Rockafellar 84]. For separable network problems, this algorithm, given a feasible point, produces a descent direction. Some research has to be done however, to establish its closedness.

Regarding the computational implementation of the algorithm, the problem of finding a suitable stopping criterion is still not fully solved. In the separable case a lower bound on the current objective value can be obtained from estimates of the error of approximation [Meyer 80]. A stopping criterion based on how close the current objective function value is to the corresponding lower bound can then be used. Another alternative is the use of a Lagrangian relaxation problem whose solution is a lower bound on the current objective value. In the separable case this problem can be solved by solving n one-dimensional problems [Kamesam and Meyer 82], and in some cases the solutions to these problems can

be obtained in closed form with very little computational effort. Unfortunately, in the nonseparable case we do not have nice estimates on the approximation error and, in general, the solution of a Lagrangian relaxation problem at each iteration could be a very expensive procedure, so these lower bound generation techniques are not immediatly applicable as bases for stopping criteria. The development of efficient ways of determining the degree of optimality of the current objective value is in order.

We now turn to the parallel implementation on CRYSTAL. One area that requires further investigation is the case of more commodities than processors. In this case several strategies may be implemented. One alternative would be to divide the set of single-commodity problems into smaller groups and have each node solve a fixed group of problems at each iteration. We suspect this static approach would cause significant differences between the amount of work each node has at each iteration and consequently increase idle times. To avoid this problem one could stop processing on all nodes at the same time, once one of them completes its optimization, and take the current solutions (perhaps nonoptimal, but certainly defining a descent direction). A new group of problems would then be sent to the slaves and the process repeated. The issue of how good these directions, arising from partial solutions, would be is a subject for further computational and theoretical study. Another alternative is to have a node exclusively for coordination and line search purposes. This node would send problems to the slaves and receive solutions from them. After all subproblems are finished, the master node performs the line search and the process is repeated. This approach has the advantage that the idle times for the slaves are reduced

because once a slave node is ready to start work, the master node provides a new subproblem to solve, except of course when there aren't any more. In all of these cases, there is still the problem that the slave nodes are idle while the master node is performing the line search. One alternative that partially minimizes this idle time is to have a coordination node that waits for a group of subproblems to be processed by the slaves and then performs a line search restricted to the corresponding subset of variables, while the slaves are optimizing the next group of subproblems. This approach leads to interesting mathematical questions in that it may be necessary to change the objective function of the optimization problems during or after the solution process as a result of the line search. It should be observed here that the line search plays a more critical role with regard to efficiency in the parallel approach than it does in the sequential implementation. Although it represents only a few percent of the total computing time, this small fraction will be significant if it is comparable to the total workload on each of the individual node machines and if the slave nodes are idle while this task is being performed. Work on making the line search as efficient and parallel as possible is needed to improve the overall efficiency of the algorithm. The most advanced alternative for the parallel implementation of the algorithm is the use of the Charlotte distributed operating systems, which automatically distributes work to processors and promises to provide the most efficient enviroment for such an application. Of course, much experimental work has to be done to determine which strategy would yield the best results.

