To appear in Handbook on Operations Research and Managment Science: Networks, published by North Holland.

March 1994 LIDS-P-2236

Parallel Computing in Network Optimization[1]

Dimitri Bertsekas*
David Castañon†
Jonathan Eckstein‡
Stavros Zenios§

March 8, 1994

1 Introduction

Parallel and vector supercomputers are today considered basic research tools for several scientific and engineering disciplines. The novel architectural features of these computers — which differ significantly from the von Neumann model — are influencing the design and implementation of algorithms for numerical computation. Recent developments in large scale optimization take into account the architecture of the computer where the optimization algorithms are likely to be implemented. In the case of network optimization, in particular, we have seen significant progress in the design, analysis, and implementation of algorithms that are particularly well suited for parallel and vector architectures. As a result of these research activities, problems with several millions of variables can be solved routinely on parallel supercomputers. In this chapter, we discuss algorithms for parallel computing in

*Laboratory for Information and Decision Systems, Massachusetts Institute of Technology
†Department of Electrical, Computer and Systems Engineering, Boston University
‡Mathematical Sciences Research Group, Thinking Machines Corporation
§Decision Sciences Department, Wharton School, University of Pennsylvania

[1]Research supported by NSF under Grant CCR-9103804.

large scale network optimization.

We have chosen to focus on a sub-class of network optimization problems for which parallel algorithms have been designed. In particular, for the most part, we address only *pure* networks (*i.e.*, without arc multipliers). We also avoid discussion on large-scale problems with embedded network structures, like the *multicommodity* network flow problem, or the *stochastic* network problem. Nevertheless, we discuss parallel algorithms for both linear and nonlinear problems, and special attention is given to the assignment problem as well as other problems with bi-partite structures (*i.e.*, transportation problems). The problems we have chosen to discuss usually provide the building blocks for the development of parallel algorithms for the more complex problem structures that we are not addressing. Readers who are interested in a broader view of parallel optimization research — both for network structured problems and mathematical programming in general — should refer to several journal issues focused on parallel optimization which have been published recently on this topic [85, 86, 90, 113] or the textbook [22].

1.1 Organization of this Chapter

The introductory section discusses parallel architectures and broad issues that relate to the implementation and performance evaluation of parallel algorithms. It also defines the network optimization problems that will be discussed in subsequent sections. Section 2 develops the topic of parallel computing for linear network optimization problems and Section 3 deals with nonlinear networks. Concluding remarks, a brief overview of additional work for multicommodity network flows and stochastic network programs, as well as open issues, are addressed in Section 5.

Each of sections 2–4 is organized in three subsections along the following thread: First, we present general methodological ideas for the design of specific algorithms for each problem class. Here, we present selectively those algorithms that have some potential for parallelism. The methodological development is followed by a subsection of parallelization ideas, *i.e.*, specific ways in which each algorithm can be implemented on a parallel computer. Finally, computational results with the parallel implementation of some of the algorithms that have appeared in the literature are summarized and discussed.

1.2 Parallel Architectures

Parallelism in computer systems is not a recent concept. ENIAC — the first large-scale, general-purpose, electronic digital computer built at the University of Pennsylvania — was designed with multiple functional units for adding, multiplying, and so forth. The primary motivation behind this design was to deliver the computing power infeasible with the sequential electronic technology of that time. The shift from diode valves to transistors, integrated circuits, and very large scale integrated circuits (VLSI) rendered parallel designs obsolete and uniprocessor systems were predominant through the late sixties.

The first milestone in the evolution of parallel computers was the Illiac IV project at the University of Illinois in the 1970's. A brief historical note on this project can be found in [48]. The array architecture of the Illiac prompted studies on the design of suitable algorithms for scientific computing. Interestingly, a study of this sort was carried out for linear programming [104] — one of the first studies in parallel optimization. The Illiac never went past the stage of the research project, however, and only one machine was ever built.

The second milestone was the introduction of the CRAY 1 in 1976. The term *supercomputer* was coined at that time, and is meant to indicate the fastest available computer. The vector architecture of the CRAY introduced the notion of *vectorization* of scientific computing. Designing or restructuring of numerical algorithms to exploit the computer architecture — in this case vector registers and vector functional units — became once more a critical issue. Vectorization of an application can range from simple modifications of the implementation with the use of computational kernels that are streamlined for the machine architecture, to more substantive changes in data structure and the design of algorithms that are rich in vector operations.

Since the mid-seventies, supercomputers and parallel computers have been evolving rapidly in the level of performance they can deliver, the size of memory available, and the increasing number of parallel processors that can be applied to a single task. The Connection Machine CM–2, for example, can be configured with up to 65,536 very simple processing elements.

Several alternative parallel architectures have been developed. Today there is no single widely accepted model for parallel computation. A classification of computer architectures was proposed by Flynn [60] and is used

to distinguish between alternative parallel architectures. Flynn proposed the following four classes, based on the interaction among instruction and data streams of the processor(s):

1. SISD - *Single Instruction stream Single Data stream.* Systems in this class execute a single instruction on a single piece of data before moving on to the next piece of data and the next instruction. Traditional uniprocessor, scalar (von Neumann) computers fall under this category.

2. SIMD - *Single Instruction stream Multiple Data stream.* A single instruction can be executed simultaneously on multiple data. This of course implies that the operations of an algorithm are identical over a set of data and that data can be arranged for concurrent execution. An example of SIMD systems is the Connection Machine of Hillis [75].

3. MISD - *Multiple Instruction stream Single Data stream.* Multiple instructions can be executed concurrently on a single piece of data. This form of parallelism has not received, to our knowledge, extensive attention from researchers. It appears in Flynn's taxonomy for the sake of completeness.

4. MIMD - *Multiple Instruction stream Multiple Data stream.* Multiple instructions can be executed concurrently on multiple pieces of data. The majority of parallel computer systems fall in this category. Multiple instructions indicate the presence of independent code modules that may be executing independently from each other. Each module may be operating either on a subset of the data of the problem, have copies of all the problem data, or access all the data of the problem together with the other modules in a way that avoids read/write conflicts.

Whenever multiple data streams are used (*i.e.*, in the MIMD and SIMD systems) another level of classification is needed for the memory organization: In *shared memory* systems, the multiple data streams are accessible by all processors. Typically, a common memory bank is available. In *distributed memory* systems, each processor has access only to its own local memory. Data from the memories of other processors need to be communicated by passing messages across some communication network.

Multiprocessor systems are also characterized by the number of available processors. "Small-scale" parallel systems have up to 16 processors,

"medium-scale" systems up to 128, and "large-scale" systems up to 1024. Systems with 1024 or more processors are considered "massively" parallel. Finally, multiprocessors are also characterized as "coarse-grain" versus "fine-grain". In the former case each processor is very powerful, typically from the 80386 family, with several megabytes of memory. Fine grain systems typically use very simple processing elements with a few kilobytes of local memory each. For example, the NCUBE system with 1024 processors is considered a coarse-grain massively parallel machine. The Connection Machine CM–2 with up to 64K processing elements is a fine-grain, massively parallel machine. Of course these distinctions are qualitative in nature, and are likely to change as technology evolves.

A mode of computing that deserves special classification is that of *vector computers.* While vector computers are a special case of SIMD machines, they constitute a class of their own. This is due to the frequent appearance of vector capabilities in many parallel systems. Also the development of algorithms or software for a vector computer — like, for example, the CRAY — poses different problems than the design of algorithms for a system with multiple processors that operate synchronously on multiple data — like, for example, the Connection Machine CM–2. The processing elements of a vector computer are equipped with functional units that can operate efficiently on long vectors. This is usually achieved by segmenting functional units so that arrays of data can be processed in a pipeline fashion. Furthermore, multiple functional units may be available both for scalar and vector operations. These functional units may operate concurrently or in a *chained* manner, with the results of one unit being fed directly into another without need for memory access. Using these machines efficiently is a problem of structuring the underlying algorithm with (long) homogeneous vectors and arranging the operations to maximize chaining or overlap of the multiple units.

1.2.1 Performance Evaluation

There has been considerable debate on how to evaluate the performance of parallel implementations of algorithms. Since different algorithms may be suitable for different architectures, a valid way to evaluate the performance of a parallel algorithm is to implement it on a suitable parallel computer and compare its performance against the "best" serial code executing on a von Neumann system for a common set of test problems (of course, the parallel

and von Neumann computers should be of comparable prices). Furthermore, it is not usually clear what is the "best" serial code for a given problem, and the task of comparing different codes on different computer platforms is tedious and time-consuming. Hence, algorithm designers have developed several measures to evaluate the performance of a parallel algorithm that are easier to observe. The most commonly used are (1) *speedup*, (2) *efficiency*, (3) *scalability* and (4) *sustained FLOPS rates.*

Speedup: This is the ratio of solution time of the algorithm executing on a single processor, to the solution time of the same algorithm when executing on multiple processors. (This is also knownm as *relative speedup.*) It is understood that the sequential algorithm is executed on one of the processors of the parallel system (although this may not be possible for SIMD architectures). *Linear* speedup is observed when a parallel algorithm on p processors runs p times faster than on a single processor. $Sub-linear$ speedup is achieved when the improvement in performance is less than p. $Super-linear$ speedup (*i.e.*, improvements larger than p) usually indicates that the parallel algorithm takes a different — and more efficient — solution path than the sequential algorithm. It is often possible in such situations to improve the performance of the sequential algorithm based on insights gained from the parallel algorithms.

Amdahl [2] developed a law that gives an upper bound on the relative speedup that can be expected from a parallel implementation of an algorithm. If k is the fraction of the code that executes serially, while $1-k$ will execute on p processors, then the best speedup that can be observed is:

$$S_p = \frac{1}{k + (1-k)/p}$$

Relative speedup indicates how well a given algorithm is implemented on a parallel machine. It provides little information on the efficiency of the algorithm in solving the underlying problem. An alternative measure of speedup is the ratio of solution time of the best serial code on a single processor to the solution time of the parallel code when executing on multiple processors.

Efficiency: This is the ratio of speedup to the number of processors. It provides a way to measure the performance of an algorithm independently from the level of parallelism of the computer architecture. Linear speedup corresponds to 100% (or 1.00) efficiency. Factors less than 1.00 indicate sublinear speedup and superlinear speedup is indicated by factors greater than 1.00.

Scalability: This is the ability of an algorithm to solve a problem n times as large on np processors, as it would take to solve the original problem using p processors. Some authors [49] define *scaleup* as a measure of the scalability of a computer/code as follows:

$$\text{Scaleup } (p, n) = \frac{\text{Time to solve problem of size } m \text{ on } p \text{ processors}}{\text{Time to solve problem of size } nm \text{ on } np \text{ processors}}$$

FLOPS: This acronym stands for *Floating-point Operations per Second.* This measure indicates how well a specific implementation exploits the architecture of a computer. For example, an algorithm that executes at 190 MFLOPS (i.e., 10^6 FLOPS) on a CRAY X-MP that has a peak rate of 210 MFLOPS can be considered a successfully vectorized algorithm. Hence, little further improvements can be expected for this algorithm on this particular architecture. This measure does not necessarily indicate whether this is an efficient algorithm for solving problems. It is conceivable that an alternative algorithm can solve the same problem faster, even if it executes at a lower FLOPS rate. As of the writting of this chapter most commerically available parallel machines are able to deliver peak performance in the GFLOPS (i.e., 10^9 FLOPS) range. Dense linear algebra codes typically run at several GFLOPS, and similar performance has been achieved for dense transportation problems [88]. The current goal of high-performance computing is to design and manufacture machines that can deliver teraflops. Systems like the Intel Paragon and the Connection Machine CM–5 can, in principle, achieve such computing rates.

For further discussion on performance measures see the feature article by Barr and Hickman [6] and the commentaries that followed.

1.3 Synchronous versus Asynchronous Algorithms

In order to develop a parallel algorithm, one needs to specify a *partitioning sequence* and a *synchronization sequence.* The partitioning sequence determines those components of the algorithm that are independent from each other, and hence can be executed in parallel. These components are called *local algorithms.* On a multiprocessor system they are distributed to multiple processors for concurrent execution. The synchronization sequence specifies an order of execution that guarantees correct results. In particular, it specifies the data dependencies between the local algorithms. In a *synchronous* implementation, each local algorithm waits at predetermined points in time for a predetermined set of input data before it can proceed with the local calculations. Synchronous algorithms can often be inefficient, as processors may have to spend excessive amounts of time waiting for data from each other.

Several of the network optimization algorithms in this chapter have *asynchronous* versions. The main characteristic of an asynchronous algorithm is that the local algorithms do not have to wait at intermediate synchronization points. We thus allow some processors to compute faster and execute more iterations than others, some processors to communicate more frequently than others, and communication delays to be unpredictable. Also, messages might be delivered in a different order than the one in which they were generated. *Totally asynchronous* algorithms tolerate arbitrarily large communication and computation delays, whereas *partially asynchronous* algorithms are not guaranteed to work unless an upper bound is imposed on the delays. In asynchronous algorithms, substantial variations in performance can be observed between runs, due to the non-deterministic nature of the asynchronous computations. Asynchronous algorithms are most relevant to MIMD architectures, both shared memory and distributed memory.

Models for totally and partially asynchronous algorithms have been developed in Bertsekas and Tsitsiklis [22]. The same reference develops some general convergence results to assist in the analysis of asynchronous algorithms and establishes convergence of both partially and totally asynchronous algorithms for several network optimization problems.

Asynchronous algorithms have, potentially, two advantages over their synchronous counterparts. First, by reducing the synchronization penalty, they can achieve a speed advantage. Second, they offer greater implementation

flexibility and tolerance to changes of problem data. Experiences with asynchronous network flow algorithms is somewhat limited. A direct comparison between synchronous and asynchronous algorithms for nonlinear network optimization problems [35, 58] has shown that asynchronous implementations are substantially more efficient than synchronous ones. Further work on asynchronous algorithms for the assignment and min-cost flow problem [14, 15, 16] supports these conclusions. A drawback of asynchronous algorithms, however, is that termination detection can be difficult. Even if an asynchronous algorithm is guaranteed to be correct at the limit, it may be difficult to identify when some approximate termination conditions have been satisfied. Bertsekas and Tsitsiklis [22, chapter 8] address the problem of termination detection once termination has occurred. The question of ensuring that global termination of an asynchronous algorithm will occur through the use of approximate local termination conditions is surprisingly intricate, and has been addressed in [23, 114]. In spite of the difficulties in implementing and testing termination of asynchronous algorithms, the studies cited above have shown that these difficulties can be addressed successfully. Asynchronous algorithms for several network optimization problems have been shown to be more efficient than their synchronous counterparts when implemented on suitable parallel architectures.

1.4 Network Optimization Problems

We introduce here our notation and define the types of network optimization problems that will be used in later sections. The most general formulation we will be working with is the following *nonlinear network program* (NLNW):

$$\begin{aligned} \min \quad & F(x) && (1)\\ \text{s.t.} \quad & Ax = b && (2)\\ & l \leq x \leq u, && (3) \end{aligned}$$

where $F : \Re^m \longmapsto \Re$ is convex and twice continuously differentiable, A is an $n \times m$ node–arc incidence matrix, $b \in \Re^n$, l and $u \in \Re^m$ are given vectors and $x \in \Re^m$ is the vector of decision variables. The node–arc incidence matrix A specifies conservation of flow constraints (3) on some network $G = (\mathcal{N}, \mathcal{A})$ with $|\mathcal{N}| = n$ and $|\mathcal{A}| = m$. It can be used to represent pure networks

in which case each column has two non–zero entries : a "+1" and a "–1". Generalized networks are also represented by matrices with two non–zero entries in each column : a "+1" and a real number that represents the arc multiplier. An arc (i,j) is viewed as an ordered pair, and is to be distinguished from the pair (j,i). We define the vector x as the lexicographic order of the elements of the set $\{x_{ij} \mid (i,j) \in \mathcal{A}\}$. x is the flow vector in the network G, and x_{ij} is the flow of the arc (i,j). For a given x_{ij}, i is the row of the corresponding column of the constraint matrix A with with entry "+1", while j denotes the row with negative entry "–1" for pure networks, or the arc's multiplier $-m_{ij}$ for generalized networks. Similarily, components of the vectors $l, u,$ and x are indexed by (i,j) to indicate the from- and to-node of the corresponding network edge.

As a special case we assume that the function $F(x)$ is separable. Hence, model (NLNW) can be written in the equivalent form:

$$\min \quad \sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij}) \tag{4}$$

$$\text{s.t.} \quad Ax = b \tag{5}$$

$$l \leq x \leq u. \tag{6}$$

If the functions $f_{ij}(x_{ij})$ are linear we obtain the *min-cost network flow* problem. It can be expressed in algebraic form (MCF):

$$\min \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{7}$$

$$\text{s.t.} \quad \sum_{j:(i,j)\in\mathcal{A}} x_{ij} - \sum_{j:(j,i)\in\mathcal{A}} m_{ji}x_{ji} = b_i \quad \forall\, i \in \mathcal{N} \tag{8}$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \qquad \forall\, (i,j) \in \mathcal{A}. \tag{9}$$

This problem is a *generalized* network since the real-valued multipliers m_{ij} appear in the flow conservation equations. If all multipliers are equal to "1", then (MCF) is the *pure* min-cost network flow problem.

Some special forms of both the nonlinear and linear network problems are of interest. In particular, we will consider problems where G is bipartite, $G = (\mathcal{N}_O \bigcup N_D, \mathcal{A})$. $\mathcal{N}_O$ is the set of origin nodes, with supply vector s;

$\mathcal{N}_D$ is the set of destination nodes, with demand vector d. The nonlinear transportation problem can be written as (NLTR)

$$\min \sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij}) \tag{10}$$

$$\text{s.t.} \sum_{j:(i,j)\in\mathcal{A}} x_{ij} = s_i \quad \forall i \in \mathcal{N}_O \tag{11}$$

$$\sum_{i:(i,j)\in\mathcal{A}} x_{ij} = d_j \quad \forall j \in \mathcal{N}_D \tag{12}$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in \mathcal{A}. \tag{13}$$

Of course, if the functions $f_{ij}(x_{ij})$ are linear, we have a *linear transportation problem.* The special case of a transhipment problem when the supply and demand vectors are everywhere equal to one (*i.e.*, $s_i = 1$ for all $i \in N_O$ and $d_j = 1$ for all $j \in N_D$) is known as the *assignment problem.* (Assignment problems are usually stated as maximization than as minimization problems.)

We will also need some additional terminology when describing the algorithms: A *path* P in a directed graph is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ with $k \geq 2$ and a corresponding sequence of $k-1$ arcs such that the ith arc in the sequence is either (n_i, n_{i+1}) (in which case it is called a *forward* path of the arc) or (n_{i+1}, n_i) (in which case it is called a *backward* arc of the path). We denote by P^+ and P^- the set of forward and backward arcs of P, respectively. The arcs in P^+ and P^- are said to *belong* to P. Nodes n_1 and n_k are called the *start node* (or *origin*) and *end node* (or *destination*) of P, respectively. In the context of the min-cost flow problems, the *cost of a path* P is defined as $\sum_{(i,j)\in P^+} c_{ij} - \sum_{(i,j)\in P^-} c_{ij}$.

A *cycle* is a path whose start and end nodes are the same. A path is said to be *simple* if it contains no repeated arcs, and no repeated nodes, except possibly for the start and end nodes (in which case the path is called a *simple cycle*). A path is said to be *forward* (or *backward*) if all of its arcs are forward (respectively, backward) arcs. We refer similarily to a forward and a backward cycle. A directed graph that contains no cycles is said to be *acyclic.* A directed graph is said to be *connected* if for each pair of nodes i and j, there is a path starting at i and ending at j. A *tree* is a connected acyclic graph. A *spanning tree* of a directed graph Γ is a subgraph of Γ that

is a tree and includes all the nodes of Γ. A *simple path flow* is a flow vector x of the form

$$x = \begin{cases} a & \text{if } (i,j) \in P^+ \\ -a & \text{if } (i,j) \in P^- \\ 0 & \text{otherwise,} \end{cases}$$

where a is some positive number and P is a simple path. Finally, some additional notation: $\nabla^2 F(x^k)$ and $\nabla F(x^k)$ denotes the Hessian matrix and gradient vector of the function $F(x)$ evaluated at x^k. The transpose of a matrix A is denoted by A^T. $A_{\cdot t}$ and $A_{t\cdot}$ denote the t-th column and row respectively of A. e will be used to denote a conformable vector of all "1".

2 Linear Network Optimization

2.1 Basic Algorithmic Ideas

We will discuss three main ideas underlying min-cost flow algorithms. These are:

(a) Primal cost improvement; here we try to iteratively improve the cost to its optimal value by constructing a corresponding sequence of feasible flows.

(b) Dual cost improvement; here we define a problem related to the minimum cost flow problem, called the *dual*, whose variables are called *prices*. We then try to iteratively improve the dual objective function to its optimal value by constructing a corresponding sequence of prices. Dual cost improvement algorithms also iterate on flows, which are related to the prices through a property called *complementary slackness*.

(c) Approximate dual coordinate ascent; here one tries to iteratively improve the dual cost in an approximate sense along price coordinate directions. In addition to prices, algorithms of this type also iterate on flows, which are related to prices through a property called *ϵ-complementary slackness*, an approximate form of the complementary slackness property.

2.1.1 Primal Cost Improvement – The Simplex Method

A first important algorithmic idea for the min-cost flow problem is to start from an initial feasible flow vector, and generate a sequence of feasible flow vectors, each having a better cost value than the preceding one. For several interesting algorithms, including the simplex method, two successive flow vectors differ by a simple cycle flow.

Let us define a path P to be *unblocked with respect to* x if $x_{ij} < u_{ij}$ for all forward arcs $(i,j) \in P^+$ and $l_{ij} < x_{ij}$ for all backward arcs $(i,j) \in P^-$. The following is a basic result shown in several sources, e.g. [102, 112, 13]:

Proposition 1 *Consider the min-cost flow problem and let x be a feasible flow vector which is not optimal. Then there exists a simple cycle flow that when added to x, produces a feasible flow vector with smaller cost than x; the corresponding cycle is unblocked with respect to x and has negative cost.*

The major primal cost improvement algorithm for the min-cost flow problem, the simplex method, uses a particularly simple way to generate unblocked cycles with negative cost. It maintains a spanning tree T of the problem graph, and a partition of the set of arcs not in T in two disjoint subsets, L and U. Each triplet (T, L, U), called a basis, defines a unique flow vector x satisfying the conservation of flow constraints (8), and such that $x_{ij} = l_{ij}$ for all $(i,j) \in L$ and $x_{ij} = u_{ij}$ for all $(i,j) \in U$. The flow of the arcs (i,j) of T is specified as follows:

$$x_{ij} = \sum_{v \in T_i} b_v - \sum_{\substack{(v,w) \in L \\ v \in T_i \\ w \in T_j}} l_{vw} - \sum_{\substack{(v,w) \in U \\ v \in T_i \\ w \in T_j}} u_{vw} + \sum_{\substack{(v,w) \in L \\ v \in T_j \\ w \in T_i}} l_{vw} + \sum_{\substack{(v,w) \in U \\ v \in T_j \\ w \in T_i}} u_{vw},$$

where T_i and T_j are the subtrees into which (i,j) separates T.