APPENDIX

In this appendix we list the codes for some of the communication routines used in the CRYSTAL implementation of the algorithm. Listed are `horec1` and `hosend` which are routines for use at the host: the first one receives the final solution data from the master node and the other sends the problem data to the master node at the start of the algorithm. Four routines for use at the master node are listed. These are `marecv` which receives the problem data from the host, `masen1` which sends the final solution data to the host, `maslr1` which receives the solution of the subproblems from all slaves at each iteration, and `masls1` which sends the current base point and gridsize vectors to the slaves at each iteration. Two routines for use at the slave nodes are listed. These are `slrec1` which receives the current base point and gridsize vectors from the master node at each iteration, and `slsen1` which sends the subproblem solution to the master node at each iteration.

```
      subroutine horec1
c
c                          HOST ROUTINE
c*    This routine receives the final solution data
c*    from the master node.
c
c*       Inserting declarations file.
c
#include "hodec.h"
c
c*       Receiving the next message ...
c
      call recbf
c
c*       Incrementing the output buffer counter.
c*       (done after RECBF to prevent loss of message)
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
c*       Checking if buffer is available.
c
 300  if(oflags(nextob)) goto 300
c
c*       Setting the destination (1 = master node)
c
      dest(nextob)=1
c
c*       Sending empty message to master, host is ready
c*       to receive next message.
c
      call sendbf(nextob)
c
c*       Receiving next message ...
c
      call recbf
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 400  if(oflags(nextob)) goto 400
c
      dest(nextob)=1
```

```
c
      call sendbf(nextob)
c
      call recbf
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 500  if(oflags(nextob)) goto 500
c
      dest(nextob)=1
c
      call sendbf(nextob)
c
      call recbf
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
c*       Checking the flag on the input buffer
c*       (just a safeguard)
c
 100  if(.not.iflags(nextib)) goto 100
c
c*       Reading the first buffer containing the final
c*       objective value, the lower bound, the number
c*       of pivots and degenerate pivots, and the number
c*       of calls to SEARCH.
c
      newobj=decbfs(1,nextib)
      lbound=decbfs(2,nextib)
      mpivot=decbfs(3,nextib)
      npivot=decbfs(4,nextib)
      nsea=decbfs(5,nextib)
c
c*       Freeing the buffer and incrementing the
c*       buffer counter.
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
 200  if(.not.iflags(nextib)) goto 200
```

```
c
c*        ...  the first half of the final solution
c*        vector ...
c
      do 1 i=1,numnet*narcs/2
          xnew(i)=decbfs(i,nextib)
 1    continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
 210  if(.not.iflags(nextib)) goto 210
c
c*        ...  and the other half.
c
      do 11 i=1,numnet*narcs/2
          xnew(i+numnet*narcs/2)=decbfs(i,nextib)
 11   continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
c
      return
      end
```

```
      subroutine hosend
c
c                        HOST ROUTINE
c*    This routine sends the problem data to the
c*    master node.
c
c*       Inserting declarations file.
c
#include "hodec.h"
c
c*       Checking the flag on the next output buffer
c
 100  if(oflags(nextob)) goto 100
c
c*       Loading the first outpu buffer with the scalar
c*       parameters of the problem:  gridsize reduction
c*       factor, tolerance for optimality, pricing frequency,
c*       lower bound, number of nodes, number of arcs, number
c*       of separate networks, number of segments in the
c*       approximation and search parameter.
c
      denbfs(1,nextob)=alpha
      denbfs(2,nextob)=epslon
      denbfs(3,nextob)=frq
      denbfs(4,nextob)=lbound
      denbfs(5,nextob)=mnode
      denbfs(6,nextob)=narcs
      denbfs(7,nextob)=numnet
      denbfs(8,nextob)=segmnt
      denbfs(9,nextob)=sepam
c
c*       Setting the destination and sending the buffer.
c
      dest(nextob)=1
      call sendbf(nextob)
c
c*       Incrementing the buffer counter.
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 200  if(oflags(nextob)) goto 200
c
```

```
c*       Loading the array of tails of the network
c
      do 1 i=1,narcs
          denbfs(i,nextob)=from(i)