A basis will be called feasible if the corresponding flow vector is feasible, that is, satisfies $l_{ij} \leq x_{ij} \leq u_{ij}$ for all $(i,j) \in T$. We say that the feasible basis (T, L, U) is *strongly feasible* if all arcs $(i,j) \in T$ with $x_{ij} = l_{ij}$ are oriented away from the root (that is, the unique simple path of T starting at the root and ending at j passes through i) and if all arcs $(i,j) \in T$ with $x_{ij} = u_{ij}$ are oriented towards the root (that is, the unique simple path from the root to i passes through j). Strongly feasible trees are used to deal with the problem of degeneracy (possible cycling through a sequence of bases).

The simplex method starts with a strongly feasible basis and proceeds in iterations, generating another feasible basis and a corresponding basic flow vector at each iteration. Each basic flow vector has cost which is no worse than the cost of its predecessor. At each iteration (also called a *pivot*, following standard linear programming terminology), the method operates roughly as follows:

(a) It uses a convenient method to add one arc to the tree so as to generate a simple cycle with negative cost.

(b) It pushes along the cycle as much flow as possible without violating feasibility.

(c) It discards one arc of the cycle, thereby obtaining another strongly feasible basis to be used at the next iteration.

To detect negative cost cycles, the simplex method fixes a root node r and associates with r a scalar p_r, which can be chosen arbitrarily. A basis (T, L, U) specifies a vector p using the formula:

$$p_i = p_r - \sum_{(v,w)\in P_i^+} c_{vw} + \sum_{(v,w)\in P_i^-} c_{vw}, \qquad \forall\, i \in \mathcal{N},$$

where P_i is the unique simple path of T starting at the root node r and ending at i, and P_i^+ and P_i^- are the sets of forward and backward arcs of P_i, respectively. We call p the *price vector* associated with the basis. The price vector defines uniquely the reduced cost of each arc (i, j) by

$$r_{ij} = c_{ij} + p_j - p_i.$$

Given the strongly feasible basis (T, L, U) with a corresponding flow vector x and price vector p, an iteration of the simplex method produces another strongly feasible basis $(\overline{T}, \overline{L}, \overline{U})$ as follows:

Typical Simplex Iteration

Find an in-arc $\overline{e} = (\overline{i}, \overline{j}) \notin T$ such that either

$$r_{\overline{i}\overline{j}} < 0 \qquad \text{if} \qquad \overline{e} \in L$$

or

$$r_{\overline{i}\overline{j}} > 0 \qquad \text{if} \qquad \overline{e} \in U.$$

(If no such arc can be found, x is primal optimal and p is dual optimal.) Let C be the cycle closed by T and $\overline{e}$. Define the forward direction of C to be the same as the one of $\overline{e}$ if $\overline{e} \in L$ and opposite to $\overline{e}$ if $\overline{e} \in U$ (that is, $\overline{e} \in C^+$ if $\overline{e} \in L$ and $\overline{e} \in C^-$ if $\overline{e} \in U$). Let also

$$\delta = \min\left\{ \min_{(i,j)\in C^-}\{x_{ij} - l_{ij}\}, \min_{(i,j)\in C^+}\{u_{ij} - x_{ij}\} \right\},$$

and let $\hat{C}$ be the set of arcs where this minimum is obtained

$$\hat{C} = \left\{(i,j) \in C^- \mid x_{ij} - l_{ij} = \delta\right\} \cup \left\{(i,j) \in C^+ \mid u_{ij} - x_{ij} = \delta\right\}.$$

Define the *join of* C as the first node of C that lies on the unique simple path of T that starts from the root and ends at $\overline{i}$. Select as out-arc the arc e of $\hat{C}$ that is encountered first as C is traversed in the forward direction starting from the join node. The new tree $\overline{T}$ is obtained from T by adding $\overline{e}$ and deleting e. The corresponding flow vector $\overline{x}$ is obtained from x by

$$\overline{x}_{ij} = \begin{cases} x_{ij} & \text{if } (i,j) \notin C \\ x_{ij} + \delta & \text{if } (i,j) \in C^+ \\ x_{ij} - \delta & \text{if } (i,j) \in C^-. \end{cases}$$

The initial strongly feasible tree is usually obtained by introducing an extra node 0 and the artificial arcs $(i,0)$, for all i with $b_i > 0$, and $(0,i)$, for all i with $b_i \leq 0$. The corresponding basic flow vector x is given by $x_{ij} = l_{ij}$ for all $(i,j) \in \mathcal{A}$, $x_{i0} = b_i$, for all i with $b_i > 0$, and $x_{0i} = -b_i$, for all i with $b_i \leq 0$. The cost of the artificial arcs is taken to be a scalar M, which is large enough to ensure that the artificial arcs do not affect the optimal solutions of the problem. This is known as the *Big M* initialization method. It can be shown that if M is large enough and the original problem (no artificial arcs) is feasible, the simplex method terminates with an optimal solution where all the artificial arcs carry zero flow. From this solution one can extract an optimal flow vector for the original problem.

2.1.2 Dual Cost Improvement

Duality theory for the min-cost flow problem can be developed by introducing a price vector $p = \{p_j \mid j \in \mathcal{N}\}$. We say that a flow-price vector pair (x,p)

satisfies *complementary slackness (or CS for short)* if x satisfies $l_{ij} \leq x_{ij} \leq u_{ij}$ and

$$x_{ij} < u_{ij} \quad \Rightarrow \quad p_i - p_j \leq c_{ij}, \qquad \forall\, (i,j) \in \mathcal{A}, \tag{14}$$

$$l_{ij} < x_{ij} \quad \Rightarrow \quad p_i - p_j \geq c_{ij}, \qquad \forall\, (i,j) \in \mathcal{A}. \tag{15}$$

Note that the above conditions imply that we must have $p_i - p_j = c_{ij}$ if $l_{ij} < x_{ij} < u_{ij}$.

The dual problem is obtained by a standard procedure in duality theory. We view p_i as a Lagrange multiplier associated with the conservation of flow constraint for node i and we form the corresponding Lagrangian function

$$\begin{aligned} L(x,p) &= \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} + \sum_{i\in\mathcal{N}} \left(b_i - \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} + \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} \right) p_i \\ &= \sum_{(i,j)\in\mathcal{A}} (c_{ij} + p_j - p_i)x_{ij} + \sum_{i\in\mathcal{N}} b_i p_i. \end{aligned} \tag{16}$$

Then, the dual function value $q(p)$ at a vector p is obtained by minimizing $L(x,p)$ over all capacity-feasible flows x:

$$q(p) = \min_x \{L(x,p) \mid l_{ij} \leq x_{ij} \leq u_{ij}, (i,j) \in \mathcal{A}\}. \tag{17}$$

Because the Lagrangian function $L(x,p)$ is separable in the arc flows x_{ij}, its minimization decomposes into A separate minimizations, one for each arc (i,j). Each of these minimizations can be carried out in closed form, yielding

$$q(p) = \sum_{(i,j)\in\mathcal{A}} q_{ij}(p_i - p_j) + \sum_{i\in\mathcal{N}} b_i p_i, \tag{18}$$

where

$$\begin{aligned} q_{ij}(p_i - p_j) &= \min_{x_{ij}} \{(c_{ij} + p_j - p_i)x_{ij} \mid l_{ij} \leq x_{ij} \leq u_{ij}\} \\ &= \begin{cases} (c_{ij} + p_j - p_i)l_{ij} & \text{if } p_i \leq c_{ij} + p_j \\ (c_{ij} + p_j - p_i)u_{ij} & \text{if } p_i > c_{ij} + p_j. \end{cases} \end{aligned} \tag{19}$$

The dual problem is to maximize $q(p)$ subject to no constraint on p. with the dual functional q given by (18).

While there are several other duality results, the following proposition is sufficient for our purposes:

Proposition 2 *If a feasible flow vector x^* and a price vector p^* satisfy the complementary slackness conditions (1) and (2), then x^* is an optimal primal solution and p^* is an optimal dual solution. Furthermore, the optimal primal cost and the optimal dual objective value are equal.*

Dual cost improvement algorithms start with a price vector and try to successively obtain new price vectors with improved dual cost value. An important method of this type is known as the *sequential shortest path method.* It is mathematically equivalent to the classical primal-dual method of Ford and Fulkerson [61]. Let us assume that the problem data are integer. The method starts with an integer pair (x, p) satisfying CS, and at each iteration, generates a new integer pair (x, p) satisfying CS such that the dual value of p is improved over the previous value.

To describe the typical iteration, for a pair (x, p) satisfying CS, define the *surplus* of a node i by

$$g_i = \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} - \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} + b_i.$$

An unblocked path is said to be an *augmenting path* if its start node has positive surplus and its end node has negative surplus. Consider the reduced costs of the arcs given by

$$r_{ij} = c_{ij} + p_j - p_i, \qquad \forall\ (i,j) \in \mathcal{A}. \tag{20}$$

We define the *length* of an unblocked path P by

$$L_P = \sum_{(i,j)\in P^+} r_{ij} - \sum_{(i,j)\in P^-} r_{ij}. \tag{21}$$

Note that since (x, p) satisfies CS, we have

$$r_{ij} \geq 0, \qquad \forall\ (i,j) \in P^+, \tag{22}$$

$$r_{ij} \leq 0, \qquad \forall\ (i,j) \in P^-. \tag{23}$$

Thus the length of P is nonnegative.

The sequential shortest path method starts each iteration with an integer pair (x, p) satisfying CS and with a set I of nodes i with $g_i > 0$, and proceeds as follows.

Sequential Shortest Path Iteration

Construct an augmenting path P with respect to x that has minimum length over all such paths that start at some node $i \in I$. Then, calculate

$$\delta = \min\Big\{\min\{u_{ij} - x_{ij} \mid (i,j) \in P^+\}, \{x_{ij} - l_{ij} \mid (i,j) \in P^-\}\Big\},$$

increase the flows of the arcs in P^+ by δ, and decrease the flows of the arcs in P^- by δ. Then, modify the node prices as follows: let $\overline{d}$ be the length of P and for each node $m \in \mathcal{N}$, let d_m be the minimum of the lengths of the unblocked paths with respect to x that start at some node in I and end at m ($d_m = \infty$ if no such path exists). The new price vector $\overline{p}$ is given by

$$\overline{p}_m = p_m + \max\{0, \overline{d} - d_m\}, \qquad \forall\, m \in \mathcal{N}. \tag{24}$$

The method terminates under the following circumstances:

(a) All nodes i have zero surplus; in this case it will be seen by Proposition 2 that the current pair (x,p) is primal and dual optimal.

(b) $g_i \leq 0$ for all i and $g_i < 0$ for at least one i; in this case the problem is infeasible, since $\sum_{i\in\mathcal{N}} b_i = \sum_{i\in\mathcal{N}} g_i < 0$.

(c) There is no augmenting path with respect to x that starts at some node in I; in this case it can be shown that the problem is infeasible.

We note that the shortest path computation can be executed using standard shortest path algorithms. The idea is to use r_{ij} as the length of each forward arc (i,j) of an unblocked path, and to reverse the direction of each backward arc (i,j) of an unblocked path and to use $-r_{ij}$ as its length [cf. the unblocked path length formula (21)]. Since by (22) and (23), the arc lengths of the residual graph are nonnegative, Dijkstra's method can be used for the shortest path computation. One can show the following result, which establishes the validity of the method (see *e.g.* [57, 13]).

Proposition 3 *Consider the min-cost flow problem and assume that a_{ij}, l_{ij}, u_{ij}, and b_i are all integer. Then, for the sequential shortest path method, the following hold:*

(a) *Each iteration maintains the integrality and the CS property of the pair* (x, p).

(b) *If the problem is feasible, then after a finite number of iterations, the method terminates with an integer optimal flow vector* x *and an integer optimal price vector* p.

(c) *If the problem is infeasible, then after a finite number of iterations, the method terminates either because* $g_i \leq 0$ *for all* i *and* $g_i < 0$ *for at least one* i*, or because there is no augmenting path from any node of the set* I *to some node with negative surplus.*

2.1.3 Approximate Dual Coordinate Ascent

Our third type of algorithm represents a significant departure from the cost improvement idea; at any one iteration, it may worsen both the primal and the dual cost, although in the end it does find an optimal primal solution. It is based on an approximate version of complementary slackness, called ϵ-*complementary slackness*, and while it implicitly tries to solve a dual problem, it actually attains a dual solution that is not quite optimal.

The main idea of this class of methods was first introduced for the symmetric assignment problem, where we want to match n persons and n objects on a one-to-one basis so that the total benefit from the matching is maximized. We denote here by a_{ij} the benefit of assigning person i to object j. The set of objects to which person i can be assigned is a nonempty set denoted $A(i)$. An *assignment* S is a (possibly empty) set of person-object pairs (i, j) such that $j \in A(i)$ for all $(i, j) \in S$; for each person i there can be at most one pair $(i, j) \in S$; and for every object j there can be at most one pair $(i, j) \in S$. Given an assignment S, we say that person i is *assigned* if there exists a pair $(i, j) \in S$; otherwise we say that i is *unassigned*. We use similar terminology for objects. An assignment is said to be *feasible* if it contains n pairs, so that every person and every object is assigned; otherwise the assignment is called *partial*. We want to find an assignment $\{(1, j_1), \ldots, (n, j_n)\}$ with total benefit $\sum_{i=1}^{n} a_{ij_j}$, which is maximal.

It is well-known that the assignment problem is equivalent to the linear program

$$\max \sum_{i=1}^{n} \sum_{j \in A(i)} a_{ij}(x_{ij}) \tag{25}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} x_{ij} = 1; \quad \forall i = 1, \ldots, n \tag{26}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j = 1, \ldots, n \tag{27}$$

$$0 \leq x_{ij} \leq 1 \qquad \forall i = 1, \ldots, n, j \in A(i). \tag{28}$$

This linear program in turn can be converted into a min-cost flow problem of the form (7)–(9) involving n nodes $i = 1, \ldots, n$ corresponding to the n persons, another n nodes $j = 1, \ldots, n$ corresponding to the n objects, the graph with the set of arcs

$$\mathcal{A} = \{(i,j) | i = 1, \ldots, n, j \in A(i)\}$$

with corresponding costs $c_{ij} = -a_{ij}$ for all $(i,j) \in \mathcal{A}$, and upper and lower bounds $l_{ij} = 0, u_{ij} = 1$ for all $(i,j) \in \mathcal{A}$. The supplies at the nodes $i = 1, \ldots, n$ and $j = 1, \ldots, n$ are set to $b_i = 1$ and $b_j = -1$ respectively.

The *auction algorithm* for the symmetric assignment problem proceeds iteratively and terminates when a feasible assignment is obtained. At the start of the generic iteration we have a partial assignment S and a price vector $p = (p_1, \ldots, p_n)$ satisfying *ϵ-complementary slackness* (ϵ-CS for short). This is the condition

$$a_{ij} - p_j \geq \max_{k \in A(i)} \{a_{ik} - p_k\} - \epsilon, \qquad \forall\, (i,j) \in S. \tag{29}$$

As an initial choice, one can use an arbitrary set of prices together with the empty assignment, which trivially satisfies ϵ-CS. The iteration consists of two phases: the *bidding phase* and the *assignment phase* described in the following.

Bidding Phase:

Let I be a nonempty subset of persons i that are unassigned under the assignment S. For each person $i \in I$:

1. Find a "best" object j_i having maximum value, that is,

$$j_i = \arg \max_{j \in A(i)} \{a_{ij} - p_j\},$$

and the corresponding value

$$v_i = \max_{j \in A(i)} \{a_{ij} - p_j\}, \tag{30}$$

and find the best value offered by objects other than j_i

$$w_i = \max_{j \in A(i), j \neq j_i} \{a_{ij} - p_j\}. \tag{31}$$

[If j_i is the only object in $A(i)$, we define w_i to be $-\infty$ or, for computational purposes, a number that is much smaller than v_i.]

2. Compute the "bid" of person i given by

$$l_{ij_i} = p_{j_i} + v_i - w_i + \epsilon = a_{ij_i} - w_i + \epsilon. \tag{32}$$

[We characterize this situation by saying that person i bid for object j_i, and that object j_i received a bid from person i. The algorithm works if the bid has any value between $p_{j_i} + \epsilon$ and $p_{j_i} + v_i - w_i + \epsilon$, but it tends to work fastest for the maximal choice of (32).]

Assignment Phase:

For each object j:

Let $P(j)$ be the set of persons from which j received a bid in the bidding phase of the iteration. If $P(j)$ is nonempty, increase p_j to the highest bid:

$$p_j := \max_{i \in P(j)} l_{ij},$$

remove from the assignment S any pair (i, j) (if j was assigned to some i under S), and add to S the pair (i_j, j), where i_j is a person in $P(j)$ attaining the maximum above.

Note that there is some freedom in choosing the subset of persons I that bid during an iteration. One possibility is to let I consist of a single unassigned person. This version, known as the *Gauss-Seidel version* in view of its similarity with Gauss-Seidel methods for solving systems of nonlinear equations, usually works best in a serial computing environment. The version

where I consists of all unassigned persons is the most well suited for parallel computation, and is known as the *Jacobi version*, in view of its similarity with Jacobi methods for solving systems of nonlinear equations.

The choice of bidding increment $v_i - w_i + \epsilon$ for a person i [cf.(32)] is such that ϵ-CS is preserved, as stated in the following proposition (see [8, 12, 22, 13]).

Proposition 4 *The auction algorithm preserves ϵ-CS throughout its execution, that is, if the assignment and price vector available at the start of an iteration satisfy ϵ-CS, the same is true for the assignment and price vector obtained at the end of the iteration.*

Furthermore, the algorithm is valid in the sense stated below.

Proposition 5 *If at least one feasible assignment exists, the auction algorithm terminates in a finite number of iterations with a feasible assignment that is within $n\epsilon$ of being optimal (and is optimal if the problem data is integer and $\epsilon < 1/n$).*

The auction algorithm can be shown to have an $O\left(A(n + nC/\epsilon)\right)$ worst-case running time, where A is the number of arcs of the assignment graph, and

$$C = \max_{(i,j)\in\mathcal{A}} |a_{ij}|$$

is the maximum absolute object value; see [8, 12, 22]. Thus, the amount of work to solve the problem can depend strongly on the value of ϵ as well as C. In practice, the dependence of the running time on ϵ and C is often significant, particularly for sparse problems.

To obtain polynomial complexity, one can use *ϵ-scaling*, which consists of applying the algorithm several times, starting with a large value of ϵ and successively reducing ϵ up to an ultimate value that is less than $1/n$. Each application of the algorithm, called a *scaling phase*, provides good initial prices for the next application. In practice, scaling is typically beneficial, particularly for sparse assignment problems, that is, problems where the set of feasible assignment pairs is severely restricted. ϵ-scaling was first proposed in [8] in connection with the auction algorithm. Its first analysis was given

in [70] in the context of the ϵ-relaxation method, which is a related method and will be discussed shortly.

The ϵ-CS condition (29) can be generalized for the min-cost flow problem. For a flow vector x satisfying $l_{ij} \leq x_{ij} \leq u_{ij}$ for all arcs (i,j), and a price vector p it takes the form

$$x_{ij} < u_{ij} \quad \Rightarrow \quad p_i - p_j \leq a_{ij} + \epsilon \qquad \forall\, (i,j) \in \mathcal{A}, \tag{33}$$

$$l_{ij} < x_{ij} \quad \Rightarrow \quad p_i - p_j \geq a_{ij} + \epsilon \qquad \forall\, (i,j) \in \mathcal{A}. \tag{34}$$

It can be shown that if x is feasible and satisfies the ϵ-CS conditions (33) and (34) together with some p, then the cost corresponding to x is within $N\epsilon$ of being optimal, where N is the number of nodes; x is optimal if it is integer, if $\epsilon < 1/n$, and if the problem data is integer.

We now define some terminology and computational operations that can be used as building blocks in various auction-like algorithms.

Definition 1 *An arc (i,j) is said to be* ϵ^+-unblocked *if*

$$p_i = p_j + a_{ij} + \epsilon \qquad \text{and} \qquad x_{ij} < u_{ij}.$$

An arc (j,i) is said to be ϵ^--unblocked *if*

$$p_i = p_j - a_{ji} + \epsilon \qquad \text{and} \qquad l_{ji} < x_{ji}.$$

The push list *of a node i is the (possibly empty) set of arcs (i,j) that are ϵ^+-unblocked, and arcs (j,i) that are ϵ^--unblocked.*

Definition 2 *For an arc (i,j) [or arc (j,i)] of the push list of node i, let δ be a scalar such that $0 < \delta \leq u_{ij} - x_{ij}$ ($0 < \delta \leq x_{ji} - l_{ji}$, respectively). A* δ-push at node i on arc (i,j) *[(j,i), respectively] consists of increasing the flow x_{ij} by δ (decreasing the flow x_{ji} by δ, respectively), while leaving all other flows, as well as the price vector unchanged.*

In the context of the auction algorithm for the assignment problem, a δ-push (with $\delta = 1$) corresponds to assigning an unassigned person to an object; this results in an increase of the flow on the corresponding arc from 0 to 1. The next operation consists of raising the prices of a subset of nodes by the maximum common increment γ that will not violate ϵ-CS.

Definition 3 *A* price rise *of a nonempty, strict subset of nodes I (i.e., $I \neq \emptyset$, $I \neq \mathcal{N}$), consists of leaving unchanged the flow vector x and the prices of nodes not belonging to I, and of increasing the prices of the nodes in I by the amount γ given by*

$$\gamma = \begin{cases} \min\{\gamma^+, \gamma^-\}, & \text{if } S^+ \cup S^- \neq \emptyset, \\ 0, & \text{if } S^+ \cup S^- = \emptyset, \end{cases}$$

where S^+ and S^- are the sets of scalars given by

$$S^+ = \{p_j + a_{ij} + \epsilon - p_i \mid (i,j) \in \mathcal{A} \text{ such that } i \in I, j \notin I, x_{ij} < u_{ij}\},$$

$$S^- = \{p_j - a_{ji} + \epsilon - p_i \mid (j,i) \in A \text{ such that } i \in I, j \notin I, l_{ji} < x_{ji}\},$$

and

$$\gamma^+ = \min_{s \in S^+} s, \qquad \gamma^- = \min_{s \in S^-} s.$$

The ϵ-relaxation method, first proposed in [10, 11], may be viewed as the extension of the auction algorithm to the min-cost flow problem. The ϵ-relaxation method uses a fixed positive value of ϵ, and starts with a pair (x,p) satisfying ϵ-CS. Furthermore, the starting arc flows are integer, and it will be seen that the integrality of the arc flows is preserved thanks to the integrality of the node supplies and the arc flow bounds. (Implementations that have good worst-case complexity also require that all initial arc flows be either at their upper or their lower bound; see *e.g.* [22]. In practice, this can be easily enforced, although it does not seem to be very important.) At the start of a typical iteration of the method we have a flow-price vector pair (x,p) satisfying ϵ-CS and we select a node i with $g_i > 0$; if no such node can be found, the algorithm terminates. During the iteration we perform several operations of the type described earlier involving node i.