 1    continue
c
      dest(nextob)=1
      call sendbf(nextob)
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 300  if(oflags(nextob)) goto 300
c
c*       ...  half the vector of lower bounds on
c*       the variables ...
c
      do 2 i=1,numnet*narcs/2
          denbfs(i,nextob)=lb(i)
 2    continue
c
      dest(nextob)=1
      call sendbf(nextob)
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 310  if(oflags(nextob)) goto 310
c
c*       ...  and the other half.
c
      do 21 i=1,numnet*narcs/2
          denbfs(i,nextob)=lb(i+numnet*narcs/2)
 21   continue
c
      dest(nextob)=1
      call sendbf(nextob)
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 400  if(oflags(nextob)) goto 400
c
```

```
c*       The vector of right hand sides
c
      do 3 i=1,numnet*mnode
          denbfs(i,nextob)=rhs(i)
 3    continue
c
      dest(nextob)=1
      call sendbf(nextob)
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 500  if(oflags(nextob)) goto 500
c
c*       The vector of heads for the network.
c
      do 4 i=1,narcs
          denbfs(i,nextob)=to(i)
 4    continue
c
      dest(nextob)=1
      call sendbf(nextob)
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 600  if(oflags(nextob)) goto 600
c
c*       ...  half the vector of upper bounds on
c*       the variables.
c
      do 5 i=1,numnet*narcs/2
          denbfs(i,nextob)=ub(i)
 5    continue
c
      dest(nextob)=1
      call sendbf(nextob)
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 610  if(oflags(nextob)) goto 610
c
```

```
c*        ...  and the other half.
c
      do 51 i=1,numnet*narcs/2
          denbfs(i,nextob)=ub(i+numnet*narcs/2)
 51   continue
c
      dest(nextob)=1
      call sendbf(nextob)
c
c
      return
      end
```

```
      subroutine marecv
c
c                        MASTER ROUTINE
c*    This routine receives the problem data from
c*    the host machine.
c
c*       Including the declarations file.
c
#include "madec.h"
c
      real*8 btime
c
c*       Timing ...
c
      tmold=btime(0)
c
c*       Waiting for the next available buffer.
c
 100  if(.not.iflags(nextib)) goto 100
c
c*       Time observation.
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       Reading the buffer containing the parameters
c*       for the problem.
c
      alpha=decbfs(1,nextib)
      epslon=decbfs(2,nextib)
      frq=decbfs(3,nextib)
      lbound=decbfs(4,nextib)
      mnode=decbfs(5,nextib)
      narcs=decbfs(6,nextib)
      numnet=decbfs(7,nextib)
      segmnt=decbfs(8,nextib)
      sepam=decbfs(9,nextib)
c
c*       Freeing the buffer for further use and
c*       incrementing the input buffer counter.
c
      call frbufr(nextib)
      nextib=nextib+1
```

```
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
 200  if(.not.iflags(nextib)) goto 200
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       Reading the vector of tails for the network.
c
      do 1 i=1,narcs
          from(i)=decbfs(i,nextib)
 1    continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
 300  if(.not.iflags(nextib)) goto 300
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       Reading first half of vector of lower bounds.
c
      do 2 i=1,numnet*narcs/2
          lb(i)=decbfs(i,nextib)
 2    continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
 310  if(.not.iflags(nextib)) goto 310
c
```

```
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       ...  and the other half.
c
      do 21 i=1,numnet*narcs/2
          lb(i+numnet*narcs/2)=decbfs(i,nextib)
 21   continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
 400  if(.not.iflags(nextib)) goto 400
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       Reading vector of right hand sides.
c
      do 3 i=1,numnet*mnode
          rhs(i)=decbfs(i,nextib)
 3    continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
 500  if(.not.iflags(nextib)) goto 500
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       Reading vector of heads for the network.
c
      do 4 i=1,narcs
          to(i)=decbfs(i,nextib)
```

```
 4     continue
c
       call frbufr(nextib)
       nextib=nextib+1
       if(nextib.gt.NUMRECBUFS) nextib=1
c
       tmold=btime(0)
       tmcomh=tmcomh+tmold-tmtemp
c
 600   if(.not.iflags(nextib)) goto 600
c
       tmtemp=btime(0)
       tmwaih=tmwaih+tmtemp-tmold
c
c*        Reading half of the vector of upper bounds ...
c
       do 5 i=1,numnet*narcs/2
           ub(i)=decbfs(i,nextib)
 5     continue
c
       call frbufr(nextib)