Typical Iteration of the ϵ-Relaxation Method

Step 1: If the push list of node i is empty go to Step 3; else select an arc a from the push list of i and go to Step 2.

Step 2: Let j be the node of arc a which is opposite to i. Let

$$\delta = \begin{cases} \min\{g_i, u_{ij} - x_{ij}\} & \text{if } a = (i,j), \\ \min\{g_i, x_{ji} - l_{ji}\} & \text{if } a = (j,i). \end{cases}$$

Perform a δ-push of i on arc a. If as a result of this operation we obtain $g_i = 0$, go to Step 3; else go to Step 1.

Step 3: Perform a price rise of node i. If $g_i = 0$, go to the next iteration; else go to Step 1.

It can be shown that for a feasible problem, the method terminates finitely with a feasible flow vector, which is optimal if $\epsilon < 1/n$.

The ϵ-scaling technique discussed for the auction algorithm is also important in the context of the ϵ-relaxation method, and improves its practical performance. A complexity analysis of ϵ-scaling (first given in [70]; see also [17, 18, 22, 71]) shows that the ϵ-relaxation method, coupled with scaling, has excellent worst-case complexity.

It is possible to define a symmetric form of the ϵ-relaxation iteration that starts from a node with negative surplus and *decreases* (rather than increases) the price of that node. Furthermore, one can mix positive surplus and negative surplus iterations in the same algorithm. However, if the two types of iterations are mixed arbitrarily, the algorithm is not guaranteed to terminate finitely even for a feasible problem; for an example, see [22, p. 373]. For this reason, some care must be exercised in mixing the two types of iterations so as to guarantee that the algorithm eventually makes progress. With the use of negative surplus iterations, one can increase the parallelism potential of the method.

2.2 Parallelization Ideas

2.2.1 Primal Cost Improvement

The most computation-intensive part of the primal simplex method is the selection of a new arc (i, j) to bring into the basis. Ideally, one would like to select an arc (i, j) which violates the optimality condition the most, so that it has the largest $|r_{ij}|$. However, such an algorithm would require computation of all the reduced costs r_{ij} of all arcs at each iteration, and would be very time-consuming. An alternative is to select the first nonbasic arc $(i, j) \notin T$ that has negative reduced cost and its flow at its lower bound, or has positive reduced cost and its flow is at its upper bound. Such an implementation may quickly find a candidate arc, but the progress towards optimality may be slow.

Successful network simplex algorithms [94, 80, 65, 72] often adopt an intermediate strategy, where a candidate list of arcs is generated, and the

algorithm searches within this list for a candidate arc to bring into the basis. In this approach, selection of arcs becomes a two-phase procedure: First, a candidate list of arcs is constructed (typically, the maximum size of this list is a preset parameter). Second, the candidate list of arcs is scanned in order to find the arc in the candidate list which violates the optimality condition the most. While performing the second search at each iteration, arcs in the candidate list which no longer violate the optimality conditions are removed; eventually, the candidate list of arcs is empty. At this point, a new candidate list would be generated, and the iterations would be continued.

The main idea for accelerating the network simplex computations is the parallel computation of the candidate list of arcs. Using multiple processors, a larger list of candidate arcs can be generated and searched efficiently in a distributed manner to find a desirable pivot arc. Many approaches are possible, depending on how many arcs are considered between pivot steps and how the different candidate lists of arcs are generated. We will briefly overview three approaches: the work of [91] for transportation problems and the work of [103] and [5] for min-cost network flow problems.

In [91], a parallel simplex algorithm is presented for transportation problems. This algorithm is structured around the concept of selecting a candidate list of pivot arcs in parallel, and then performing the individual pivots in that candidate list in a sequential manner. Each processor evaluates a fixed number of arcs (for transportation problems, this corresponds to a fixed number of rows in the transportation matrix) in order to find the best pivot arc among its assigned arcs. The union of the sets of arcs evaluated by the parallel processors may only be a subset of the network. The set of pivot arcs found by all the processors becomes the candidate arc list. If no admissible pivot arcs were found by any of the processors, a different subset of arcs is searched in parallel for admissible pivot arcs. For transportation problems, this is coordinated by selecting which rows will be searched in parallel. Once a candidate list of pivot arcs is generated (with length less than or equal to the number of processors), the pivots in this candidate list are executed and reevaluated in sequential fashion.

The algorithm of [91] is essentially a synchronous algorithm, with processor synchronization after each processor has searched its assigned arcs. Using this synchronization guarantees that all of the processors have completed their search before pivoting begins. Synchronization also facilitates recognition of algorithm convergence; at the synchronization point, if no

candidate pivot arcs are found, the algorithm has converged to an optimal solution.

In [103], a different approach is pursued. Multiple processors are used to search in parallel the entire set of arcs in order to find the most desirable pivot arc in each iteration. At first glance, this appears inefficient. However, the parallel network simplex algorithm of [103] performs this search asynchronously, while the pivot operations are taking place. The processors are organized into a single pivot processor and a number of search processors. While the pivot processor is performing a pivoting operation, the search processors are using node prices (possibly outdated, since the pivoting operation may change them) to compute reduced costs and identify potential future pivot arcs. Each search processor is responsible for a subset of the arcs. Once a pivot operation is complete, the pivot processor obtains the combined results of the searches (which may not be complete), selects a new pivot arc, and performs a new pivot operation.

In contrast with [91], the algorithm of [103] has no inherent sequential computation time, since the search processors continue their search for pivot arcs while pivot operations are in progress. In order to guarantee algorithm convergence, the pivot processor must verify that the results of the search processors have indeed produced an acceptable pivot. In essence, one can view the operation of the search processors as continuously generating a small candidate list of pivot arcs; this list is then searched by the pivot processor to find the best pivot arc in the list. If this list is empty, the pivot processor will search the entire network to either locate a pivot arc or verify that an optimal solution has been reached. Thus, convergence of the asynchronous algorithm is established because a single processor (the pivot processor) selects valid pivot arcs, performs pivots and checks whether an optimal solution has been reached.

The algorithm of [5] is a refinement of the asynchronous approach of [103]. Instead of having processors dedicated to search tasks or pivot tasks, Barr and Hickman create a shared queue of tasks which are allocated to available processors using a monitor [76]. This approach offers the advantage that the algorithm can operate correctly with an arbitrary number of processors, since each processor is capable of performing each task. Barr and Hickman divide the search operation into multiple search tasks involving groups of outgoing arcs from disjoint subsets of nodes; they also divide the pivot operation into two steps: basis and flow update, and dual price update. Thus, the

shared queue of tasks includes three types of tasks: searching a group of arcs to identify potential future pivot arcs, selecting a candidate pivot arc and performing a basis and flow update, and updating the prices on a group of nodes. Among these tasks, the task of selecting a candidate pivot arc and performing a basis and flow update is assigned the highest priority.

In Barr and Hickman's algorithm, an idle processor will first confirm that there is no current pivot operation in progress, and check the list of candidate arcs to determine whether any eligible pivot arcs have been found. If this is the case, the processor will begin a pivot operation and, after completing the basis and flow update, the same processor will perform the dual price update. In the interim, other processors will continue the search operations, generating candidates while using potentially-outdated prices. However, if a pivot is currently in progress, then the idle processor will select a search task and search a subset of arcs to generate pivot candidates. An additional refinement introduced by Barr and Hickman is that, when the number of dual price updates is sufficiently high, the task of updating dual prices is split into two tasks, and scheduled for two different processors. Convergence is achieved when all search tasks have been completed after the dual prices have been updated, and no candidate pivot arcs are identified.

As the discussion indicates, all of the above parallel network simplex algorithms perform pivots one at a time. An alternative approach was proposed in [103], where multiple pivot operations would be performed simultaneously in parallel. As [103] points out, this is possible provided the different pivots affect non-overlapping parts of the network; otherwise, significant synchronization overhead is incurred. Computational results using this approach ([103]) indicate that, for general network problems, little speedup is possible from parallel computation. In contrast, the approaches discussed above obtain significant reductions in computation time using parallel computation, as described in the computation experience section.

The above concepts for parallel network algorithms have been extended to the case of linear generalized network problems by Clark and Meyer and their coworkers [36, 38, 40, 37, 39]. In their work, they develop variations of the parallel network simplex algorithms discussed above. In addition, due to the special structure of the basis for generalized network problems (discussed more extensively in the next chapter), it is often possible to perform parallel pivots which do not overlap. Clark and Meyer and their colleagues show that, for generalized networks, algorithms which perform multiple piv-

ots in parallel often outperform the generalizations of the network simplex algorithms discussed above.

2.2.2 Dual Cost Improvement

In contrast with the network simplex algorithms of the previous section, there are two successful classes of parallel sequential shortest path algorithms for min-cost network flow problems: we denote these as *single node* and *multinode* parallel algorithms. By a single node parallel sequential shortest path algorithm, we mean an algorithm where a single augmenting path is constructed efficiently using parallel computation. In contrast, a multinode algorithm constructs several augmenting paths in parallel (from multiple sources), and potentially executes multiple augmentations in parallel.

The key step in single node parallel shortest path algorithms is the computation of a shortest path tree from a single origin. Most sequential implementations use a variation of Dijkstra's algorithm [50], thereby exploiting the fact that only the distances to nodes which are closer than the length of the shortest path are needed. In a parallel context, alternative shortest path algorithms (e.g. Bellman-Ford [7] or the recently-proposed parallel pabel-correcting algorithm of [20] and auction algorithm of [107]) can be introduced which would be more amenable for parallel implementation. Nevertheless, the increased efficiency may be offset by the additonal computation of shortest paths to all nodes.

Parallelization of Dijkstra's algorithm can take place at two levels: parallel scanning of multiple nodes, and parallel scanning of the arcs connected to a single node. The effectiveness of the first approach is limited by the nature of Dijkstra's algorithm, which uses parallel scanning only when the shortest distance to two nodes is the same. Thus, most parallel implementations of Dijkstra's algorithm have been limited to parallel scanning of the arcs connected to a single node (*e.g.* [79, 123]) The effectiveness of this approach to parallelization is limited by the density of the network; the work of [79] and [123] focuses on fully dense assignment problems with large numbers of persons, so that significant speedups can be obtained. For sparse problems, the effective speedup is very limited.

The alternative multinode approach is to compute multiple augmenting paths (starting from different sources with positive surplus), and to combine the outcome of the computations to perform multiple augmentations in par-

allel and to obtain a new set of prices which satisfy CS with the resulting flows. Balas, Miller, Pekny, and Toth [3] introduced such an algorithm for the assignment problem. They developed a coordination protocol for combining the computations from multiple augmenting paths in order to increase the number of assignments and the node prices while preserving CS. In [15], these results were extended to allow for different types of coordination protocols, including asynchronous ones. Subsequently, the results of [3] and [15] were extended to obtain multinode parallel algorithms for the general network flow problem [16]. We discuss these results below.

The following parallel algorithm [16] is a direct generalization of the assignment algorithm of [3] to min-cost network flow problems. It starts with a pair (x,p) satisfying CS and generates another pair $(\bar{x},\bar{p})$ as follows:

Parallel Synchronous Sequential Shortest Path Iteration:

Choose a subset $I = \{i_1,\ldots,i_K\}$ of nodes with positive surplus. (If all nodes have nonpositive surplus, the algorithm terminates.) For each i_k, $k = 1,\ldots,K$, let $\bar{p}(k)$ and $\bar{P}(k)$ be the price vector and augmenting path obtained by executing a sequential shortest path iteration starting at i_k, and using the pair (x,p). Then generate sequentially the pairs $(x(k),p(k))$, $k = 1,\ldots,K$, as follows, starting with $(x(0),p(0)) = (x,p)$:

For $k = 0,\ldots,K-1$, if $\bar{P}(k+1)$ is an augmenting path with respect to $x(k)$, obtain $x(k+1)$ by augmenting $x(n)$ along $\bar{P}(k+1)$, and set

$$p_j(k+1) = \max\{p_j(k),\bar{p}_j(k)\}, \qquad \forall\, j \in \mathcal{N}.$$

Otherwise set

$$x(k+1) = x(k), \qquad p(k+1) = p(k).$$

The pair $(\bar{x},\bar{p})$ generated by the iteration is

$$\bar{x} = x(K), \qquad \bar{p} = p(K).$$

In [16], the above algorithm is shown to converge to an optimal solution of problem (MCF). Note that the protocol used for combining information from multiple parallel sequential shortest path computations consists of two conditions: First, multiple augmenting paths can be used provided that the

paths do not have any nodes in common. Second, whenever multiple augmenting paths are used, the prices of nodes are raised to the maximum level associated with any of the paths used for augmenting the flow in the network. The first condition guarantees that the arc flows stay within bounds, while the second condition guarantees that CS is preserved.

The preceding algorithm can be parallelized by using multiple processors to compute the augmenting paths and the associated prices of an iteration in parallel. On the other hand the algorithm is synchronous in that iterations have clear "boundaries". In particular, all augmenting paths generated in the same iteration are computed on the basis of the same pair (x, p). Thus, it is necessary to synchronize the parallel processors at the beginning of each iteration, with an attendant synchronization penalty.

In [16], an asynchronous version of the above algorithm is introduced. Let us denote the flow-price pair at the times $k = 1, 2, 3, \ldots$ by $(x(k), p(k))$. (In a practical setting, the times k represent "event times", that is, times at which an attempt is made to modify the pair (x, p) through an iteration.) We require that the initial pair $(x(1), p(1))$ satisfies CS. The algorithm terminates when during an iteration, either a feasible flow is obtained or else infeasibility is detected.

kth Asynchronous Primal-Dual Iteration:

At time k, the results of a primal-dual iteration performed on a pair $(x(\tau_k), p(\tau_k))$ are available, where τ_k is a positive integer with $\tau_k \leq k$; let $\bar{P}_k$ denote the augmenting path and $\bar{p}_k$ the resulting desired prices. The iteration (and the path $\bar{P}_k$) is said to be *incompatible* if $\bar{P}_k$ is not an augmenting path with respect to $x(k)$; in this case we discard the results of the iteration, that is, we set

$$x(k+1) = x(k), \qquad p(k+1) = p(k).$$

Otherwise, we say that the iteration (and the path $\bar{P}_k$) is *compatible*, we obtain $x(k+1)$ from $x(k)$ by augmenting $x(k)$ along $\bar{P}_k$, and we set

$$p_j(k+1) = \max\{p_j(k), \bar{p}_j(k)\}, \qquad \forall\, j \in \mathcal{N}.$$

Parallel implementation of this asynchronous algorithm is quite straightforward. The main idea is to maintain a "master" copy of the current flow-price pair; this is the pair $(x(k), p(k))$ in the preceding mathematical description. To execute an iteration, a processor copies the current master

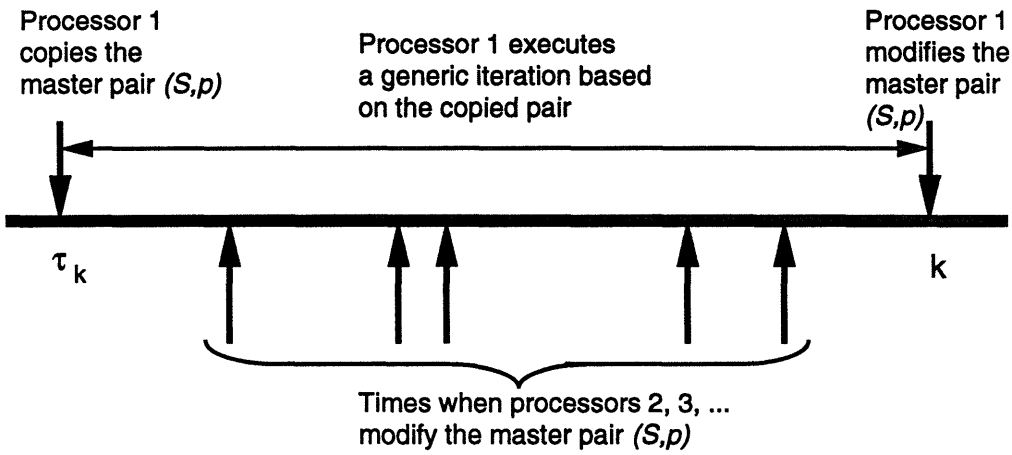


Figure 1: Operation of the asynchronous primal-dual algorithm in a shared memory machine. A processor copies the master flow-price pair at time τ_k, executes between times τ_k and k a generic iteration using the copy, and modifies accordingly the master flow-price pair at time k. Other processors may have modified unpredictably the master pair between times τ_k and k.

flow-price pair; during this copy operation the master pair is locked, so no other processor can modify it. The processor performs a primal-dual iteration using the copy obtained, and then attempts to modify the master pair (which may be different from the start of the iteration); the results of the iteration are checked for compatibility with the current flows $x(k)$, and if compatible, they are used to modify the master flow-price pair. The times when the master pair is copied and modified correspond to the indices τ_k and k of the asynchronous algorithm, respectively, as illustrated in Fig. 1.

In [16], the asynchronous algorithm is shown to converge under the condition

$$\lim_{k\to\infty} \tau_k = \infty.$$

This is a natural and essential condition, stating that the algorithm iterates with increasingly more recent information. A simpler sufficient condition guaranteeing convergence is that the maximum computation time plus communication delay in each iteration is bounded by an arbitrarily large constant.

2.2.3 Approximate Dual Coordinate Ascent

The auction and ϵ-relaxation algorithms of Section 2.2.3 allow for a variety of parallel implementations. Most of the experimental work (e.g. [105, 14, 123, 121, 122]) has focused on auction algorithms for assignment problems, for which the sequential performance of the auction algorithm is among the fastest. Thus, we restrict our parallelization discussion to the auction algorithm for assignment problems. Extensions of the concepts for parallel auction algorithms to parallel ϵ-relaxation algorithms for min-cost network flow problems have been developed recently by Li and Zenios [83] for implementation on the Connection Machine CM-2, and by Narendran et. al. in [96] for implementation on the CM-5.

The auction algorithm for the asignment problem was designed to allow an arbitrary nonempty subset I of unassigned persons to submit a bid at each iteration. This gives rise to a variety of possible implementations, named after their analogs in relaxation and coordinate descent methods for solving systems of equations or unconstrained optimization problems (see e.g. [101, 22]):

Jacobi where I is the set of all unassigned persons at the beginning of the iteration.

Gauss-Seidel where I consists of a single person, who is unassigned at the beginning of the iteration.

Block Gauss-Seidel where I is a subset of the set of all unassigned persons at the beginning of the iteration. The method for choosing the persons in the subset I may vary from one iteration to the next. This implementation contains the preceding two as special cases.

Similar to the dual improvement methods of the previous section, there are two basic approaches for developing parallel auction algorithms: in the first approach ("multinode"), the bids of several unassigned persons are carried out in parallel, with a single processor assigned to each bid; this approach is suitable for both the *Jacobi* and *block Gauss-Seidel* implementations. In the second approach ("single-node"), there is only one bid carried out at a time, but the calculation of each bid is done in parallel by several processors; this approach is suitable for the *Gauss-Seidel* implementation.

The above two approaches can be combined in a *hybrid* approach, whereby multiple bids are carried out in parallel, and the calculation of each bid is shared by several processors. This third approach, with proper choice of the number of processors used for each parallel task, has the maximum speedup potential.

An important characteristic of the auction algorithm is that it will converge to an optimal solution under a totally asynchronous implementation. The following result from [14] describes this asynchronous implementation:

We denote

$p_j(t)$ = Price of object j at time t

$r_j(t)$ = Person assigned to object j at time t [$r_j(t) = 0$ if object j is unassigned]

$U(t)$ = Set of unassigned persons at time t [$i \in U(t)$ if $r_j(t) \neq i$ for all objects j].

We assume that $U(t)$, $p_j(t)$, and $r_j(t)$ can change only at *integer* times t (t may be viewed as the index of a sequence of physical times at which events of interest occur.)

In addition to $U(t)$, $p_j(t)$, and $r_j(t)$, the algorithm maintains at each time t, a subset $R(t) \subset U(t)$ of unassigned persons that may be viewed as having a "ready bid" at time t. We assume that by time t, a person $i \in R(t)$ has used prices $p_j(\tau_{ij}(t))$ and $p_j(\overline{\tau}_{ij}(t))$ from some earlier times $\tau_{ij}(t)$ and $\overline{\tau}_{ij}(t)$ with $\tau_{ij}(t) \leq \overline{\tau}_{ij}(t) \leq t$ to compute the best value

$$v_i(t) = \max_{j|(i,j)\in A} \{a_{ij} - p_j(\tau_{ij}(t))\}, \tag{35}$$

a best object $j_i(t)$ attaining the above maximum,

$$j_i(t) = \arg \max_{j|(i,j)\in A} \{a_{ij} - p_j(\tau_{ij}(t))\}, \tag{36}$$

the second best value

$$w_i(t) = \max_{j|(i,j)\in A, j\neq j_i(t)} \{a_{ij} - p_j(\overline{\tau}_{ij}(t))\}, \tag{37}$$

and has determined a bid

$$\beta_i(t) = a_{ij_i(t)} - w_i(t) + \epsilon. \tag{38}$$

(Note that ordinarily the best and second best values should be computed simultaneously, which implies that $\tau_{ij}(t) = \overline{\tau}_{ij}(t)$. In some cases, however, it may be more natural or advantageous to compute the second best value after the best value, with more up-to-date price information, which corresponds to the case $\tau_{ij}(t) \leq \overline{\tau}_{ij}(t)$ for some pairs (i, j).)

Assumption 1:

$$U(t) : \text{nonempty} \quad \Rightarrow \quad R(t') : \text{nonempty for some } t' \geq t.$$

Assumption 2: For all i, j, and t,

$$\lim_{t \to \infty} \tau_{ij}(t) = \infty.$$

The above assumptions guarantee that unassigned persons do not stop submitting bids and old information is eventually discarded. At each time t, if all persons are assigned ($U(t)$ is empty), the algorithm terminates. Otherwise, if $R(t)$ is empty nothing happens. If $R(t)$ is nonempty the following occur:

(a) A nonempty subset $I(t) \subset R(t)$ of persons that have a bid ready is selected.

(b) Each object j for which the corresponding bidder set

$$B_j(t) = \{i \in I(t) \mid j = j_i(t)\}$$

is nonempty, determines the highest bid

$$b_j(t) = \max_{i \in B_j(t)} \beta_i(t)$$

and a person $i_j(t)$ for which the above maximum is attained

$$i_j(t) = \arg \max_{i \in B_j(t)} \beta_i(t).$$

Then, the pair $(p_j(t), r_j(t))$ is changed according to

$$(p_j(t+1), r_j(t+1)) = \begin{cases} (b_j(t), i_j(t)) & \text{if } b_j(t) \geq p_j(t) + \epsilon \\ (p_j(t), r_j(t)) & \text{otherwise.} \end{cases}$$

The following proposition [14] establishes the validity of the above asynchronous algorithm.