       nextib=nextib+1
       if(nextib.gt.NUMRECBUFS) nextib=1
c
       tmold=btime(0)
       tmcomh=tmcomh+tmold-tmtemp
c
 610   if(.not.iflags(nextib)) goto 610
c
       tmtemp=btime(0)
       tmwaih=tmwaih+tmtemp-tmold
c
c*        ...  and the other half.
c
       do 51 i=1,numnet*narcs/2
           ub(i+numnet*narcs/2)=decbfs(i,nextib)
 51    continue
c
       call frbufr(nextib)
       nextib=nextib+1
       if(nextib.gt.NUMRECBUFS) nextib=1
c
       tmold=btime(0)
```

```
      tmcomh=tmcomh+tmold-tmtemp
c
c
      return
      end
```

```
      subroutine masen1
c
c                        MASTER ROUTINE
c*    This routine sends the final solution to the host.
c
c*       Inserting the declarations file.
c
#include "madec.h"
c
      real*8 btime
c
c*       Time observation.
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
c*       Waiting for the next output buffer.
c
 100  if(oflags(nextob)) goto 100
c
c*       Timing ...
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       Loading the first buffer with the final objective
c*       value, the lower bound, number of pivots and
c*       degenerate pivots, and number of calls to SEARCH.
c
      denbfs(1,nextob)=newobj
      denbfs(2,nextob)=lbound
      denbfs(3,nextob)=mpivot
      denbfs(4,nextob)=npivot
      denbfs(5,nextob)=nsea
c
c*       Specifying the destination (0 = host machine),
c*       sending the message and incrementing the
c*       output buffer counter.
c
      dest(nextob)=0
      call sendbf(nextob)
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
```

```
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
 200  if(oflags(nextob)) goto 200
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       Loading the buffer that contains half of
c*       the final solution vector.
c
      do 1 i=1,numnet*narcs/2
          denbfs(i,nextob)=xnew(i)
 1    continue
c
      dest(nextob)=0
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
c*       Checking if the host is ready to receive
c
 300  if(.not.iflags(nextib)) goto300
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
c*       Sending the buffer ...
c
      call sendbf(nextob)
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
 210  if(oflags(nextob)) goto 210
c
```

```
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
c*       ...  loading the other half.
c
      do 11 i=1,numnet*narcs/2
          denbfs(i,nextob)=xnew(i+numnet*narcs/2)
 11   continue
c
      dest(nextob)=0
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
c*       Checking if the host is ready.
c
 400  if(.not.iflags(nextib)) goto400
c
      tmtemp=btime(0)
      tmwaih=tmwaih+tmtemp-tmold
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      call sendbf(nextob)
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
      tmold=btime(0)
      tmcomh=tmcomh+tmold-tmtemp
c
c
      return
      end
```

```
      subroutine maslr1
c
c                        MASTER ROUTINE
c*    This routine receives the subproblem solution
c*    from the slave nodes.
c
c*       Inserting declarations file.
c
#include "madec.h"
c
      real*8 btime
c
c*       For each of the slave nodes 2, ..., numnet
c
      do 2 kk=2,numnet
c
c*       Timing ...
c
      tmold=btime(0)
      tmcomn=tmcomn+tmold-tmtemp
c
c*       Checking if a message has arrived in the next
c*       input buffer
c
 100  if(.not.iflags(nextib)) goto 100
c
c*       Time observation.
c
      tmtemp=btime(0)
      tmwain=tmwain+tmtemp-tmold
c
c*       Determining where the message came from.
c
      mach=source(nextib)
c
c*       Unloading the buffer containing number of pivots
c*       (MPIVOT is # of pivots this iteration, NPIVOT is
c*       total # of pivots), number of degenerate pivots,
c*       and the solution vector of the subproblem.
c
      mpivot=mpivot+decbfs(1,nextib)
      npivot=npivot+decbfs(1,nextib)
      idegp=idegp+decbfs(2,nextib)
```

```
      do 1 i=1,narcs
          xnew(i+(mach-1)*narcs)=decbfs(i+2,nextib)
 1    continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
c
 2    continue
c
c
      return
      end
```

```
      subroutine masls1
c
c                        MASTER ROUTINE
c*    This routine sends the current gridsize and base
c*    point to the slaves at each iteration.
c
c*       Including declaration file
c
#include "madec.h"
c
      real*8 btime
      integer mach
c
c*       For each slave ...
c
      do 1 mach=2,numnet
c
c*       Timing ...
c
      tmold=btime(0)