Proposition 6 *Let Assumptions 1 and 2 hold and assume that there exists at least one complete assignment. Then for all t and all j for which $r_j(t) \neq 0$, the pair $(p_j(t), r_j(t))$ satisfies the ϵ-CS condition*

$$\max_{k|(i,k)\in A}\{a_{ik} - p_k(t)\} - \epsilon \leq a_{ij} - p_j(t), \qquad \text{if } i = r_j(t).$$

Furthermore, there is a finite time at which the algorithm terminates. The complete assignment obtained upon termination is within $n\epsilon$ of being optimal, and is optimal if $\epsilon < 1/n$ and the benefits a_{ij} are integer.

Notice that if $\tau_{ij}(t) = t$ and $U(t) = R(t)$ for all t, then the asynchronous algorithm is equivalent to the auction algorithm given in Section 2.1.3. The asynchronous model becomes relevant in a parallel computation context where some processors compute bids for some unassigned persons, while other processors simultaneously update some of the object prices and corresponding assigned persons. Suppose that a single processor calculates a bid of person i by using the values $a_{ij} - p_j(\tau_{ij}(t))$ prevailing at times $\tau_{ij}(t)$ and then calculates the maximum value at time t; see Figure 2. Then, if the price of an object $j \in A(i)$ is updated between times $\tau_{ij}(t)$ and t by some other processor, the maximum value will be based on out-of-date information. The asynchronous algorithm models this possibility by allowing $\tau_{ij}(t) < t$. A similar situation arises when the bid of person i is calculated cooperatively by several processors rather than by a single processor.

It is interesting to contrast the asynchronous auction algorithm described above with the asynchronous sequential shortest path algorithm of [15]. In the sequential shortest path algorithm, computation of each shortest path must be based on a complete set of flows and prices (x, p) which may be outdated, but which represent a single previous state of the computation. In contrast, the asynchronous auction algorithm can use prices and assignments from many different times; thus, this algorithm requires much less coordination, and allows different processors to modify assignments and prices simultaneously.

Similar to the auction algorithms, the ϵ-relaxation algorithm of Section 2.1.3 also admits Gauss- Seidel, Jacobi and hybrid parallel implementations,

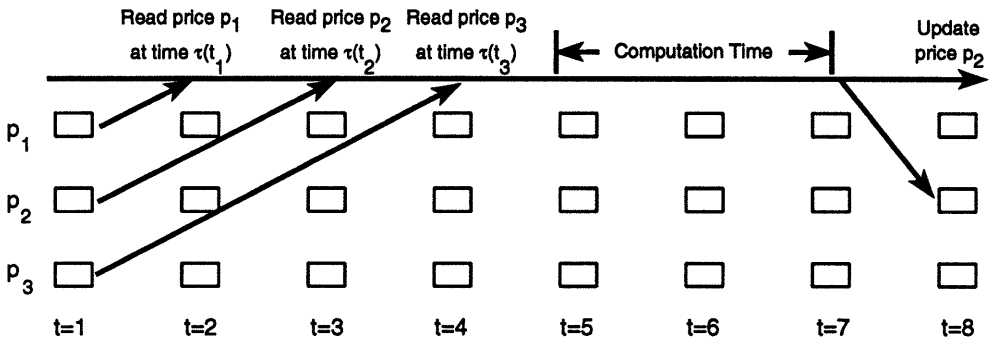


Figure 2: Illustration of asynchronous calculation of a bid by a single processor, which reads from memory the values p_j at different times $\tau_{ij}(t)$ and calculates at time t the best object $j_i(t)$ and the maximum and second maximum values. The values of p_j may be out-of-date because they may have been updated by another processor between the read time $\tau_{ij}(t)$ and the bid calculation time t.

as well as asynchronous implementations. For a convergence analysis of several synchronous and asynchronous implementations of ϵ-relaxation, the reader should consult [22].

2.3 Computation Experience

2.3.1 Primal Cost Improvement

In this section, we overview the results of parallel computation experiments reported in [91], [103], and [5] using parallel simplex algorithms for transportation and min-cost network flow problems.

As discussed previously, Miller, Pekny, and Thompson [91] implemented a synchronous version of the transportation simplex algorithm for dense transportation problems on the BBN Butterfly Plus [108] computer. The Butterfly Plus is a distributed- memory computer; each processor (Motorola 68020/68881) has 4 Megabytes of local memory. Processors may access information in other processor's memory, but such accesses are slower than accesses to local memory. The implementation of [91] divides of the rows of the transportation matrix among the processors; each processor only stores

the elements for which it is responsible. In addition, each processor keeps a complete local copy of the basis tree and dual variables.

The algorithm of [91] works as follows: At the beginning of each iteration, each processor searches a subset of its rows to locate suitable pivot arcs; the best arc found by each processor is stored in a *pivot queue.* A synchronization point is introduced to make sure that all processors have completed the search before going to the next part of the iteration. After synchronization, the processors check the pivot queue; if it is empty, a new search operation takes place using different rows. If the pivot queue is not empty, every processor uses the identical logic to select pivot elements and update its local copy of the basis tree and dual variables. Due to the distributed memory nature of the Butterfly Plus, it is more efficient to replicate these computations than to communicate the results among processors. At the end of an iteration, the processors start the search process for the next iteration.

The algorithm of [91] was tested on dense assignment and transportation problems with equal numbers of sources and sinks, using between 500 and 3000 sources and costs generated uniformly over a variable range. In order to evaluate the speedup, two sets of experiments were conducted: First, a single processor was used (although the data was stored over all 14 processors due to memory limitations); second, all 14 processors were used. Unfortunately, distributing the problem data across all 14 processors slows down the execution of the single-processor algorithm significantly, as most of the reduced cost computations require referencing to memory which is not colocated with the processor. In contrast, the parallel algorithm does not require use of any non-local memory references; this results in increased effectiveness of the parallel algorithm. In the computation experiments with arc costs in [0,1000] and [0,10000], the parallel algorithm ran between 3 and 7.5 times faster than the single-processor algorithm [91]; notably, this speedup increased with problem size. This is because the number of arcs to be searched increases quadratically with problem size, so that the search time becomes a larger fraction of the overall computation time.

Similar results were observed by Peters [103] in test experiments using the parallel network simplex code PARNET. The PARNET code was implemented on a shared-memory multiprocessor (Sequent Symmetry S-81 [116]). The PARNET implementation requires a minimum of 3 processors (1 pivot processor and 2 search processors), so that a direct comparison with a single-processor version was not possible. Instead, Peters [103] compares the PAR-

NETGEN Problem	NETFLO (1)	PARNET (3)	PARNET (7)
110	430.50	28.19	12.42
122	802.60	72.28	21.91
134	195.40	23.94	6.26
150	802.60	71.62	15.13

Table 1: Computation time in seconds on Sequent Symmetry for PARNET and NETFLO.

NET performance with the performance of NETFLO [80] on a variety of NETGEN [81] problems. The results indicate that PARNET is often 10-20 times faster using only three processors. However, this speedup is more indicative of the different search strategies used by NETFLO and PARNET than the advantage of using parallel processing. In essence, the use of multiple processors to search the entire set of arcs for candidate pivots reduces the total number of pivot iterations required for convergence. An even greater difference is observed when PARNET is implemented using 7 processors; although the number of search processors is tripled, the computation times compared to PARNET with three processors is often reduced by factors greater than 3! This superlinear speedup is again due to the use of different pivot strategies; by using more processors, more arcs can be searched between pivots. Table 1 summarizes some of the results reported in [103].

It should be noted that the above times represent a single run of the algorithm for each problem. Given the asynchronous nature of the PARNET algorithm, some variability in computation time should be expected, as small variations in the relative timing of search and pivot iterations may result in very different sequences of iterations.

Peters also examined the question of whether parallelization effectiveness increases with problem size. Two sets of experiments were conducted. In the first set, a set of network problems with constant number of arcs per node were generated (roughly 50 arcs per node, with problems ranging from 20,000 to 50,000 nodes). Interestingly, the results in [103] indicate that, once the number of processors is around 7, little additional speedup is obtained from using more processors. In essence, using 6 search processors provides enough processing to search the entire set of arcs while the pivot processor performs a

NETGEN Problem	PPNET (1)	PARNET (3)	PPNET (3)
110	39.98	28.19	13.99
122	79.91	72.28	32.84
134	64.94	23.94	15.42
150	84.93	71.62	34.49

Table 2: Computation time in seconds on Sequent Symmetry for PARNET and PPNET

pivot operation. In the second set of experiments, Peters generated problems with a constant number of nodes (1000), but an increasing number of arcs (from 25,000 to 500,000). In these experiments, Peters found that the number of processors which could be used efficiently increased from 6 in the sparsest problems to 12 in the denser problems. For additional details and results, the interested reader is referred to [103].

Barr and Hickman [5] developed their parallel network simplex code, PPNET, as an extension of the sequential NETSTAR code written by Barr based on the ARC-II code of [4]. Their experimental results on a comparable Symmetry S81 showed that the PPNET algorithm is roughly twoice as fast as the PARNET algorithm of [103]. This is due in part to the superiority of the task scheduling approach, and to the division of of the lengthier price update tasks among two different processors. Table 2 summarizes some of the results in [5], and compares the performance of the PPNET and PARNET algorithms using 3 processors for the same NETGEN problems.

It should be noted that the PARNET times in Table 2 include input, initialization, computation, and output times. In contrast, the PPNET times exclude input and output times. Furthermore, all times reported are the results of single runs of the algorithms; the results of [5] and [103] do not provide any indications concerning the run-to-run variability of the computation times.

Barr and Hickman [5] conducted an exhaustive test of PPNET for NETGEN problems with 1,000 to10,000 nodes and 12,500 to 75,000 arcs, using from 1 to 10 processors. Their results show that, for most problems in this range, maximum speedups are obtained using 5 or 6 processors; adding additional processors can reduce the performance of the algorithm due to

additional synchronization overhead associated with the monitor operations. With 6 processors, an average speedup of 4.68 was obtained. As in [103], the critical limitation in speedup is the time required to perform a pivot operation; once enough search processors are available to perform a complete search of the arcs during a pivot operation, adding additional processors merely creates extra overhead. In PPNET, this limit is reached sooner because the average duration of pivot operations is reduced as compared to PARNET due to the use of parallel dual price updates. Indeed, Barr and Hickman's experiments confirm that when 3 or more processors are used, a pivot is in progress over 85% of the time. For additional results and details, the reader is referred to [5].

2.3.2 Dual Cost Improvement

Most of the work on parallel dual cost improvement algorithms has focused on implementations of the algorithm (JV) of Jonker and Volgenant [77] for dense assignment problems. The JV algorithm is a hybrid algorithm consisting of two phases: an initialization phase (which uses an auction-like procedure for finding a partial assignment satisfying CS) and a sequential shortest path phase, which uses Dijkstra's algorithm. The first parallel implementation of the JV algorithm was performed by Kennington and Wang [79] on an 8-processor Sequent Symmetry S81. In their implementation, both the initialization and sequential shortest path phases are executed in parallel. For the sequential shortest path phase, multiple processors are used to update node distances and to find which node to scan next in parallel. Experimental results in [79] using dense assignment problems with 800, 1000 and 1200 persons indicate speedups in the range of 3-4 as the number of processors is increased from 1 to 8. However, this is the combined speedup of the initialization phase; no information was provided concerning how much speedup was accomplished during the shortest path phase of the algorithm. Futhermore, the speedups increase with problem size, since the number of arcs which must be examined in parallel increases with problem size (for dense problems).

In order to shed some insight on the effectiveness of single-node parallelization for sequential shortest path methods, we developed a parallel implementation of the JV algorithm on the Encore Multimax, a 20-processor shared-memory parallel computer. In these experiments, we used 1000 per-

1000 x 1000 assignment, cost range [0- 1000]					
No. Processors	1	2	4	6	8
Shortest Path time (100% dense)	112.7	77.5	60.0	54.0	52.0
Total time (100% dense)	132.5	87.9	65.7	59.0	56.2
Shortest Path time (10% dense)	14.6	10.4	8.6	8.6	X
Total time (10% dense)	17.4	12.1	9.7	9.7	X

Table 3: Computation time of the parallel JV algorithm on the Encore Multimax.

son assignment problems of different densities, and varied the number of processors. Table 3 summarizes the results of the experiments using fully-dense and 10% dense assignment problems. Note that the speedup of the sequential shortest path phase is significantly smaller for the successive shortest path phase than for the overall algorithm. Furthermore, the overall speedups are in the order of 2.5 for dense problems, and decrease to under 2 for 10% dense problems.

The experimental results discussed above have used shared-memory parallel architectures for implementing single-node parallelization. These architectures permit the use of sophisticated data structures to take advantage of problem sparsity in the algorithms; however, the above results illustrate that the effectiveness of single-node parallel algorithms is limited by the density of the network. As an alternative, these parallel algorithms can be implemented using massively parallel single-instruction, multiple-data stream (SIMD) processors such as the Connection Machine. Castañon *et al.* [27] have developed parallel implementations of the JV algorithm on the DAP 510 (1024 single-bit processors) and the Connection Machine CM-2 (using only 1024 of its 65,536 processors). Both of these machines are array processors attached to a sequential processor. In order to minimize communications, the cost matrix was stored as a dense array, spread across the processors so that each

processor has one row of the cost matrix. For the same problems discussed in Table 3, the computation times on the DAP 510 were 1 and 1.6 seconds for the dense and sparse problem, respectively. On the CM-2, the computation times were 18.7 and 29.1 seconds respectively.

Note the effectiveness of the massively parallel DAP architecture for solving both dense and sparse assignment problems (compared with the Encore Multimax results of Table 2.) The difference between the DAP and CM-2 times is due to the CM architecture, which is designed to work with 65,536 processors; our implementation required the use of only 1000 processors. In contrast, the DAP 510 architecture is optimized for 1024 processors. The above results highlight the advantage of SIMD architectures for implementing single-node parallel algorithms.

The theory of multinode parallel dual cost improvement algorithms has been worked out recently [3, 15, 16]; thus, computational experience with these methods is limited. For fully- dense assignment problems, Balas *et al.* [3] developed a synchronous parallel sequential shortest path algorithms and implemented it on the 14-processor Butterfly Plus computer discussed previously. Subsequently, Bertsekas and Castañon [15] conducted further experiments on the Encore Multimax with other synchronous algorithms as well as an asynchronous variation of the sequential shortest path algorithm (corresponding closely to the theoretical algorithm of Section 2.1.3), in order to identify relative advantages and disadvantages of the synchronous and asynchronous algorithms. We discuss these experimental results below.

In [15], three different parallel variations of the successive shortest path algorithm were evaluated:

(1) *Single path synchronous (SS)*: At each iteration, every processor finds a single shortest augmenting path from an unassigned person to an unassigned object.

(2) *Self-scheduled synchronous (SSS)*: At each iteration, every processor finds a variable number of shortest augmenting paths sequentially until the total number of augmenting paths equals some threshold number, which depends on the number of unassigned persons and the number of processors. This variation closely resembles the implementation of [3].

(3) *Single path asynchronous (AS)*: At each iteration, each processor finds a single shortest augmenting path from an unassigned person to an unassigned object, but the processors execute the iterations asyn-

chronously.

The purpose of the self-scheduled synchronous algorithm is to reduce the coordination overhead among processors, while improving computation load balance among the processors. Processors which find augmenting paths quickly will be assigned more work. Furthermore, synchronization is used less often than in single-path synchronous augmentation, because a fixed number of augmenting paths must be computed.

The experimental results of [15] illustrate many of the limitations in reducing computation time using multinode parallelization in dual improvement algorithms. Figure 3 summarizes the computation time in the sequential shortest paths phase (averaged across three runs) of the three algorithms for a 1000 person, 30% dense assignment problem, cost range [1,1000]. When a single processor is used, the self-scheduled synchronous algorithm is the slowest because it must find additional shortest paths (as a result of incompatibility problems). As the number of processors increases, the reduced coordination required by the self-scheduled synchronous algorithm makes it faster than the single-path synchronous algorithm. For these experiments, the single path asynchronous algorithm is fastest.

In [15], a measure of coordination overhead is proposed. This measure, called the wait time, is the time that a processor which has work to do spends waiting for other processors. For the synchronous algorithms, the majority of the wait time occurs at the synchronization point in each iteration, when processors which have already completed their augmenting path computations wait for other processors. For the asynchronous algorithm, it is the time waiting to get access to the master copy (while other processors are modifying it). Figure 4 shows the average wait time per processor for the results in Figure 3.

The above results indicate the existence of some fundamental limits in the speedups which can be achieved from parallel processing of dual improvement algorithms. As discussed in [15], the principal limitation is the decreasing amount of parallel work in each iteration. Initially, there are many augmentations which must be found, so that many processors can be used effectively. However, as the algorithm progresses, there are fewer unassigned persons, so the number of parallel augmentations decreases significantly. Furthermore, the later augmentations are usually harder to compute, as they involve reversing many previous assignments. This limits the net speedup which can be achieved across all iterations. Other factors such as the synchronization

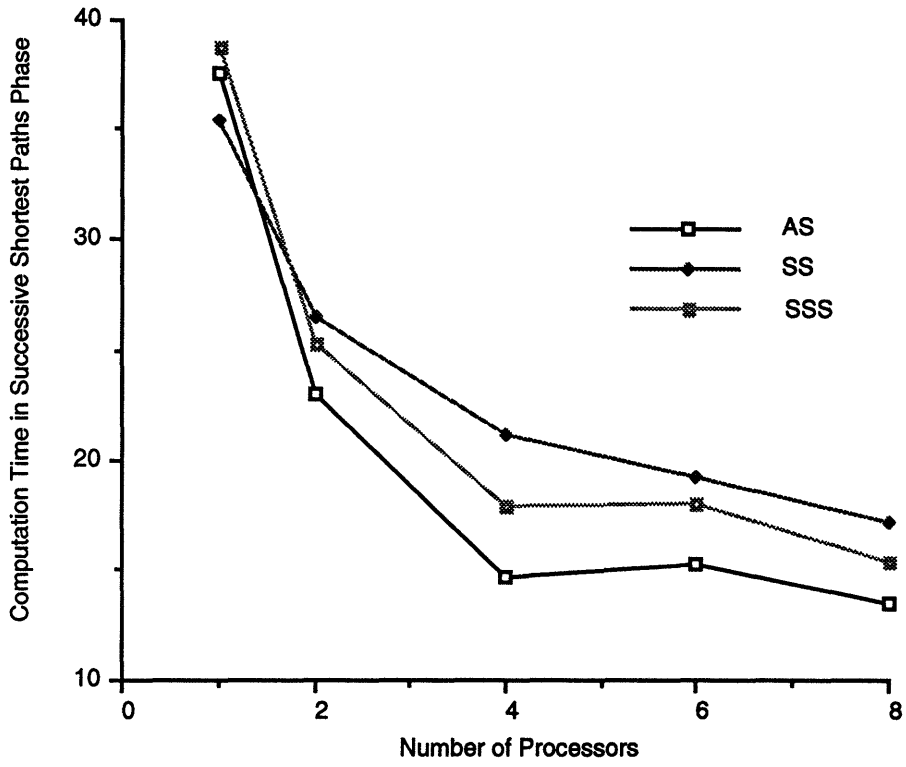


Figure 3: Computation time of parallel multinode Hungarian algorithms for 1000 person, 30% dense assignment problem, with cost range [1,1000], as a function of the number of processors used on the Encore Multimax.

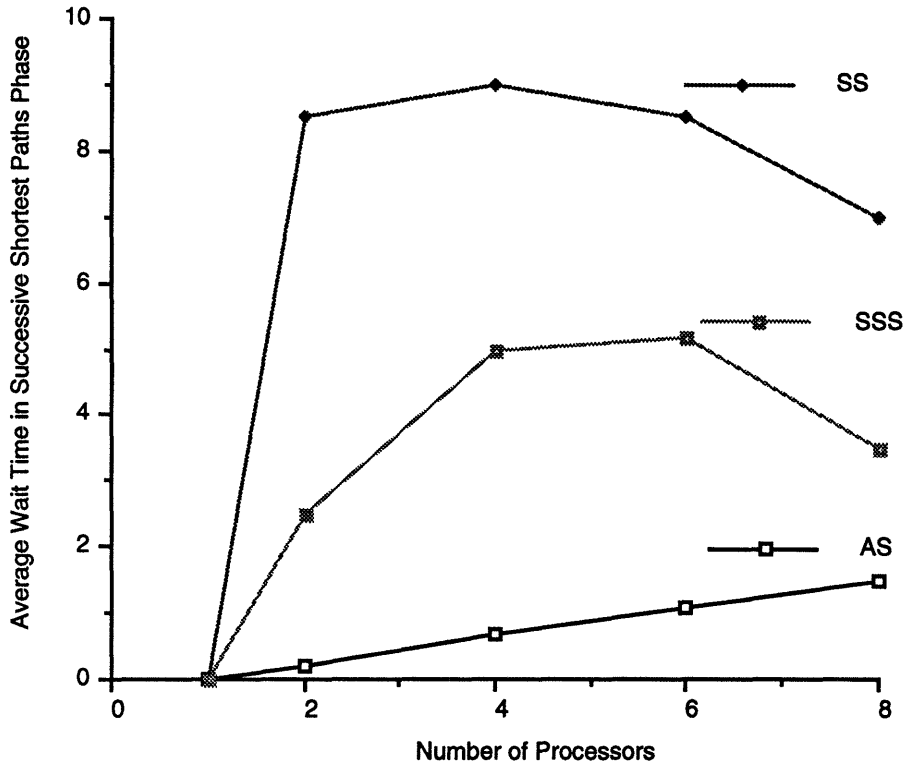


Figure 4: Average wait time per processor for 1000 person, 30% dense assignment problem, with cost range [1,1000], as a function of the number of processors used on the Encore Multimax.

overhead further limit the achievable speedup.

In terms of synchronous versus asynchronous algorithms, additional results in [15] indicate that, as the problems become sparser, the coordination overhead of the asynchronous algorithms may become larger than the overhead of the synchrous algorithms! This is because the time required to compute an augmenting path is much shorter for the sparse problems; this reduces the coordination overhead for synchronous algorithms because there is less variability in computation time across processors. However, it also increases the coordination overhead for asynchronous algorithms because more of the time is spent on modifying the master copy (rather than computation), so that the probability of access conflicts is increased. For a greater discussion of these topics, the reader is referred to [15].

Although the above discussion focused on assignment problems, Bertsekas and Castañon [16] have extended the single- path synchronous algorithm to min-cost network flow problems, and evaluated its performance using NETGEN uncapacitated transshipment problems. Extensions of the self- scheduled synchronous algorithm and the asynchronous algorithm to these classes of problems are straightforward, based on the results of Section 2.2.3.

2.3.3 Approximate Dual Coordinate Ascent

The auction algorithm for assignment problems has been tested widely across a variety of parallel implementations on different parallel computers. The simplicity of the auction algorithm makes it a natural candidate for parallel implementation. In addition, many variations exist which are amenable for either Gauss-Seidel, Jacobi or hybrid parallelization.

One of the earliest parallel implementations of the auction algorithm was reported by Phillips and Zenios [105] on the Connection Machine CM-2 for dense assignment problems (also subsequently by Wein and Zenios [121]). In their implementation, they used the large number of processors in the CM-2 to simultaneously compute bids for many different persons and to compute in parallel the bids of each person (thus using the hybrid approach discussed earlier). Their work was the first to illustrate the utility of massively parallel architectures for assignment problems.

A similar hybrid implementation was reported by Kempa *et al.* [78] on the Alliant FX/8 parallel computer. In their work, they experimented with various synchronous implementations of the auction algorithm for dense as-

signment problems. In their hybrid implementation, the vector processing capability of the Alliant's processors was used to compute in parallel the bid of each person, while the multiprocessor capability was used for computing in parallel multiple bids. For 1000-person dense assignment problems, with cost range [1,1000], Kempa *et al.* obtained total speedups of 8.6 for their hybrid auction algorithm using 8 vector processors. Subsequent work by Zaki [123] on the Alliant FX/8 produced similar results.

In [26], several synchronous and asynchronous implementations of the auction algorithm were developed and tested for dense and sparse assignment problems on different parallel computers (Encore Multimax, Alliant FX/8, DAP 510 and CM-2). Similar to the dual improvement algorithms, the SIMD implementations of the auction algorithm on the DAP 510 were extremely efficient, solving 1000-person dense assignment problems in under 3 seconds! Other important results in [26] are evaluations of the relative computation advantage of asynchronous auction algorithms versus their synchronous counterparts.

In [14], where a number of variations of the auction algorithm were implemented and evaluated on the Encore Multimax. The auction algorithm variations tested were:

1. *Synchronous Gauss-Seidel Auction* : Parallelization using only one bidder at a time.
2. *Synchronous Jacobi Auction* : A block Gauss- Seidel parallelization where each processor generates a bid from a different person.
3. *Synchronous Hybrid Auction* : A hybrid parallel algorithm, where processors are hierarchically organized into groups; each group computes a single bid in parallel, while multiple groups generate different bids.
4. *Asynchronous Jacobi Auction* : A block Gauss- Seidel parallel algorithm, where each processor generates a bid from a different person.
5. *Asynchronous Hybrid Auction* : A hybrid parallel algorithm where processors are divided into search processors (for computing a single bid) and bid processors (for computing multiple bids).

The most interesting results in [14] are the comparisons between the synchronous and asynchronous algorithms. For example, for 1000 person, 20% dense assignment problems, the synchronous Jacobi auction algorithm achieves a maximum speedup of 4, whereas the asynchronous Jacobi auction algorithm achieves speedups of nearly 6 due to its lower synchronization over-

head (which allows for efficient utilization of larger numbers of processors). This asynchronous advantage is the consequence of the strong asynchronous convergence results for the auction algorithm, which allow processors to perform computations with little need to maintain data consistency or integrity.

The asynchronous hybrid auction (AHA) algorithm of [14] is similar in structure to the parallel network simplex algorithms discussed previously [103, 5]. In each iteration of the AHA algorithm, some processors are designated as search processors, which evaluate arcs in the network and produce information for generation of bids. Other processors are bid processors, which process the information generated by the search processors and actually conduct the auction process. The search processors and the bid processors operate concurrently; thus, bids are often generated based on "old" information. In contrast with the network simplex algorithms, the AHA algorithm uses multiple bid processors, and does not fix *a priori* whether a processor is a search or a bid processor. Rather, there is a task queue containing a set of search and bid tasks which must be performed; these tasks are generated as part of the auction process. Whenever a processor is available, it proceeds to the queue and selects the next task.

Unlike the network simplex algorithms, the AHA algorithm does not need to recompute the information provided by the search processors in order to "validate" it (validation in the auction context would be very expensive, unlike in the network simplex context); instead, the AHA algorithm is based on the asynchronous convergence theory which guarantees that, if the information is outdated, the coordination mechanism in the auction algorithm will reject it. When tested across a set of 1000 person assignment problems with varying density, cost range [1,1000], the AHA algorithm was nearly twice as fast as the corresponding hybrid synchronous algorithm for every problem tested, highlighting the advantages of the asynchronous algorithm. For additional results on parallel auction algorithms, the reader should consult [14].

Unlike the auction algorithm for assignment problems, sequential implementations of the ϵ-relaxation method have been considerably slower than state-of-the-art min-cost network flow codes [18]. Thus, there have been fewer parallel implementations of the algorithm. A notable exception is the work of Li and Zenios [83] on the Connection Machine CM-2. They developed a modification of the ϵ-relaxation algorithm which assigns flows in fractional quantities, and implemented a parallel hybrid algorithm on the CM-2. Their

results indicate that, for some problems derived from military transportation, the CM-2 implementation of ϵ-relaxation is substantially faster than their network simplex implementation on the Cray Y-MP; for other classes of problems, the network simplex code was faster [83].

2.4 Summary

In this section we have discussed results on parallel algorithms for min-cost network flow problems. We have roughly characterized parallelization approaches into two types: single-node and multinode, depending on the level of the function which is done in parallel. The experimental results indicate that both approaches are limited in their ability to use parallelization to reduce computation requirements.

The key limit in single-node approaches is imposed by the density of the network; for sparse networks, the amount of work which can be performed in parallel is a smaller part of the overall computation time. For network simplex algorithms, sparse problems have fewer arcs which must be inspected to find a desirable pivot, so that adding search processors beyond a point in these algorithms actually slows down performance because of increased synchronization requirements. Similarly, in sequential shortest path algorithms and auction algorithms, sparsity limits the number of arcs which must be examined per node, so that again the ratio of parallel work to total work is reduced.

In contrast, the key limit in multinode approaches is imposed by the time-varying load across iterations. For parallel dual cost improvement algorithms and auction algorithms, our convergence theories [15, 14, 16] provide the basis for medium scale parallelization. The effectiveness of these parallel algorithms is not limited by sparsity, but rather by the fact that the number of nodes with positive surplus decreases with each iteration, so that the total amount of parallel work per iteration decreases. On average, the overall speedup using this approach for sparse problems is limited to factors of 3-5.

The computation experiments also highlight several critical issues such as the impact of parallel processor architectures and the choice of synchronous versus asynchronous algorithms. For dense network problems, the best processors are the massively parallel SIMD processors; however, these are limited in their ability to effectively use sparse data structures. When the problem density is below 1%, the best processors are shared-memory processors, which

are best- suited for using multinode parallelism and sparse data structures. The multinode parallelism is limited to using at most 10-12 processors in parallel; with these few processors, there is little contention for shared memory and communications resources.

In terms of synchronous versus asynchronous algorithms, it is interesting to find that, in many problems, asynchronous algorithms offer computational advantages. This is particularly true of auction algorithms; indeed, the asynchronous convergence theory allows for efficient combination of single-node and multinode parallelism within the same algorithm. In contrast, the asynchronous theory for dual cost improvement algorithms requires excessive data integrity, and results in inefficient algorithms for sparse problems. Development of a less restrictive asynchronous convergence theory remains a problem for future research.

A class of parallel algorithms for linear network flow problems, mentioned here for completeness and discussed in greater detail in Section 3.2.2 and 3.3.2, combines nonlinear perturbations of the linear objective function and then uses parallel algorithms for the resulting nonlinear program. The solution of the min-cost network flow problem, combining the PMD algorithm of [34] with the row-action algorithms of [30, 124] was suggested by Censor and Zenios [33], and extensive computational studies were conducted by Nielsen and Zenios [99, 100].

3 Nonlinear Network Optimization

3.1 Basic Algorithmic Ideas

In this section, we discuss three approaches to parallel solution of convex nonlinear network optimization problems: primal truncated Newton methods, dual coordinate ascent methods, and alternating direction methods. To handle problem for which the objective function is convex, but not *strictly* convex, the dual coordinate ascent methods need to be embedded in some sort of convexification scheme, such as the proximal minimization algorithm. Therefore, we also include some discussion of proximal minimization and related methods. Other approaches to parallel nonlinear network optimization are certainly worthy of study; for brevity, we restrict ourselves to methods which have already been substantially investigated in the specific context of

networks.

3.1.1 Primal Methods

One of the most efficient primal algorithms for solving the nonlinear network optimization program (NLNW) is the primal truncated Newton (PTN) algorithm [47], implemented within the active set framework [95]. The combination of both techniques for pure network problems is given in [45] and for generalized networks in [1]. PTN has received considerable attention both in solving large scale problems, and in parallel computation. We describe the algorithm in two steps: First we give a model Newton's algorithm for unconstrained optimization problems. Second, we discuss the active set method which reduces a constrained optimization problem into a sequence of (locally) unconstrained problems in lower dimensions. Our general reference for this section is Gill *et al.* [66].

A Truncated Newton Algorithm for Unconstrained Optimization: Consider the unconstrained problem

$$\min_{x \in \Re^n} \quad F(x), \tag{39}$$

where $F(x)$ is convex and twice continuously differentiable. The primal truncated Newton (PTN) algorithm starts from an arbitrary feasible point $x^0 \in \Re^n$ and generates a sequence $\{x^k\}$, $k = 1, 2, 3, \ldots$ such that

$$\lim_{k \to \infty} x_k = x^*,$$

where x^* belongs to the set of optimal solutions to (39) (*i.e.*, $x^* \in X^* = \{x | F(x) \leq F(y), \forall\ y \in \Re^n\}$). The iterative step of the algorithm is the following:

$$x^{k+1} = x^k + \alpha^k d^k. \tag{40}$$

$\{d^k\}$ is a sequence of descent directions computed by solving the system of (Newton's) equations :

$$\nabla^2 F(x^k) d^k = -\nabla F(x^k). \tag{41}$$

This system is solved inexactly (hence the term *truncated*), i.e., a solution d^k is obtained that satisfies

$$||\nabla^2 F(x^k) d^k - \nabla F(x^k)||_\infty \leq \eta^k. \tag{42}$$

A scale independent measure of the residual error is :

$$r^k = \frac{\|\nabla^2 F(x^{k-1})p^k + \nabla F(x^{k-1})\|_2}{\|\nabla F(x^{k-1})\|_2}. \tag{43}$$

The step direction is computed from (41) such that the condition $r^k \leq \eta^k$ is satisfied and the sequence $\{\eta^k\} \to 0$ as $k \to \infty$. $\{\alpha^k\}$ is a sequence of step sizes computed by solving

$$\alpha^k = \arg\min_{\alpha>0}\{F(x^k + \alpha d^k)\}, \tag{44}$$

(*i.e.*, at iteration k the scalar α^k is the step size that minimizes the function $F(x)$ along the direction p^k starting from point x^k). Computing α^k from equation (44) corresponds to an exact minimization calculation that may be expensive for large scale problems. It is also possible to use an inexact linesearch. The global convergence of the algorithm is preserved if the step length computed by inexact solution of (44) produces a sufficient descent of $F(x)$, satisfying Goldstein–Armijo type conditions (see *e.g.* [9]).

An Active Set Algorithm for Constrained Optimization:

Consider now the transformation of (NLNW) into a locally unconstrained problem. Following [95] we partition the matrix A into the form:

$$A = [\, B \;\; S \;\; N \,]. \tag{45}$$

B is a non–singular matrix of dimension $n \times n$ whose columns form a basis. For the case of network problems a basis can be constructed using a greedy heuristic, as given in [46]. First-order estimates of the Lagrange multipliers that correspond to this basis are obtained by solving for p the system $p^T B = -\nabla F(x^k)$. S is a matrix of dimension $n \times r$. It corresponds to the superbasic variables, *i.e.*, variables at their lower bound with negative reduced gradient, variables at their upper bound with positive reduced gradient, or free variables. For the set of superbasic variables it is possible to take a non-zero step that will reduce the objective function value. N is a matrix of dimension $n \times (m - n - r)$. It corresponds to the non-basic variables, *i.e.*, variables that have positive reduced gradient and are at their lower bound, or have a negative reduced gradient and are at their upper bound.

We use $\mathcal{B}$, $\mathcal{S}$ and $\mathcal{N}$ to denote the sets of basic, superbasic and non–basic variables respectively. The vector x^k is partitioned into

$$x^k = [\, x_B^k \;\; x_S^k \;\; x_N^k \,]. \tag{46}$$

$x_B^k \in \Re^m$ are the basic variables, $x_S^k \in \Re^r$ are the superbasic variables and $x_N^k \in \Re^{n-m-r}$ denote non–basic variables. Non–basic variables — for a given partitioning (45)–(46) — are kept fixed to one of their bounds. If we now partition the step direction d as

$$d^k = [\, d_B^k \;\; d_S^k \;\; d_N^k \,] \tag{47}$$

we require $d_N^k \equiv 0$ (*i.e.*, non–basic variables remain fixed) and furthermore d^k should belong to the nullspace of A (*i.e.*, $Ad^k = 0$), so that d^k is a feasible direction. Hence d^k must satisfy

$$Bd_B^k + Sd_S^k = 0 \;\text{ or }\; d_B^k = -(B^{-1}S)d_S^k. \tag{48}$$

If the superbasic variables are strictly between their bounds and the basis B is maximal as defined in [46] (*i.e.*, a non–zero step in the basic variables x_B is possible for any choice of direction $(d_B^k \;\; d_S^k \;\; 0)$), then the problem is locally unconstrained with respect to the superbasic variables. Hence a descent direction for d_S^k can be obtained by solving the (projected) Newton's equations:

$$(Z^T\nabla^2 F(x^k)Z)d_S^k = -Z^T\nabla F(x^k) + \eta^k e, \tag{49}$$

where Z is a basis for the nullspace of A defined as

$$Z = \begin{bmatrix} -B^{-1}S \\ I \\ 0 \end{bmatrix} \tag{50}$$

The primary computational requirement of the algorithm is in solving the system of equations (49) of dimension $r \times r$. This system is solved using a conjugate gradient method with a preconditioner matrix equal to the inverse of the diagonal of the reduced Hessian matrix $Z^T\nabla^2 F(x)Z$. Calculation of p_B^k from (48) involves only a matrix–vector product and is in general an easy computation. The partitioning of the variables and the matrix A into basic, superbasic and non–basic elements is also in general very fast. For some of the bigger problems reported in the literature the solution of system (49) takes as much as 99% of the overall solution time. Efforts in parallelizing the PTN algorithm have concentrated in the solution of projected Newton equations.

3.1.2 Dual Coordinate Ascent Methods

Now consider a separable, nonlinear network optimization problem of the form (NLNW). where the cost component functions $f_{ij} : \Re \rightarrow \Re$ are for the moment assumed to be *strictly* convex and lower semicontinuous. For any flow $x \in \Re^m$, we let $g_i(x)$ denote the surplus at node i under the flow x (we make the dependence on x explicit because some nonlinear network algorithms manipulate several flow vectors simultaneously). Assume that (NLNW) is feasible and has an optimal solution. Defining for each $(i,j) \in \mathcal{A}$,

$$\bar{f}_{ij}(x_{ij}) = \begin{cases} f_{ij}(x_{ij}) , & l_{ij} \leq x_{ij} \leq u_{ij} \\ +\infty & \text{otherwise} \end{cases} , \tag{51}$$

the problem may be written

$$\begin{array}{ll} \text{minimize} & \bar{f}(x) \triangleq \sum\limits_{(i,j)\in\mathcal{A}} \bar{f}_{ij}(x_{ij}) \\ \text{such that} & Ax = b. \end{array} \tag{52}$$

Attaching a vector of Lagrange multipliers $p \in \Re^n$ to the equality constraints, one obtains the Lagrangian

$$\begin{aligned} L(x,p) &= \sum_{(i,j)\in\mathcal{A}} \bar{f}_{ij}(x_{ij}) - p^T(Ax - b) \\ &= \sum_{(i,j)\in\mathcal{A}} \left(\bar{f}_{ij}(x_{ij}) - (p_i - p_j)x_{ij}\right) - p^T b. \end{aligned}$$

Therefore, the dual functional of the problem is

$$q(p) = \inf_{x\in\Re^{|\mathcal{A}|}} \{L(x,p)\} = \sum_{(i,j)\in\mathcal{A}} q_{ij}(p_i - p_j) - p^T b, \tag{53}$$

where

$$q_{ij}(t_{ij}) = \inf_{x_{ij}\in\Re} \left\{\bar{f}_{ij}(x_{ij}) - t_{ij}x_{ij}\right\}. \tag{54}$$

Thus, we may define a problem dual to (NLNW) and (52) to be

$$\begin{array}{ll} \text{maximize} & q(p) \\ \text{such that} & p \in \Re^n \quad . \end{array} \tag{55}$$

Now, each q_{ij} is the pointwise infimum of a set of affine (hence concave) functions in t_{ij}, and is necessarily concave. Thus, q is concave. Basic nonlinear programming duality theory implies that the optimal values of (NLNW)

and (55) are equal. Readers familiar with aspects of convex analysis [109, 112] will recognize that q_{ij} is the negative of the convex conjugate of $\bar{f}_{ij}$. In simple terms, the following equivalences therefore hold:

$$\bar{f}_{ij} \text{ has a subgradient of slope } t_{ij} \text{ at } x_{ij} \tag{56}$$

$$\iff q_{ij} \text{ has a supergradient of slope } -x_{ij} \text{ at } t_{ij} \tag{57}$$

$$\iff x_{ij} \text{ attains the infimum in (54).} \tag{58}$$

The strict convexity of the f_{ij} implies strict convexity of the $\bar{f}_{ij}$. This means that at any two distinct points $x_{ij}^1, x_{ij}^2 \in [l_{ij}, u_{ij}]$, $\bar{f}_{ij}$ cannot have subgradients with the same slope. It follows that for each argument t_{ij}, q_{ij} has exactly one supergradient, that is, q_{ij} is differentiable. If follows that (55) is a differentiable, unconstrained minimization problem. The key idea in parallel dual methods for network optimization is to exploit this extremely advantageous dual structure.

The differentiability and convexity of q imply that, from any non-maximizing p, it is possible to increase q by changing only a single component p_i of p. This property suggests the following iteration for maximizing $q(p)$:

(i) Given a price vector $p(t)$, choose any node $i \in \mathcal{N}$.

(ii) Compute $p(t+1)$ such that $p_j(t+1) = p_j(t)$ for all $j \neq i$, while $p_i(t+1)$ maximizes $q(p)$ with respect to the i-coordinate, all other coordinates being held fixed at $p_j(t)$.

It can be shown that, so long as each node $i \in \mathcal{N}$ is selected an infinite number of times, that the sequence of price vectors $p(t)$ generated by this iteration converges to an optimal solution of the dual problem (55) [22, Section 5.5]. Algorithms of this general sort have been discussed in such diverse sources as [21, 30, 41, 42, 74]. They are usually called *relaxation* methods, because maximizing the dual cost with respect to the coordinate p_i is equivalent to adjusting the primal solution x so that the flow constraint for node i is satisfied while constraints for other nodes are relaxed. To understand this, it is necessary to exploit the duality relationship between (NLNW)/(52) and (55) encapsulated by (56)-(58).

The equivalent conditions (56)-(58) are known as *complementary slackness.* Dual methods enforce these conditions at all times, where we define

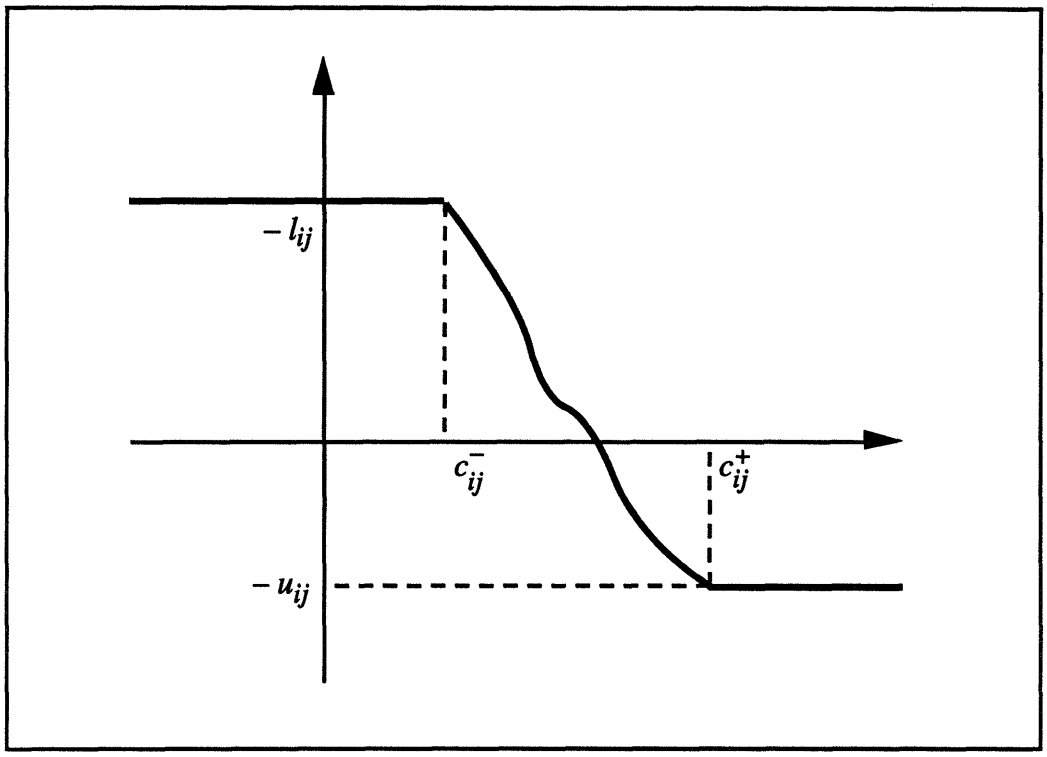


Figure 5: Example of the function ∇q_{ij}.

the *tension* t_{ij} of arc (i,j) to be $p_i - p_j$. We denote by $x(p)$ the unique flow vector satisfying complementary slackness with the price vector p on all arcs, whence

$$x_{ij}(p) = -\nabla q_{ij}(p_i - p_j) \quad .$$

Let c_{ij}^- be the derivative of f_{ij} at l_{ij}, and c_{ij}^+ be the derivative of f_{ij} at u_{ij}. For values of $t_{ij} = p_i - p_j$ below c_{ij}^-, the infimum in (54) is attained for $x_{ij} = l_{ij}$, so $x_{ij}(p)$ must be at its lower bound of l_{ij}. If $p_i - p_j \geq c_{ij}^+$, then the infimum is attained at $x_{ij} = u_{ij}$, and so $x_{ij}(p)$ is set to its upper bound. For $p_i - p_j \in [c_{ij}^-, c_{ij}^+]$, $x_{ij}(p) = -\nabla q_{ij}(p_i - p_j)$, by the concavity of q_{ij}, is a nondecreasing function of $p_i - p_j$. Thus, ∇q_{ij} is constant at $-l_{ij}$ for arguments below c_{ij}^-, constant at $-u_{ij}$ for arguments above c_{ij}^+, and nonincreasing in between, as shown in Figure 5. Accordingly, q_{ij} itself is linear with slope $-l_{ij}$ for arguments below c_{ij}^-, linear with slope $-u_{ij}$ for arguments above c_{ij}^+, and