      tmcomn=tmcomn+tmold-tmtemp
c
c*       Waiting for next output buffer.
c
 400  if(oflags(nextob)) goto 400
c
c*       Timing ...
c
      tmtemp=btime(0)
      tmwain=tmwain+tmtemp-tmold
c
c*       Load the buffer containing the gridsizes.
c
      do 3 i=1,narcs
          denbfs(i,nextob)=delta(i+(mach-1)*narcs)
 3    continue
c
c*       Set destination, send buffer and increment buffer
c*       counter.
c
      dest(nextob)=mach
      call sendbf(nextob)
      nextob=nextob+1
```

```
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 1    continue
c
c*       For each slave node ...
c
      do 2 mach=2,numnet
c
      tmold=btime(0)
      tmcomn=tmcomn+tmold-tmtemp
c
 500  if(oflags(nextob)) goto 500
c
      tmtemp=btime(0)
      tmwain=tmwain+tmtemp-tmold
c
c*       Load half the base point XOLD.
c
      do 4 i=1,numnet*narcs/2
          denbfs(i,nextob)=xold(i)
 4    continue
c
c*       Sending ...
c
      dest(nextob)=mach
      call sendbf(nextob)
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 2    continue
c
c*       For each slave node ...
c
      do 7 mach=2,numnet
c
      tmold=btime(0)
      tmcomn=tmcomn+tmold-tmtemp
c
 510  if(oflags(nextob)) goto 510
c
      tmtemp=btime(0)
      tmwain=tmwain+tmtemp-tmold
c
```

```
c*        Load other half of XOLD.
c
      do 41 i=1,numnet*narcs/2
          denbfs(i,nextob)=xold(i+numnet*narcs/2)
 41   continue
c
c*        Sending ...
c
      dest(nextob)=mach
      call sendbf(nextob)
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
 7    continue
c
c
      return
      end
```

```
      subroutine slrec1
c*
c                        SLAVE ROUTINE
c*    This subroutine receives the new iteration data from
c*    the master node.
c
c*    Inserting the file of declarations.
c*
#include "sldec.h"
      real*8 btime
c
c*       Time observation.
c
      tmold=btime(0)
      tmcomm=tmcomm+tmold-tmtemp
c
c*       Waiting for next input buffer to become available.
c
 200  if(.not.iflags(nextib)) goto 200
c
c*       Timing ....
c
      tmtemp=btime(0)
      tmwait=tmwait+tmtemp-tmold
c
c*       Reading buffer containing gridsize information.
c
      do 2 i=1,narcs
          delta(i)=decbfs(i,nextib)
 2    continue
c
c*       Freeing the input buffer for further use and
c*       incrementing the buffer counter.
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      tmold=btime(0)
      tmcomm=tmcomm+tmold-tmtemp
c
 400  if(.not.iflags(nextib)) goto 400
c
```

```
      tmtemp=btime(0)
      tmwait=tmwait+tmtemp-tmold
c
c*       Unloading buffer containing half of
c*       new base point ...
c
      do 3 i=1,numnet*narcs/2
          xold(i)=decbfs(i,nextib)
 3    continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      tmold=btime(0)
      tmcomm=tmcomm+tmold-tmtemp
c
 410  if(.not.iflags(nextib)) goto 410
c
      tmtemp=btime(0)
      tmwait=tmwait+tmtemp-tmold
c
c*       .....  and the other half.
c
      do 31 i=1,numnet*narcs/2
          xold(i+numnet*narcs/2)=decbfs(i,nextib)
 31   continue
c
      call frbufr(nextib)
      nextib=nextib+1
      if(nextib.gt.NUMRECBUFS) nextib=1
c
      tmold=btime(0)
      tmcomm=tmcomm+tmold-tmtemp
c
c
      return
      end
```

```
      subroutine slsen1
c
c                        SLAVE ROUTINE
c*    This routine sends the solution of the corresponding
c*    subproblem from each slave node to the master node.
c
c*       Inserting the declarations file.
c
#include "sldec.h"
c
      real*8 btime
c
c*       Reading current time.
c
      tmold=btime(0)
      tmcomm=tmcomm+tmold-tmtemp
c
c*       Waiting for next output buffer to become available.
c
 100  if(oflags(nextob)) goto 100
c
c*       Time observation ...
c
      tmtemp=btime(0)
      tmwait=tmwait+tmtemp-tmold
c
c*       Loading the buffer with solution information.
c*       The first two entries are # of pivots this
c*       iteration and # of degenerate pivots and the
c*       next NARCS entries contain the solution of the
c*       subproblem.
c
      denbfs(1,nextob)=rtn(4)
      denbfs(2,nextob)=rtn(5)
      do 1 i=1,narcs
          denbfs(i+2,nextob)=xnew(i)
 1    continue
c
c*       Setting the destination for the message
c*       (1 = master node).
c
      dest(nextob)=1
c
```

```
c*       Sending the buffer.
c
      call sendbf(nextob)
c
c*       Incrementing the buffer counter.
c
      nextob=nextob+1
      if(nextob.gt.NUMSENDBUFS) nextob=1
c
      tmold=btime(0)
      tmcomm=tmcomm+tmold-tmtemp
c
c
      return
      end
```