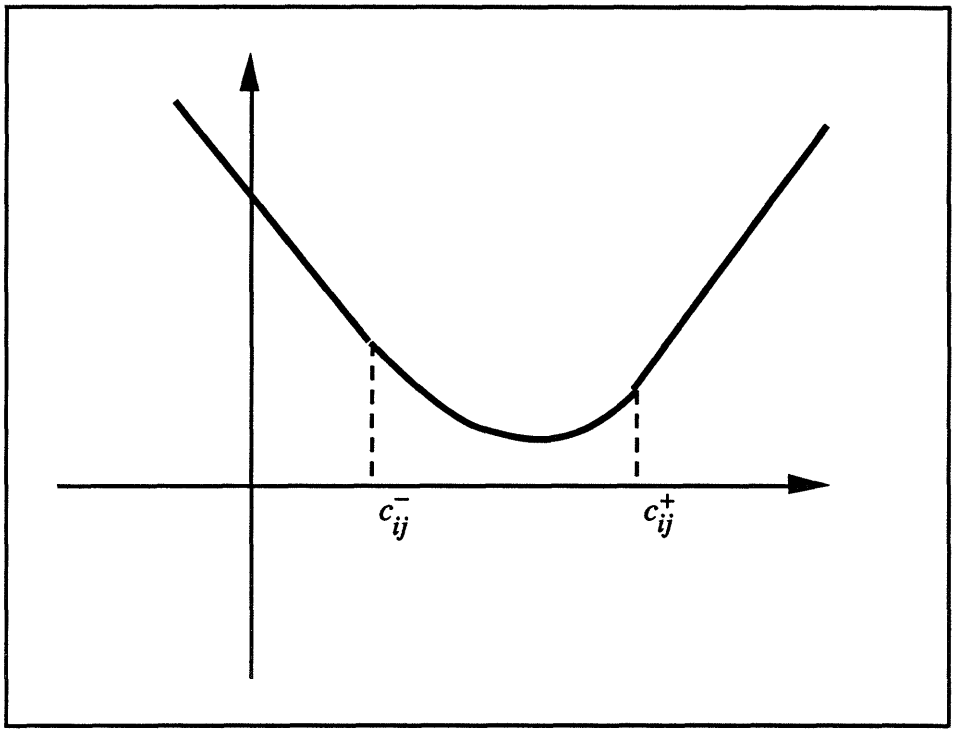


Figure 6: Example of the function q_{ij}.

concave in between, as depicted in Figure 6. Now let us consider the partial derivative of the entire dual functional q with respect to the variable p_i:

$$\begin{aligned}
\frac{\partial}{\partial p_i} q(p) &= \frac{\partial}{\partial p_i} \left[\sum_{(k,l)\in\mathcal{A}} q_{kl}\,(p_k - p_l) \right] + \frac{\partial}{\partial p_i} \left[p^T b \right] \\
&= \sum_{j:(i,j)\in\mathcal{A}} \nabla q_{ij}(p_i - p_j) \; - \sum_{j:(j,i)\in\mathcal{A}} \nabla q_{ji}(p_j - p_i) \; + \; b_i \\
&= \; - \sum_{j:(i,j)\in\mathcal{A}} x_{ij}(p) \; + \sum_{j:(j,i)\in\mathcal{A}} x_{ji}(p) \; + \; b_i,
\end{aligned}$$

which is just the *surplus* $g_i(x(p))$ at node i under the flow $x(p)$. Thus, the coordinate ascent algorithm may be restated

(i′) Given $p(t)$, choose any node $i \in \mathcal{N}$.

(ii′) Compute $p(t+1)$, equal to $p(t)$ in all but the i^{th} coordinate, such that $x(p(t+1)$ meets the flow balance constraint $g_i(x(p(t+1))) = 0$ at node i.

In the parlance of solving systems of nonlinear equations, we are "relaxing" flow balance equation i at each iteration (although "enforcing" might be a more appropriate word). Below, we will use the phrase "relaxing node i" to denote applying the operation (ii′) at node i. Note that if $g_i(x(p))$ is positive, one must raise p_i in order to "drive away" flow from i; raising p_i increases flow on the outgoing arcs of i, and reduces flow on i's incoming arcs. Conversely, if $g_i(x(p)) < 0$, then p_i must be *reduced* in order to attract flow to i.

A number of variations on the basic iteration are possible. For instance, instead of exactly maximizing q with respect to p_i, one may elect only to reduce the magnitude of its gradient by some factor $\delta \in [0, 1)$, as follows; see [22, Section 5] and [21]:

(i″) Given $p(t)$, choose any node $i \in \mathcal{N}$.

(ii″) If $g_i(x(p(t))) = 0$, do nothing. Otherwise, compute $p(t+1)$, equal to $p(t)$ in all but the i^{th} coordinate, such that

$$\begin{aligned} 0 \le g_i(x(p(t+1))) &\le \delta g_i(x(p(t))) \quad \text{if} \quad g_i(x(p(t))) > 0 \\ \delta g_i(x(p(t))) &\le g_i(x(p(t+1))) \le 0 \quad \text{if} \quad g_i(x(p(t))) < 0. \end{aligned}$$

Another possibility is to compute a maximizing value $\tilde{p}_i$ for p_i, but then set

$$p_i(t+1) = (1-\gamma)p_i(t) + \gamma\tilde{p}_i,$$

where $\gamma \in (0, 1)$ is some stepsize.

Algorithms in the dual relaxation family are also known as "row-action" methods [28], in that, at each iteration, they select a single row of the constraint system $Ax = b$, and satisfy it at the expense of the other rows. In the case that the f_{ij} are differentiable, the flow $x(p(t+1))$ obtained by maximizing the dual cost along the p_i coordinate is equivalent to the projection of $x(p(t))$ onto the hyperplane given by the i^{th} flow balance constraint with respect to a special distance measure, or "D-function" derived from the f_{ij}. When the f_{ij} have certain special properties, and flow bound constraints are

absent, it is actually possible to prove convergence of the method without any appeal to the dual problem; see [24].

However, the main contribution of the row-action literature, from our perspective, is not in the handling of the equality constraints $Ax = b$, but in treatment of the *inequality* constraints $x \geq l$ and $x \leq u$, for which some use of duality is required [24, 30]. Specifically, one may dualize the interval constraints $l_{ij} \leq x_{ij} \leq u_{ij}$ by attaching multipliers $v_{ij} \leq 0$ and $w_{ij} \geq 0$, respectively. This yields a new dual functional

$$\hat{q}(p, v, w) = \inf_{x \in \Re^m} \left\{ \sum_{(i,j) \in \mathcal{A}} f_{ij}(x_{ij}) - (p_i - p_j)x_{ij} - v_{ij}(x_{ij} - l_{ij}) - w_{ij}(u_{ij} - x_{ij}) \right\}$$

to be maximized with respect to $v \leq 0$, $w \geq 0$, and $p \in \Re^n$. It is possible to construct an algorithm that successively maximizes the functional $\hat{q}(p, v, w)$ along not only the individual p_i coordinates, but also the individual v_{ij} and w_{ij} coordinates. Furthermore, for any optimal solution, the Karush-Kuhn-Tucker conditions guarantee that at most one of v_{ij} and w_{ij} will be nonzero for any (i, j), so the two variables may be condensed into a single quantity z_{ij} via $v_{ij} = (z_{ij})^-$ and $w_{ij} = (z_{ij})^+$. Censor and Lent [30] give a procedure for handling such combined dual variables. When $z_{ij} \geq 0$, this procedure is equivalent to a dual-objective-maximizing step along the w_{ij} coordinate, followed by a maximizing step along the v_{ij} coordinate. When $z_{ij} < 0$, the roles of v_{ij} and w_{ij} are reversed.

The row action literature also suggests, for certain specific forms of the objective function, a special "MART" step, which is an easily computed secant approximation to the true relaxation step [31].

3.1.3 Proximal Point Methods

We now consider the problem (NLNW) with the assumption that the f_{ij} are convex, but not necessarily *strictly* convex. The dual coordinate methods described above rely on the differentiability of the dual problem (55), and hence on the strict convexity of the primal objective. When applied to problems whose objective functions are not strictly convex, including linear programs, such dual coordinate methods may "jam" at a primal-infeasible (dual-suboptimal) point, due to the absence of dual smoothness. For purely

linear problems, one may attempt to overcome this difficulty by using the special techniques of Section 2.1.3; however, it may be difficult to maintain much parallelism in the later stages of such methods. To address these deffi-ciencies, and also to handle general (non-strictly) convex objectives, renewed attention has turned to the solution of non-strictly-convex problems via a *se-quence* of strictly convex ones. So far, attention has centered on the *proximal mininization algorithm* (PMA) [87, 110] and variants thereof [34]. There are several ways to motivate the PMA, but the simplest, perhaps, is to consider the following problem equivalent to (NLNW) [22, pp. 232-243]:

$$\begin{array}{ll} \text{minimize} & \left[\sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij})\right] + \frac{1}{2\lambda}\|x-y\|^2 \\ \text{such that} & Ax = b \\ & l \le x \le u \\ & x, y \in \Re^m. \end{array} \tag{59}$$

Here, λ is a positive scalar. This problem is equivalent to (NLNW), the optimum being attained for $x = y = x^*$, where x^* solves (NLNW). One can imagine solving (59) by a block Gauss-Seidel method [22, Section 3.3.5] by which one repeatedly minimizes the objective with respect to x, holding y fixed, and then minimizes with respect to y, holding x fixed. The latter operation just sets $y = x$, so one obtains the algorithm

$$\begin{aligned} x(t+1) &= \arg\min_{\substack{Ax=b \\ l\le x\le u}} \left\{ \sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij}) + \frac{1}{2\lambda}(x_{ij}-y_{ij})^2 \right\} \\ y(t+1) &= x(t+1), \end{aligned}$$

or simply

$$x(t+1) = \arg\min_{\substack{Ax=b \\ l\le x\le u}} \left\{ \sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij}) + \frac{1}{2\lambda}(x_{ij}-x_{ij}(t))^2 \right\}. \tag{60}$$

Each of the successive subproblems in the method (60) is strictly convex due to the strong convexity of the terms $\frac{1}{2\lambda}(x_{ij}-x_{ij}(t))^2$. Therefore, $x(t+1)$ is unique.

The convergence of the PMA can be proven even if λ is replaced by some $\lambda(t)$ that varies with t, so long as $\inf_{t\ge 0}\{\lambda(t)\} > 0$. This can be

shown from first principles [22, pp. 232-243], or by appeal to the general theory of the proximal point algorithm [111, 110]. The latter body of theory establishes that *approximate* calculation of each iterate $x(t+1)$ is permissible, an important practical consideration. Heuristically, one may think of (60) as an "implicit" gradient method for (NLNW) in which the step $x(t+1)-x(t)$ is codirectional with the negative gradient of the objective not at $x(t)$, as is the case in most gradient methods, but at $x(t+1)$. Similar results using even weaker assumptions on $\{\lambda(t)\}$ are possible [25].

Without the background of established proximal point theory [111, 110], the choice of $\frac{1}{2\lambda}\|x-y\|^2$ as the strictly convexifying term in (59) seems somewhat arbitrary. For example, $\frac{1}{2\lambda}\|x-y\|^3$ might in principle have served just as well. Recent theoretical advances [34] indicate that any "D-function" $D(x,y)$ of the form described in [30] may take the place of $\frac{1}{2}\|x-y\|^2$ in the analysis. Among the properties of such D-functions are

$$\begin{aligned} &D(x,y) \geq 0 \quad \forall x,y \\ &D(x,y) = 0 \quad \Longleftrightarrow \quad x = y \\ &D(x,y) \text{ strictly convex in } x. \end{aligned}$$

Of particular interest is the choice of $D(x,y)$ to be

$$\sum_{(i,j)\in\mathcal{A}} \left[x_{ij} \log\left(\frac{x_{ij}}{y_{ij}}\right) - (x_{ij}-y_{ij}) \right],$$

sometimes referred to as the *Kullback-Leibler divergence* of x and y. For additional information on the theory of such methods, see [118, 53, 120].

3.1.4 Alternating Direction Methods

Alternating direction methods are another class of parallelizable algorithms that do not require strict convexity of the objective. Consider a general optimization problem of the form

$$\begin{aligned} \text{mininize} \quad & h_1(x) + h_2(z) \\ \text{such that} \quad & z = Mx, \end{aligned}$$

where $h_1 : \Re^r \to (-\infty, \infty]$ and $h_2 : \Re^s \to (-\infty, \infty]$ are convex, and M is a $r \times s$ matrix. A standard augmented Lagrangian approach to this problem,

the *method of multipliers* (see [9] for a comprehensive survey), is, for some scalar $\lambda > 0$,

$$(x(t+1), z(t+1)) = \arg\min_{(x,z)} \left\{ h_1(x) + h_2(z) + \langle \pi(t), Mx - z \rangle + \frac{\lambda}{2} \|Mx - z\|^2 \right\} \quad (61)$$

$$\pi(t+1) = \pi(t) + \frac{\lambda}{2}(Mx(t+1) - z(t+1)). \quad (62)$$

Here, $\{\pi(t)\} \subseteq \Re^q$ is a sequence of Lagrange multiplier estimates for the the constraint system $Mx - z = 0$. The minimization in (61) is complicated by the presence of the nonseparable $z^T Mx$ term in $\|Mx - z\|^2$. However, one conceivable way to solve for $x(t+1)$ and $z(t+1)$ in (61) might be to minimize the augmented Lagrangian alternately with respect to x, with z held fixed, and then with respect to z with x held constant, repeating the process until both x and z converge to limiting values. Interestingly, it turns out to be possible to proceed directly to the multiplier update (62) after a single cycle through this procedure, without truly minimizing the augmented Lagrangian. The resulting method may be written

$$\begin{aligned} x(t+1) &= \arg\min_x \left\{ h_1(x) + \langle \pi(t), Mx \rangle + \frac{\lambda}{2} \|Mx - z(t)\|^2 \right\} \\ z(t+1) &= \arg\min_z \left\{ h_2(z) - \langle \pi(t), z \rangle + \frac{\lambda}{2} \|Mx(t+1) - z\|^2 \right\} \\ \pi(t+1) &= \pi(t) + \frac{\lambda}{2}(Mx(t+1) - z(t+1)), \end{aligned}$$

and is called the *alternating direction method of multipliers.* It was introduced in [69, 64, 62]; see also [22, pp. 253-261]. Note that the problem of nonseparability of x and z in (61) has been removed, and also that h_1 and h_2 do not appear in the same minimization. In [63], Gabay made a connection between the alternating direction method of multipliers and a generalization of an alternating direction method for solving discretized differential equations [84]; see [51, 55] for comprehensive treatments.

One way to apply this method to a separable, convex-cost network problem of the form (NLNW) (without any assumption of *strict* convexity) is to

let

$$
\begin{aligned}
r &= m \\
s &= 2m \\
z &= (\eta, \xi) \in \Re^m \times \Re^m \\
M &= \begin{bmatrix} I \\ I \end{bmatrix} \\
h_1(x) &= \sum_{(i,j)\in\mathcal{A}} \bar{f}_{ij}(x_{ij}) \qquad \text{(as defined in (51))} \\
h_2(\eta,\xi) &= \begin{cases} 0 & \sum_{j:(i,j)\in\mathcal{A}} \eta_{ij} - \sum_{j:(j,i)\in\mathcal{A}} \xi_{ji} = b_i \quad \forall i \in \mathcal{N} \\ +\infty & \text{otherwise} \end{cases}.
\end{aligned}
$$

The idea here is to let $z = (\eta, \xi) \in \Re^m \times \Re^m$, where η_{ij} is the flow on (i,j) as perceived by node i, while ξ_{ij} is the flow on (i,j) as perceived by node j. The objective functon term $h_2(\eta,\xi)$ essentially enforces the constraint that the perceived flows be in balance at each node, while the constraint $z = Mx$ requires that each η_{ij} and ξ_{ij} take a common value x_{ij}, that is, that flow be conserved along arcs. The function h_1 plays the role of the original objective function of (NLNW), and also enforces the flow bound constraints. Applying the alternating direction method of multipliers to this setup reduces, after considerable algebra [54, 51], to the algorithm

$$
\begin{aligned}
\hat{x}_{ij}(t) &= x_{ij}(t) + \frac{g_i(x(t))}{d(i)} - \frac{g_j(x(t))}{d(j)} \\
x_{ij}(t+1) &= \operatorname*{arg\,min}_{l_{ij}\le x_{ij}\le u_{ij}} \left\{ f_{ij}(x_{ij}) - (p_i - p_j)x_{ij} + \frac{\lambda}{2}\left(x_{ij} - \hat{x}_{ij}(t)\right)^2 \right\} \\
p_i(t+1) &= p_i(t) + \frac{\lambda}{d(i)} g_i(x(t+1)),
\end{aligned}
$$

where $d(i)$ denotes the degree of node i in the network. This iteration is basic form of the *alternating step method*; see also [22, p. 254]. The initial flow vector $x(0)$ and node prices $p(0)$ are arbitrary, and need fulfill neither feasibility nor complementary slackness conditions. For linear or quadratic f_{ij}, the one-dimensional minimization required to compute the $x_{ij}(t+1)$ can be done analytically. An "overrelaxed" version of the method is also possible;

letting $\{\rho(t)\}_{t=1}^{\infty}$ be a sequence of scalars such that

$$0 < \inf_{t\geq 1}\{\rho(t)\} \leq \sup_{t\geq 1}\{\rho(t)\} < 2,$$

one can derive the more general alternating step method [54]

$$\hat{x}_{ij}(t) = y_{ij}(t) + \frac{g_i(y(t))}{d(i)} - \frac{g_j(y(t))}{d(j)} \tag{63}$$

$$x_{ij}(t+1) = \underset{l_{ij}\leq x_{ij}\leq u_{ij}}{\arg\min}\left\{f_{ij}(x_{ij}) - (p_i - p_j)x_{ij} + \frac{\lambda}{2}(x_{ij} - \hat{x}_{ij}(t))^2\right\} \tag{64}$$

$$p_i(t+1) = p_i(t) + \frac{\lambda\rho(t)}{d(i)}g_i(x(t+1)) \tag{65}$$

$$y_{ij}(t+1) = (1-\rho(t))y_{ij}(t) + \rho(t)x_{ij}(t+1). \tag{66}$$

Here, the initial flow vector $y(0)$ and node prices $p(0)$ are again arbitrary. Approximate calculation of the $x_{ij}(t+1)$ is possible [55, 54].

For convex-cost transportation problems, a different application of the alternating direction method of multipliers is given in [22, Exercise 5.3.10], [51, Section 7.3], and [56]. This approach decomposes the problem network somewhat less aggressively than (63)-(66).

3.2 Parallelization Ideas

3.2.1 Primal Methods

Parallelization of the primal truncated Newton algorithm has concentrated on the linear algebra calculations required in solving Newton's equations (49). The first approach, proposed in [130], views the matrix $Z^T\nabla^2 F(x^k)Z$ as a general, sparse matrix without any special structure. The conjugate gradient method is used to solve this system. In order to implement this method we need to form the products Zv, Hv and Z^Tv, where v is a vector of the conjugate directions. On shared memory systems these products can be parallelized very efficiently by distributing the Hessian matrix row-wise across multiple processors, while distributing the Z matrix column-wise.

Another important feature of this algorithm is that it can be implemented efficiently on systems with vector features. To this end we need data structures that allow the algorithm to form compact, dense, vectors from the

sparse matrix representations of Z and H. Details on implementation designs for PTN on a vector architecture are rather technical. See [128] for a description of the data structures and a complete discussion of implementations.

Similar parallelization ideas for PTN have been proposed in [82] for cases when the objective function is *partially separable*, *i.e.*, it is of the form $F(x) = \sum f_i(x)$ where each element function $f_i(x)$ has a Hessian matrix of lower rank than the original problem. These proposals have been very efficient in solving some large unconstrained optimization problems. However, they do not offer any real advantage over the methods discussed above for the network problems that are, usually, separable.

An alternative procedure for parallelizing the primal truncated Newton algorithm exploits the sparsity structure of the network basis and partitions Newton's equations in independent blocks. The blocks can then be distributed among multiple processors for solution using again the conjugate gradient solver. The *block-partitioned* truncated Newton method was developed in [125]. To describe this algorithm, we need to impose an ordering of the arcs (i,j). We will use $t \sim (i,j)$ to denote the lexicographic order of arc (i,j), and hence x_t is the t-th variable that corresponds to flow x_{ij} on arc (i,j).

Block–partitioning of Newton's Equations:

We return now to equation (49), and try to identify a partitioning of the matrix $(Z^T \nabla^2 F(x^k) Z)$ into a block-diagonal form. Recall that

$$Z = \begin{bmatrix} -B^{-1}S \\ I \\ 0 \end{bmatrix} \tag{67}$$

and that the function $F(x) = \sum_{t=1}^{n} F_t(x_t)$ is separable, so that the Hessian matrix is diagonal. If we ignore momentarily the dense submatrix $(B^{-1}S)$ and assume that

$$Z = \hat{Z} = \begin{bmatrix} I \\ 0 \end{bmatrix} \tag{68}$$

(the identity I and null matrix 0 chosen such that Z is conformable to $\nabla^2 F(x^k)$) then the product

$$\hat{Z}^T \nabla^2 F(x^k) \hat{Z}$$

is a matrix of the form diag[H_I 0], where H_I is a diagonal matrix with the t-th diagonal element given by $\frac{\partial^2 F_t(x_t^k)}{\partial x_t^2}$. Hence, the complication in partitioning (49) is the presence of the submatrix $(B^{-1}S)$. The structure of this submatrix is examined next.

The structure of $(B^{-1}S)$:

The matrix B is a basis for the network flows of problem (NLNW). It is well-known — see, *e.g.*, Dantzig [44], Kennington and Helgason [80] or Glover et al. [68, 67] — that the basis of a pure network problem is a lower triangular matrix. The graph associated with this basis matrix is a rooted tree. The basis of a generalized network is characterized by the following theorem (see, *e.g.*, [44, p.421]).

Theorem 1 *Any basis B of a generalized network problem can be put in the form*

$$B = \begin{bmatrix} B^1 & & & & & \\ & B^2 & & & & \\ & & \ddots & & & \\ & & & B^\ell & & \\ & & & & \ddots & \\ & & & & & B^L \end{bmatrix}$$

where each square submatrix B^ℓ is lower triangular with at most one element above the diagonal.

The graph associated with each submatrix B^ℓ is a tree with one additional arc, making it either a rooted tree or a tree with exactly one cycle, and is called a quasi–tree (abbreviated: q–tree). The graph associated with a generalized network basis is a forest of q–trees.

To describe the structure of $(B^{-1}S)$ we first define the *basic–equivalent–paths* (BEP) for a superbasic variable x_t with incident nodes (i, j). For pure network problems it is the set of arcs on the basis tree that lead from node j to node i. The arcs on BEP together with arc $t \sim (i, j)$ create a loop. In the case of a generalized network, it is the set of arcs that lead from nodes i and j to either the root of the tree or the cycle of a q-tree; the BEP includes all arcs on the cycle. The t-th column of $(B^{-1}S)$ has non–zero entries corresponding to the BEP of the t-th superbasic variable. The numerical values of $(B^{-1}S)$

are ± 1 for pure network problems and arbitrary real numbers for generalized networks; the numerical values are of no consequence to our development.

To illustrate the preceding discussion we show in Figure 7 the basis of a pure network problem together with the BEP for a superbasic arc and the corresponding column of $(B^{-1}S)$. Figure 8 illustrates the same definitions for generalized network problems.

The matrix $(B^{-1}S)$ can be partitioned into submatrices without overlapping rows if the columns of each submatrix have BEP with no basic arcs in common with the columns of any other submatrix.

Partitioning of $(B^{-1}S)$ for Pure Networks:

Let β_t denote an ordered set of binary indices such that $(\beta_t)_l = 1$ if the l-th arc $(i,j) \in \mathcal{B}$ is in the BEP of the t-th arc $(i',j') \in \mathcal{S}$, and $(\beta_t)_l = 0$ otherwise. We seek a partitioning of the set $\mathcal{S}$ into K disjoint independent subsets, say $\mathcal{S}^k$, $k \in \mathcal{K} = \{1, 2, \ldots, K\}$ such that

$$\mathcal{S} = \bigcup_{k=1}^{K} \mathcal{S}^k \tag{69}$$

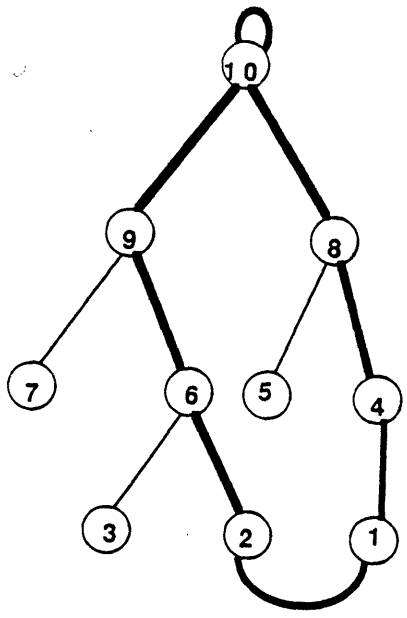
$$\text{and } t \in \mathcal{S}^{k_1} \quad \text{and} \quad u \in \mathcal{S}^{k_2} \quad \text{iff} \quad \beta_t \wedge \beta_u = 0 \; \forall \, k_1 \neq k_2 \in \mathcal{K} \tag{70}$$

(*i.e.*, the sets β_t and β_u have no overlapping non–zeroes, and hence there is no common basic arc in the BEP of t-th and u-th superbasic arcs).

Escudero [59] was the first one to propose the partitioning of $\mathcal{S}$ into independent superbasic subsets $\mathcal{S}^k$ according to equation (69)–(70), for *replicated* pure networks. Replicated networks consist of subnetworks with identical structure and are connected by linking arcs. (These linking arcs represent inventory flow for his problems that are multiperiod networks.) In the same reference Escudero gives a procedure for identifying the independent superbasic sets $\mathcal{S}^k$.

The problem of identifying independent superbasic sets can be formulated as a problems from graph theory (*i.e.*, finding connected components or articulation points of the adjacency graph of the matrix $Z^T Z$). It can be solved efficiently using algorithms developed in [117]. For additional details see [125].

Partitioning of $(B^{-1}S)$ for Generalized Networks: The graph partitioning schemes discussed above for pure network problems can also be applied in the case of generalized networks. It is, however, possible to develop



	1	2	3	4	5	6	7	8	9	10
1	*									
2		*								
3			*							
4	*			*						
5					*					
6		*	*			*				
7							*			
8				*	*			*		
9						*	*		*	
10								*	*	*

Basis of Pure Network Problem

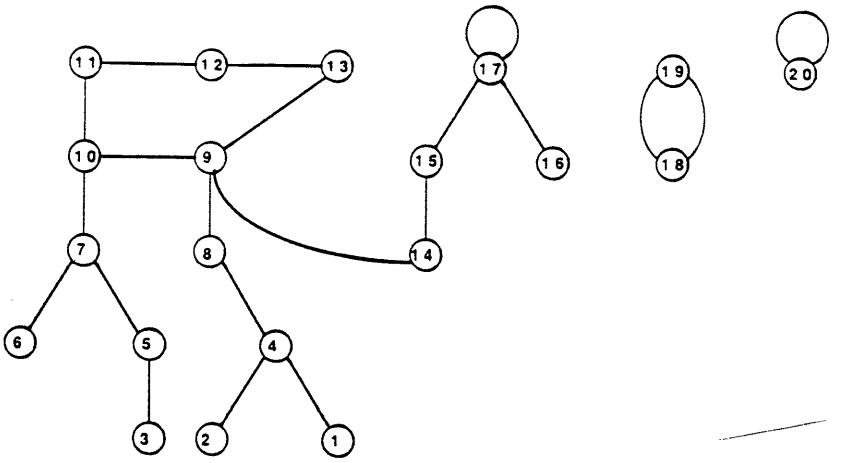
Basic–Equivalent–Path (BEP) for arc with incident nodes $(1,2)$:

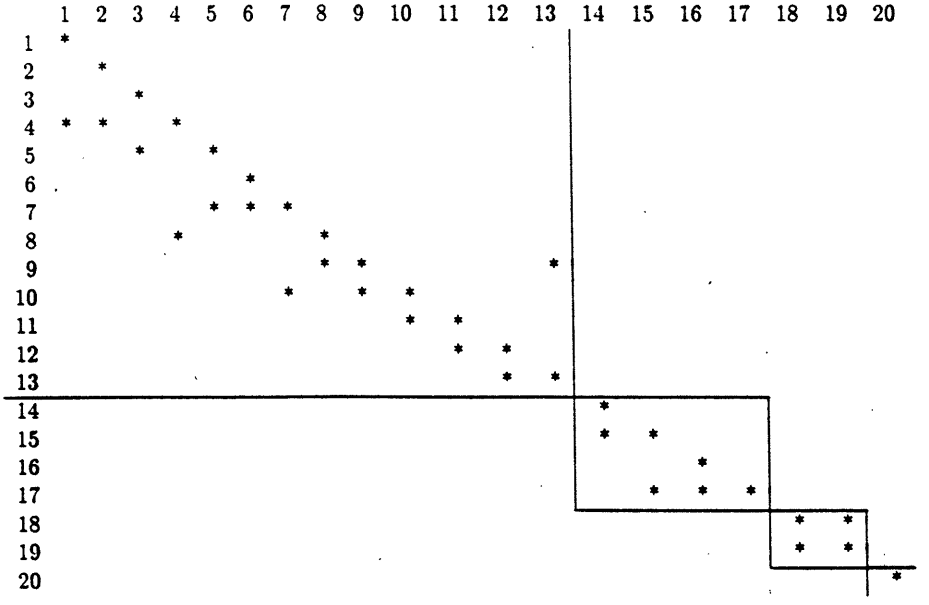
$$\{(2,6),(6,9),(9,10),(10,10),(10,8),(8,4),(4,1)\}.$$

Sparsity pattern of $(B^{-1}S)$ corresponding to superbasic $(1,2)$:

Row no.		Corresponding Basic Arc
1	*	(1,4)
2	*	(2,6)
3	0	
4	*	(4,8)
5	0	
6	*	(6,9)
7	0	
8	*	(8,10)
9	*	(9,10)
10	*	(10,10)

Figure 7: Pure network basis: matrix and graph representation, and an example of a basic equivalent path.





Basis of Generalized Network Problem

Basic-Equivalent-Path (BEP) for arc (9, 14):

$\{(9,10),(10,11),(11,12),(12,13),(13,9),(14,15),(15,17),(17,17)\}$.

Sparsity pattern of $B^{-1}S$ corresponding to superbasic arc (9, 14):

Row No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	0	0	0	0	0	0	0	0	*	*	*	*	*	*	*	0	*	0	0	0

Figure 8: Generalized network basis: matrix and graph representation, and an example of a basic equivalent graph.

alternative — and much simpler — techniques to partition the superbasic set of generalized network problems that take advantage of the block structure of the generalized network basis.

Previously, we observed that the graph associated with the basis of a generalized network problem is a collection of quasi–trees. Suppose the basis matrix B consists of submatrices B^ℓ, $\ell = 1, \ldots, L$. We denote the graph(quasi–tree) associated with B^ℓ by $G_\ell = (N_\ell, E_\ell)$. The superbasic set $\mathcal{S}$ can be partitioned in subsets $\mathcal{S}^k$ defined by

$$\mathcal{S}^\ell = \{(i,j) \in \mathcal{S} | i,j \in N_\ell\} \; \forall \; \ell = 1,2,\ldots,L \tag{71}$$

with $\cup_{\ell=1}^{L} \mathcal{S}^\ell \subseteq \mathcal{S}$. This partitioning scheme will ignore any superbasic variables that connect basis submatrices. A partitioning scheme that includes additional superbasic variables is the following: Given indices k, $p(k) \leq L$ and $q(k) \leq L$, $p(k) \neq q(k)$ choose $B^{p(k)}$ and $B^{q(k)}$ and define

$$\mathcal{S}^{p_k q_k} = \{(i,j) \in \mathcal{S} | i \in N_{p(k)}, j \in N_{q(k)}\} \tag{72}$$

$$\mathcal{S}^{p_k} = \{(i,j) \in \mathcal{S} | i,j \in N_{p(k)}\} \tag{73}$$

$$\mathcal{S}^{q_k} = \{(i,j) \in \mathcal{S} | i,j \in N_{q(k)}\} \tag{74}$$

and finally $\mathcal{S}^k = \mathcal{S}^{p_k q_k} \cup \mathcal{S}^{p_k} \cup \mathcal{S}^{q_k}$. To ensure that two set $\mathcal{S}^{k_1}, \mathcal{S}^{k_2}$ are independent we require

$$B^{p(k_1)} \neq B^{p(k_2)} \neq B^{q(k_2)}$$

$$B^{q(k_1)} \neq B^{p(k_2)} \neq B^{q(k_2)}.$$

A procedure, that was found to work well in practice, to identify these independent subsets $\mathcal{S}^k$ of superbasic arcs is described in [105]. Finally, we point out that the partitioning techniques described in this section can be extended to handle non-separable problems as explained in the same reference.

3.2.2 Dual and Proximal Methods

Dual coordinate methods are amenable to both synchronous and asynchronous parallel implementation. In general, the basic idea is to perform the

relaxation iterations for many nodes concurrently. On coarse-grain multiprocessors, each processor may be assigned to multiple nodes, whereas on extremely fine-grain machines, a cluster of processors might handle each node of the problem network.

The simplest synchronous approach involves the idea of a *coloring scheme* [22, pp. 21-27]; see also [129]. In a serial environment, one of the most natural implementations of dual relaxation is the *Gauss-Seidel* method, in which one lists the nodes of $\mathcal{N}$ in some fixed order, and cycles repeatedly through this list, relaxing each node in sequence. A *coloring* of the network graph $(\mathcal{N}, \mathcal{A})$ is some partition of the node set $\mathcal{N}$ into subsets (colors) such that no two nodes of the same color are adjacent in the network. Suppose now that one adopts a Gauss-Seidel node ordering in which all nodes of a given color are consecutive in the list.

From the form of (53), the maximizing value of p_i in each relaxation iteration depends only on the prices of the adjacent nodes p_j (for which there exists $(i,j) \in \mathcal{A}$ or $(j,i) \in \mathcal{A}$). and hence does not directly depend on the prices of any nodes having the same color as i. It follows that the price updates for all nodes of a given color may be performed simultaneously without altering the course of the algorithm. Such procedures work particularly well on transportation problems, for which only two colors suffice, one for the origin nodes, and one for the destination nodes. On more general networks, coloring is not guaranteed to be effective, but may still manage to employ a fairly large number of processors efficiently. For example, Zenios and Mulvey [129] present simulator-based results for up to 200 processors, using a greedy heuristic to color arbitrary problem networks. We also note that any network can be made bipartite by introducing an artificial node into the middle of each arc. Two colors will always suffice for the resulting expanded network.

The theory of synchronous dual methods other than those based on coloring schemes is subsumed in that of asynchronous methods, which we now summarize. We consider first a totally asynchronous environment in which there are no bounds on computational latency or communication delays. Each processor is associated with a single node i, and at time t stores the current price of i, $p_i(t)$, and also prices $p_j(i,t) = p_j(\tau_{ij}(t))$ for each neighbor j of i. These neighbor prices may be out of date, that is, $\tau_{ij}(t) \leq t$. As the algorithm progresses, the neighboring processors send messages carrying their prices to i, causing the $p_j(i,t)$ to be updated. As for the timing of the algorithm, one assumes only that for each time T and node i,

1. Node i is selected for relaxation at an infinite number of times after T.

2. All neighbors of i receive an infinite number of messages communicating the price $p_i(t)$ for some time $t > T$.

3. There exists some time $T' > T$ such that all messages carrying prices from before time T are no longer in transit at time T'.

Here, assumptions 1 and 2 intuitively say that processors never stop computing, nor do they ever cease successfully communicating their results to their neighbors. Assumption 3 says that outdated information is eventually purged from the communication system. The last two assumptions together imply $\tau_{ij}(t) \to \infty$ as $t \to \infty$. To obtain convergence results in this general framework, one must take the (non-restrictive) step of fixing one node price p_i in each connected component of $(\mathcal{N}, \mathcal{A})$. In the following, we assume that the problem network is connected, and set $p_1 = 0$. Under these conditions, it still does not follow that $\{p(t)\}$ converges to an optimal solution. Instead, one can assert only that every limit point p_i^∞ of each coordinate sequence $\{p_i(t)\}$ is such that there exists an optimal solution p^* of (55) with $p_1^* = 0$ and $p_i^\infty = p_i^*$. To obtain true convergence, one must assume

1. The set $P^* = \arg\max \{q(p) \mid p_1 = 0\}$ is bounded above in all coordinates.

2. When node i is relaxed, $p_i(t+1)$ is set to the *largest* value maximizing $q(p)$ with respect to the i^{th} coordinate.

3. $p(0) \geq p^*$ (componentwise) for all $p^* \in P^*$.

There is an analogous set of assumptions that gives convergence if P^* is bounded *below* in all coordinates, and $p_i(t+1)$ is set as *small* as possible when relaxing node i. For a complete analysis, see [19] or [22, Section 6.6].

We now consider *partially asynchronous* implementations, in which we assume a certain maximum time interval B between updates of each p_i, and that the prices of neighboring nodes used in each update are no more than B time units out of date, that is, $\tau_{ij}(t) \geq t - B$ for all i, j, and t. In this case, it is not necessary to fix the price of any node. Instead, convergence may be proven [22, Section 7.2] under the assumption that, when relaxing node i,

$$p_i(t+1) = (1-\gamma)p_i(t) + \gamma\tilde{p}_i \quad ,$$

where $\gamma \in (0,1)$ is fixed throughout the algorithm, and $\tilde{p}_i$ is, among all values maximizing $q(p)$ along the i^{th} coordinate, the *farthest* from $p_i(t)$. This result can be applied, for instance, to a synchronous Jacobi implementation in which all nodes simultaneously perform an update based on $p(t)$, and then exchange price information with their neighbors.

The main difficulty with all synchronous parallel dual methods is the complexity of the line search needed to maximize $q(p)$ along a given coordinate. Even when the f_{ij} have a simple functional form, $q(p)$ tends to have a large number of "breakpoints" along each coordinate, as the various arc flows $x_{ij}(p) = -\nabla q_{ij}(p_i - p_j)$ and $x_{ji}(p) = -\nabla q_{ji}(p_j - p_i)$ attain or leave their upper or lower bounds. If p_i must be moved across many such breakpoints, relaxing node i may be very time-consuming, possibly causing processors responsible for simultaneously relaxing other nodes be kept idle while waiting for the computation at i to be completed. To address this difficulty, Tseng [119] has proposed a line search that is itself parallelizable, although it may result in small steps; see also [22, pp. 413-414]. Several other stepsize procedures are proposed in [131].

For row-action methods in which inequality constraints are explicitly dualized, practical computational research has concentrated on the use of coloring schemes: once the nodes have been partitioned into colors, one can add one more color to handle the interval constraints [124]. More general "block iterative" parallel implementations have also been proposed; see [32] and the comprehensive survey in [29].

There are no parallelization ideas specific to proximal minimization methods; however, if dual coordinate methods are used to solve the sequence of strictly convex subproblems (60) generated by proximal minimization, any of the above parallelization approaches may be applicable.

3.2.3 Alternating Direction Methods

Alternating direction optimization methods are designed with massive, synchronous parallelism in mind. In the alternating step method, the primal update (63)-(64) for each arc (i,j) is completely independent of that for all other arcs; therefore, all such updates may be performed concurrently. Likewise, (66) may also be processed simultaneously for all arcs. The dual update (65) can be done concurrently for all nodes i. The simplicity of both the primal and dual calculations implies that they can be performed with

little or no synchronization penalty. In fact, the most challenging part of (63)-(66) to parallelize efficiently is the computation of the surpluses $g_i(y(t))$ and $g_i(x(t+1))$ It turns out that, of these, only the $g_i(x(t+1))$ requires much effort, as the $g_i(y(t))$ can be found quickly and in parallel via the identity

$$g_i(y(t+1)) = (1-\rho(t))g_i(y(t)) + \rho(t)g_i(x(t+1)).$$

This identity follows from (66) because $g_i(x)$ is an affine function of x [54, 51].

Implementating the more "aggregated" alternating direction method of [56] for nonlinear transportation problems is more complicated. For every origin node i, each iteration requires an (approximate) optimization over a simplex of dimension $d(i)$. All these optimizations are independent, and can be performed concurrently. Every destination node j requires a $d(j)$-element averaging operation at each iteration. Again, all these calculations can be performed at the same time.

3.3 Computational Experiences

3.3.1 Primal Methods

There have been substantial experiences with vector and parallel computing using the truncated Newton algorithm and its parallel block variant. Experiments with the vectorization of truncated Newton on a CRAY X-MP/48 are reported in [128], which used test problems derived from several applications: water distribution systems, matrix balancing and stick percolation. The performance of the vectorized algorithm was on average a factor of 5 faster than the scalar implementation. It is worth pointing out that a version of the program that was vectorized automatically by the compiler was only 15% faster than the scalar code. Substantial improvements in performance were achieved when appropriate data structures and a re-design of the implementation were developed.

A subsequent paper [130] reports on the performance of the parallel truncated Newton implementation on the CRAY X-MP/48 system. The algorithm achieved speedups of approximately 2.5 when executing on three processors. The observed speedup was very close to the upper bound provided by Amdhal's law, given that a fraction of the PTN algorithm was not parallelized.

The block-truncated Newton method was also tested empirically in [125]. The block-partitioning techniques have been tested on both pure and generalized network problems. For two sets of pure network test problems (*i.e.*, water distribution and stick percolation models) it was observed that the superbasic sets did not yield good partitionings. While the partitioning methods used are very efficient, poor partitioning resulted in insignificant improvements in performance. For the generalized network test problems, however, very good partitionings were obtained. In this case even a serial implementation of the block-partitioned algorithm was superior to the non-partitioned code. Improvements varied from a factor of 1.5 to 5.3 with average improvement across all problems of 2.6. As expected, better performance was observed for the larger test problems.

Parallel implementations were carried out on a 4 processor Alliant FX/4 and the CRAY X-MP/48. In both cases some modest speedups in performance were observed, in the range of 2 to 3. This is far from the linear speedup of 4. The discrepancy is due to a load balancing effect: the blocks of independent superbasic sets are not of equal size. Hence some processors need more time to complete their calculations than others. Overall, however, the parallel block-partitioned algorithm was shown to be much faster than the serial non-partitioned algorithm. The larger test problems have 15K nodes and 37K arcs, and the projected Newton equations in the neighborhood of the optimal solution are of dimension 22K $\times$ 22K. This problem required approximately 1 hour on a 4-processor Alliant FX/4 and 15 min. on the CRAY X-MP/48 using the block-partitioned algorithm. The (non-partitioned) algorithm executing on a single processor of the Alliant required 7 hours.

Figure 9 illustrates the performance of the PTN algorithm when implemented without the block-partitioning ideas on a VAX 8700 mainframe, then implemented with block-partitioning on the same (serial) architecture, and finally implemented with parallelization of the block-partitioned algorithm on the Alliant FX/4.

3.3.2 Dual and Proximal Methods

Computational experience with dual methods has been fairly extensive. Results for serial workstations have been reported by Tseng (see, for example, [124]). A group of quadratic-cost test problems, known as TSENG1 through

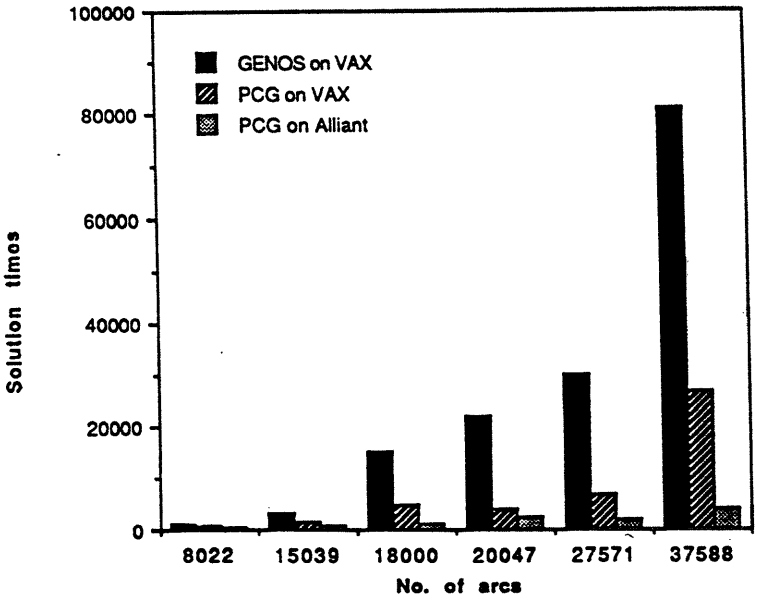


Figure 9: Performance of Primal Truncated Newton and block-partitioned Truncated Newton on serial and parallel architectures.