REFERENCES

Barr, R. S. and Turner, J. S. [1981]: "Microdata file merging through large-scale network technology", *Mathematical Programming Study* 15, 1-22.

Beck, P., Lasdon, L. and Engquist, M. [1983]: "A reduced gradient algorithm for nonlinear network problems", *ACM Transactions on Mathematical Software*, Vol. 9, No. 1, 57-70.

Bertsekas, D. P. and Gafni, E. M. [1982]: "Projection methods for variational inequalities with application to the traffic assignment problem", *Mathematical Programming Study* 17, 139-159.

Cantor, D. G. and Gerla, M. [1974]: "Optimal routing in packet switched computer networks", *IEEE Transactions on Computing* C-23, 1062-1068.

Cook, R., Finkel, R., DeWitt, D., Landweber, L. and Virgilio, T. [1983]: "The Crystal Nugget", Technical Report 499, Computer Sciences Department, The University of Wisconsin-Madison.

Cryer, C. W. [1971]: "The solution of quadratic programming problems using systematic overrelaxation",*SIAM Journal of Control*, Vol. 9, 385-392.

Dantzig, G., Harvey, H., Lansdowne, Z., Robinson, D., and Maier, S. [1979]: "Formulating and solving the network design problem by decomposition", *Transportation Research* 13B, 5-17.

Dembo, R. S. [1981]: "Large scale nonlinear optimization", in *Nonlinear Optimization 1981*, M. J. D. Powell (Ed.), Academic Press.

Dembo, R. S. and Klincewicz, J. G. [1981]: "A scaled reduced gradient algorithm for network flow problems with convex separable costs", *Mathematical Programming Study* 15, 125-147.

DeWitt, D., Finkel, R., and Solomon, M. [1984]: "The CRYSTAL Multicomputer: Design and Implementation Experience", Technical Report 553, Computer Sciences Department, The University of Wisconsin-Madison.

Frank, M. and Wolfe, P. [1956]: "An algorithm for quadratic programming", *Naval Research Logistic Quarterly*, 3, 95-110.

Gavish, B. and Hantler, S. L. [1982]: "An algorithm for optimal route selection in SNA networks", Research Report RC 9549, IBM T. J. Watson Research Center, Yorktown Heights, N. Y.

Grigoriadis, M. D. and Hsu, T. [1979]: "RNET the Rutgers minimum cost network flow subroutine", *SIGMAP Bulletin*, 17-18.

Hanscom, M. A., Lafond, L., Lasdon, L. and Pronovost, G. [1980]: "Modeling and resolution of the medium term energy planning problem for a large hydro-electric system", *Management Science* 26, 659-668.

Kamesam, P. V. and Meyer, R. R. [1982]: "Multipoint methods for nonlinear networks", Technical Report 468, Computer Sciences Department, The University of Wisconsin-Madison. To appear in *Mathematical Programming Studies*.

Kao, C. Y. and Meyer, R. R. [1981]: "Secant approximation methods for convex optimization", *Mathematical Programming Study* 14, 143-162.

Lawphongpanich, S. and Hearn, D. W. [1983]: "Restricted Simplicial Decomposition with Application to the traffic assignment problem", Research Report 83-8, Department of Industrial and Systems Engineering, University of Florida, Gainesville, Fl.

Luenberger, D. G. [1984]: *Linear and Nonlinear Programming*, Second Edition, Addison-Wesley.

Magnanti, T. L. and Wong, R. T. [1984]: "Network design and transportation planning: models and algorithms", *Transportation Science* 18, 1-55.

Mangasarian, O. L. [1977]: "Solutions of linear complementarity problems by iterative methods", *Journal of Optimization Theory and Applications*, Vol. 22, 465-485.

Mangasarian, O. L. [1981]: "Sparsity preserving SOR algorithms for separable quadratic and linear programs", University of Wisconsin Mathematics Research Center, Technical Report 2260.

McCallum, C. J. [1976]: "A Generalized Upper Bounding approach to Communications network planning problems", *Networks* 7, 1-23.

Meyer, R. R. [1979]: "Two segment separable programming", *Management Science*, 25, 385-395.

Meyer, R. R. [1980]: "Computational aspects of two-segment separable programming", *Mathematical Programming* 26, 21-39.

Murtagh, B. A. and Saunders, M. A. [1983]: "MINOS 5.0 user's guide", Technical Report SOL 83-20, Stanford University, Stanford.

Pang, J. S. [1983]: private communication.

Pang, J. S. and Yu, C. S. [1982]: "Linearized Simplicial Decomposition Methods for computing traffic eqilibria on networks", Technical Report, University of Texas at Dallas, Richardson, Texas.

Rockafellar, R. T. [1984]: *Network Flows and Monotropic Programming*, Wiley.

Rosenthal, R. E. [1981]: "A nonlinear network flow algorithm for maximization of benefits in a hydroelectric power system", *Operations Research* 29, 763-786.

Steenbrink, P. A. [1974]: *Optimization of Transport networks* Wiley, London.

Thakur, L. S. [1978]: "Error analysis for convex separable programs", *SIAM Journal of Applied Mathematics*, 704-714.

Wolfe, P. [1967]: "Methods of nonlinear programming", in *Nonlinear Programming*, J. Abadie (Ed.).