TSENG16, and especially TSENG1 through TSENG8, have become *de facto* standard test problems. These problems were generated using a version of NETGEN [81] altered to supply quadratic cost functions. TSENG1-TSENG8 are moderately ill-conditioned, positive definite, separable transportation problems ranging in size from 500×500 to 1250×1250. In TSENG1-TSENG4, the average node degree is about 10, whereas in TSENG5-TSENG8, the average node degree is about 20.

On parallel machines, early computational testing has focused primarily on the Alliant FX/8, a high-performance 8-processor, shared-memory system, and the CM–2 ([129] also gives some early, simulator-based results).

For the Alliant FX/8, [35] gives a lengthy account of the details of implementing dual relaxation, both synchronously using a coloring scheme, and with various degrees of asynchronism. The algorithm seems to work faster the more asynchronism is allowed; the fully asynchronous, 8-processor version requiring between 3 and 15 seconds to solve each of the problems TSENG1-TSENG8. The speedups over a one-processor implementation averaged around 6 (75% efficiency), but were as low as 3 for TSENG2, which appears to be a hard problem for dual relaxation. These runs were done at

an accuracy of 10^{-3}, meaning that the algorithm was terminated when the absolute value of the flow imbalance at all nodes was less than

$$\frac{10^{-3}}{n}\left(\sum_{i\in\mathcal{N}}|b_i|\right).$$

In other words,

$$\|r(x(p(t)))\|_\infty < (10^{-3})\,\|b\|_1/n.$$

Runs for the same Alliant FX/8 implementation at an accuracy of 10^{-6} given in [126] are longer by about an order of magnitude (and considerably more on the troublesome TSENG2). This phenomemon illustrates one drawback of dual coordinate methods — a "tailing" effect by which final convergence near the optimum may be slow. At an accuracy of 10^{-8}, which is more standard in mathematical programming, tailing effects would be even more pronounced.

Another architecture on which dual methods have been extensively tested is the Connection Machine, starting with [127] on the CM–1, and continuing with [89, 124, 131] on the CM–2. The work in [127] introduced the fundamental "segmented scan" representation of sparse networks for the Connection Machine architecture, but studied only specialized cost functions conducive to efficient line search, with encouraging preliminary results. The Connection Machine, having a SIMD architecture, cannot be truly asynchronous in a hardware sense, but [127] established that, under such an architecture, it seems best to iterate on all nodes simultaneously, using price information possibly outdated by one time unit, rather than to use a coloring scheme. To prove convergence of such a method, one must appeal to the partially asynchronous convergence analysis of [22, Section 7.2]. Still another architecture in which dual methods have been tested is a network of transputers [58].

A Connection Machine dual relaxation implementation for general separable positive definite quadratic cost functions is discussed in [131], which examines four different line searches: an iterative procedure based on the row-action literature, an exhaustive exact search method based on [73], a small-step procedure like that of Tseng [119], and an original, Newton-like step. Of these, the small-step and Newton-like procedures seemed to give the most consistently good results. Solution times for the 16K-processor CM–2 appeared to be comparable to those for the Alliant FX/8 (*e.g.*, using the Tseng-type line search, 12.4 seconds for TSENG8, versus 11.4 seconds

for asynchronous relaxation of the FX/8). With twice as many processors, CM–2 solution times decreased by about 30%.

Row-action dual methods designed specifically for transportation problems with quadratic or entropy ($f_{ij}(x_{ij}) = x_{ij}[\log(x_{ij}/a_{ij})-1]$) cost structures are described in [124]. The constraints are partitioned into three sets, the origin node flow balance equations, the destination node flow balance equations, and the arc flow bounds; a coloring scheme is used to alternate between the three sets. A simple grid data structure is used in place of the more general segmented scan representation of [127, 131]. At the fairly low accuracy level of 10^{-3}, a 32K-processor CM–2 gave very low run times, between 0.1 and 4.2 seconds, on TSENG1-TSENG8. Similar run times are given for test problems with an entropy cost structure, but at higher (10^{-6}) accuracy. For the quadratic case, [89] gives an even more efficient, microcoded implementation of the same algorithm.

A direct comparison of dual methods on the FX/8 and CM–2 appears in [126]. Here, the asynchronous FX/8 implementation of [35] is pitted against the row-action CM–2 method of [124] at 10^{-6} accuracy on the quadratic TSENG1-TSENG8 test set. On average, a 32K-processor Connnection Machine was about three times as fast as the (much cheaper) FX/8, but was actually significantly slower on two of the eight problems.

Comprehensive quantitative comparisons between the results of [35, 89, 124, 126, 127, 131] are difficult because of the varying precision levels, programming environments, hardware configurations, and test problems. Table 4 gives a partial summary for TSENG1-TSENG8. It is taken from [52], and therefore also includes data for an alternating direction method.

3.3.3 Alternating Direction Methods

Alternating direction methods have been tested less exhaustively than dual relaxation approaches, the main results appearing in [52]. This paper tests the alternating step method on quadratic and linear cost networks. The algorithm is implemented on the CM–2 using two data structures, one resembling that of [127], and one similar to the simple transportation grid approach of [124]. In agreement with preliminary results in [51, Chapter 7], performance on purely linear-cost problems proved to be very disappointing. On the other hand, the results compare favorably with the dual relaxation implementations of [131] on the purely quadratic TSENG1-TSENG8 problems, at

Problem	Run Time (Seconds) Dual Relaxation [131]	Row Action [124]	Alternating Direction [52]
TSENG1	6.77	1.55	2.11
TSENG2	N/A	2.45	4.32
TSENG3	9.75	2.03	3.50
TSENG4	13.92	2.06	4.68
TSENG5	3.34	1.63	2.95
TSENG6	5.56	3.43	3.91
TSENG7	15.90	3.45	5.17
TSENG8	9.68	2.85	4.64

Table 4: 16K-processor CM–2 Computational results for TSENG1-TSENG8, with with accuracy 10^{-3}, double precision arithmetic, and C/PARIS implementation. The dual relaxation method uses the Newton-like linesearch, which appeared to be the most efficient.

similar (low) levels of accuracy. Performance was not quite as good as the row-action approach of [124], which was specialized to transportation problems. Furthermore, the alternating step method was able to handle mixed linear-quadratic problems with as many as 20% linear-cost arcs without major degradation in performance, confirming the theoretical result that strict convexity is not necessary for its convergence. [52] also derives and tests a version of (63)-(66) for networks with gains, finding performance similar to that obtained for pure networks.

Alternating direction methods seem to exhibit a similar "tailing" behaviour to that of dual methods, in that convergence near the solution can sometimes be extremely slow. However, the phenomenon does not appear to operate identically in the two cases. For instance, the alternating step method has little difficulty with TSENG2, which produces a prolonged tailing effect under dual relaxation, but converges slowly near the optimum of TSENG1, which dual relaxation solves easily.

Recent results from [56] show that the amount of tailing in alternating direction methods depends strongly on the way the problem has been split. On

transportation problems similar to TSENG1-TSENG8, an alternating direction method slightly different from that [52] converged without discernable tailing effects to considerably higher accuracy (10^{-6}).

The combination of PMD (proximal minimization with D-functions) with row-action algorithms for the parallel solution of min-cost network flow problems has been tested by Nielsen and Zenios [99]. They use both quadratic and entropic nonlinear purturbations, and report numerical results for a set of test problems with up to 16 million arcs on the Connection Machine CM–2. In general they find that for the smaller test problems — up to 20,000 arcs — the GENOS [93] implementation of network simplex on a CRAY Y-MP is faster than the parallel code on a 16K CM–2. As problems get larger the difference in performance between the two codes is reduced. GENOS could not solve the extremely large test problems, with 0.5 to 16 million arcs. The parallel code solved these problems in times that range from 20 minutes to 1.5 hours. The same reference provides details on implementation of the PMD algorithm on the massively parallel machine, with particular discussion of termination criteria and the use of internal tactics that improve the performance of the algorithm. It also compares the quadratic with the entropic proximal algorithms. Experiences with the massively parallel implementation of PMD algorithms for problems with embedded network structures, *i.e.* two-stage and multi-stage stochastic network programs, are reported in [98, 100]

4 Conclusions

Research activities in parallel optimization can be traced back to the early days of linear programming: Dantzig-Wolfe decomposition can be viewed as a parallel optimization algorithm. However, it was not until the mid-eighties that parallel computer architectures became practical. This prompted the current research in parallel optimization. Several new algorithms have been developed for the parallel solution of network optimization problems: linear and nonlinear problems, assignment problems, transportation problems and problems with embedded network structures like the multicommodity network flow problem and the stochastic programming problem with network recourse.

On the theoretical front this research has produced a broad body of the-

ory on asynchronous algorithms. Furthermore, the insights obtained from looking into the parallel decompositions of network problems has occasionally produced algorithms that are very efficient even without the advantage of parallelism. In the domain of computational investigation, we have seen the design of general procedures and data structures that facilitate the parallel implementation of mathematical algorithms on a broad range of parallel architectures: from coarse-grain parallel vector computers, like the CRAY series of supercomputers, to massively parallel systems with thousands of processing elements, like the Connection Machines. The results is that problems with millions of variables can now be solved very efficiently.

The rapid progress towards the parallel solution of "building-block" algorithms for network problems has motivated research in more complex problem structures that arise in several areas of application. For example, multicommodity network flow problems that arise in military logistics applications are now solvable within minutes of computer time on a parallel supercomputer. A few years ago these applications required multi-day runs with state-of-the-art interior point algorithms [115, 106]. A new area of investigation, that has been prompted by development of parallel algorithms, deals with planning under uncertainty. The field of stochastic programming again goes back to the early days of linear programming [43]. However, the size of these optimization problems grows exponentially with the number of scenarios and time-periods. Decomposition algorithms involving network subproblems, and implemented on suitable parallel architectures, are now being used to solve problems with thousands of scenarios and millions of variables [92, 97].

References

[1] D. P. Ahlfeld, R. S. Dembo, J. M. Mulvey, and S. A. Zenios. Nonlinear programming on generalized networks. *ACM Transactions on Mathematical Software*, 13:350–368, 1987.

[2] G. Amdahl. The validity of single processor approach to achieving large scale computing capabilities. In *AFIPS Proceedings*, volume 30, pages 483–485, 1967.

[3] E. Balas, D. Miller, J. Pekny, and P. Toth. A parallel shortest path algorithm for the assignment problem. *Journal of the ACM*, 38:985–1004, October 1991.

[4] R. S. Barr, F. Glover, and D. Klingman. Enhancements of spanning tree labeling procedures for network optimization. *INFOR*, 17:16–34, 1979.

[5] R. S. Barr and B. L. Hickman. A new parallel network simplex algorithm and implementation for large time-critical problems. Technical report, Department of Computer Science and Engineering, Southern Methodist University, Dallas,Texas, August 1990.

[6] R. S. Barr and B. L. Hickman. Reporting computational experiments with parallel algorithms: Issues, measures and experts' opinion. *ORSA Journal on Computing*, 5(1):2–18, 1993.

[7] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16, 1958.

[8] D. P. Bertsekas. A distributed algorithm for the assignment problem. paper, Laboratory for Information and Decision Systems, MIT, Cambridge, MA, 1979.

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

[10] D. P. Bertsekas. Distributed asynchronous relaxation methods for linear network flow problems. Technical report LIDS-P-1606, Laboratory for Information and Decision Systems, MIT, Cambridge, MA, 1986.

[11] D. P. Bertsekas. Distributed relaxation methods for linear network flow problems. In *Proceedings 25th IEEE Conference on Decision and Control*, 1986.

[12] D. P. Bertsekas. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of Operations Research*, 14:105–123, 1988.

[13] D. P. Bertsekas. *Linear Network Optimization: Algorithms and Codes.* MIT Press, Cambridge, MA, 1991.

[14] D. P. Bertsekas and D. A. Castañon. Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Computing*, 17:707–732, 1991.

[15] D. P. Bertsekas and D. A. Castañon. Parallel asynchronous hungarian methods for the assignment problem. *ORSA Journal on Computing*, 5, 1993.

[16] D. P. Bertsekas and D. A. Castañon. Parallel primal-dual methods for the minimum cost network flow problem. *Computational Optimization and Applications*, 2:319–338, 1993.

[17] D. P. Bertsekas and J. Eckstein. Distributed asynchronous relaxation methods for linear network flow problems. In *International Federation of Automatic Control Congress*, 1987.

[18] D. P. Bertsekas and J. Eckstein. Dual coordinate step methods for linear network flow problems. *Mathematical Programming*, 42:203–243, 1988.

[19] D. P. Bertsekas and D. El Baz. Distributed asynchronous relaxation methods for the convex network flow problem. *SIAM Journal on Control and Optimization*, 25:74–85, 1987.

[20] D. P. Bertsekas, F. Guerriero, and R. Musmano. Parallel label correcting algorithms for shortest paths. Technical report, LIDS, M.I.T., Cambridge, MA, 1994.

[21] D. P. Bertsekas, P. Hossein, and P. Tseng. Relaxation methods for network flow problems with convex arc costs. *SIAM Journal on Control and Optimization*, 25:1219–1243, 1987.

[22] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[23] D. P. Bertsekas and J. N. Tsitsiklis. Some aspects of parallel and distributed iterative algorithms – a survey. *Automatica*, 27:3–21, 1991.

[24] L. M. Bregman. The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming. *USSR Computational Mathematics and Mathematical Physics*, 7:200–217, 1967.

[25] H. Brézis and P.-L. Lions. Produits infinis de resolvantes. *Israel Journal of Mathematics*, 29:329–345, 1978.

[26] D. A. Castañon. Development of advanced WTA algorithms for parallel processing. Technical report ONR N00014-88-C-0718, ALPHATECH, Inc., Burlington, MA, 1989.

[27] D. A. Castañon, B. Smith, and A. Wilson. Performance of parallel assignment algorithms on different multiprocessor architectures. Technical report TP-1245, ALPHATECH, Inc., Burlington, MA, 1989.

[28] Y. Censor. Row-action methods for huge and sparse systems and their applications. *SIAM Review*, 23:444–464, 1981.

[29] Y. Censor. Parallel application of block-iterative methods in medical imaging and radiation therapy. *Mathematical Programming*, 42:307–325, 1988.

[30] Y. Censor and A. Lent. An iterative row-action method for interval convex programming. *Journal of Optimization Theory and Applications*, 34:321–353, 1981.

[31] Y. Censor, A. R. D. Pierro, T. Elfving, G. Herman, and A. Iusem. On iterative methods for linearly constrained entropy maximization. In A. Wakulicz, editor, *Numerical Analysis and Mathematical Modelling, Vol. 24*, pages 145–163. Banach Center Publications, PWN - Polish Scientific Publisher, Warsaw, Poland, 1990.

[32] Y. Censor and J. Segman. On block-iterative entropy maximization. *Journal of Information and Optimization Sciences*, 8:275–291, 1987.

[33] Y. Censor and S. A. Zenios. On the use of D-functions in primal-dual methods and the proximal minimization algorithm. In A. Ioffe, M. Marcus, and S. Reich, editors, *Optimization and Nonlinear Analysis, Pitman Research Notes in Mathematics, Series 244*, pages 76–97. Longman, PWN - Polish Scientific Publisher, Warsaw, Poland, 1992.

[34] Y. Censor and S. A. Zenios. The proximal minimization algorithm with d-functions. *Journal of Optimization Theory and Applications*, 1992.

[35] E. Chajakis and S. Zenios. Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization. *Parallel Computing*, 17:873–894, 1991.

[36] M. D. Chang, M. Engquist, R. Finkel, and R. R. Meyer. A parallel algorithm for generalized networks. Technical report no. 642, Computer Science Department, University of Wisconsin, Madison, Wisconsin, 1987.

[37] R. Chen and R. Meyer. Parallel optimization for traffic assignment. *Mathematical Programming*, 42:327–345, 1988.

[38] R. Clark and R. Meyer. Multiprocessor algorithms for generalized network flows. Technical report # 739, Computer Science Department, The University of Wisconsin-Madison, Madison, WI, 1987.

[39] R. Clark and R. Meyer. Parallel arc allocation algorithms optimizing generalized networks. Technical report # 862, Computer Science Department, The University of Wisconsin-Madison, Madison, WI, July 1989.

[40] R. H. Clark, J. L. Kennington, R. R. Meyer, and M. Ramamurti. Generalized networks: Parallel algorithms and an empirical analysis. *ORSA Journal on Computing*, 4:132–145, 1992.

[41] R. W. Cottle, S. G. Duvall, and K. Zikan. A Lagrangean relaxation algorithm for the constrained matrix problem. *Naval Research Logistics Quarterly*, 33:55–76, 1986.

[42] C. W. Cryer. The solution of a quadratic programming using systematic overrelaxation. *SIAM Journal on Control*, 9:385–392, 1971.

[43] G. Dantzig. Planning under uncertainty using parallel computing. *Annals of Operations Research*, 14:1–16, 1988.

[44] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, 1963.

[45] R. S. Dembo. A primal truncated newton algorithm for large-scale nonlinear network optimization. *Mathematical Programming Study 31*, pages 43–72, 1987.

[46] R. S. Dembo and J. G. Klincewicz. Dealing with degeneracy in reduced gradient algorithms. *Mathematical Programming*, 31(3):357–363, March 1985.

[47] R. S. Dembo and T. Steihaug. Truncated–newton algorithms for large-scale unconstrained optimization. *Mathematical Programming*, 26:190–212, 1983.

[48] G. R. Desrochers. *Principles of Parallel and Multi-Processing.* McGraw Hill, New York, NY, 1987.

[49] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[50] E. Dijkstra. A note on two problems in connection with graphs. *Numeriche Mathematics*, 1, 1959.

[51] J. Eckstein. *Splitting Methods for Monotone Operators, with Applications to Parallel Optimization.* PhD thesis, Massachusetts Institute of Techonology, Cambridge, MA, 1989. Report LIDS-TH-1877, Laboratory for Information and Decision Systems, M.I.T.

[52] J. Eckstein. Implementing and running the alternating step method on the Connection Machine CM-2. *ORSA Journal on Computing*, 5(1):84–96, 1993.

[53] J. Eckstein. Nonlinear proximal point algorithms using Bregman functions, with applications to convex programming. *Mathematics of Operations Research*, 18(1):202–226, 1993.

[54] J. Eckstein. Parallel alternating direction multiplier decomposition of convex programs. *Journal of Optimization Theory and Applications*, 80(1):39–62, 1994.

[55] J. Eckstein and D. P. Bertsekas. On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55(3):293–312, 1992.

[56] J. Eckstein and M. Fukushima. Some reformulations and applications of the alternating direction method of multipliers. In W. W. Hager, D. W. Hearn, and P. M. Pardalos, editors, *Large-Scale Optimization: State of the Art*, pages 119–138. Kluwer Scientific, 1994.

[57] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the Association of Computing Machinery*, 19:248–264, 1972.

[58] D. ElBaz. A computational experience with distributed asynchronous iterative methods for convex network flow problems. In *Proceedings 28th IEEE Conference on Decision and Control*, Tampa, Florida, Dec. 1989.

[59] L. F. Escudero. Performance evaluation of independent superbasic sets on nonlinear replicated networks. *European Journal of Operational Research*, 23:343–355, 1986.

[60] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C–21(9):948–960, 1972.

[61] L. R. Ford and D. R. Fulkerson. A primal-dual algorithm for the capacitated hitchcock problem. *Naval Research Logistics Quarterly*, 4:47–54, 1957.

[62] M. Fortin and R. Glowinski. On decomposition-coordination methods using an augmented Lagrangian. In M. Fortin and R. Glowinski, editors, *Augmented Lagrangian methods: Applications to the Solution of Boundary Value Problems*. North-Holland, Amsterdam, 1983.

[63] D. Gabay. Applications of the method of multipliers to variational inequalities. In M. Fortin and R. Glowinski, editors, *Augmented Lagrangian methods: Applications to the Solution of Boundary Value Problems*. North-Holland, 1983.

[64] D. Gabay and B. Mercier. A dual algorithm for the solution of nonlinear variational inequalities via finite element approximations. *Computational Mathematics and Applications*, 2:17–48, 1976.

[65] D. Gibby, D. Glover, D. Klingman, and M. Mead. A comparison of pivot selection rules for primal simplex based network codes. *Operations Research Letters*, 2, 1983.

[66] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, London, 1981.

[67] F. Glover, J. Hultz, D. Klingman, and J. Stutz. Generalized networks: A fundamental computer-based planning tool. *Management Science*, 24(12):1209–1220, 1978.

[68] F. Glover, D. Klingman, and J. Stutz. Extensions of the augmented predecessor index method to generalized network problems. *Transportation Science*, 7:377–384, 1973.

[69] R. Glowinski and A. Marroco. Sur l'approximation, par elements d'ordre un, et la resolution, par penalisation-dualité, d'une classe de problemes de Dirichlet non lineares. *Revue Française d'Automatique, Informatique, et Recherche Operationelle*, 9(R-2):41–76, 1975.

[70] A. V. Goldberg. Efficient graph algorithms for sequential and parallel computers. Report TR-374, Laboratory for Computer Science, MIT, Cambridge, MA, 1987.

[71] A. V. Goldberg and R. E. Tarjan. Solving minimum cost flow problems by successive approximation. *Mathematics of Operations Research*, 15:430–466, 1990.

[72] M. D. Grigoriadis. An efficient implementation of the network simplex method. *Mathematical Programming Study*, 26, 1986.

[73] R. Helgason, J. Kennington, and H. Lall. A polynomially bounded algorithm for singly constrained quadratic programs. *Mathematical Programming*, 18:338–343, 1980.

[74] C. Hildreth. A quadratic programming procedure. *Naval Research Logistics Quarterly*, 4:79–85, 1957. Erratum, p. 361.

[75] W. D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, Massachusetts, 1985.

[76] C. A. R. Hoare. Monitors: An operating systems scheduling concept. *Communications of the Association for Computing Machinery*, 17:549–557, 1974.

[77] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[78] D. Kempa, J. Kennington, and H. Zaki. Performance characteristics of the Jacobi and Gauus-Seidel versions of the auction algorithm on the Alliant FX/8. Technical report OR-89–008, Department of Mechanical and Industrial Engineering, University of Illinois, Champaign-Urbana, 1989.

[79] J. Kennington and Z. Wang. Solving dense assignment problems on a shared memory multiprocessor. Report 88-OR-16, Dept. of Operations Research and Applied Science, Southern Methodist University, Dallas, TX, 1988.

[80] J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming.* John Wiley, N. York, 1980.

[81] D. Klingman, A. Napier, and J. Stutz. NETGEN - a program for generating large-scale (un)capacitated assignment, transportation, and minimum cost flow network problems. *Management Science*, 20:814–822, 1974.

[82] M. Lescrenier and P. L. Toint. Large scale nonlinear optimization on the FPS164 and CRAY X-MP vector processors. *The International Journal of Supercomputer Applications*, 2:66–81, 1988.

[83] X. Li and S. A. Zenios. Data-level parallel solution of min-cost network flow problems using ϵ-relaxations. *European Journal of Operations Research*, 1994.

[84] P.-L. Lions and B. Mercier. Splitting methods for the sum of two nonlinear operators. *SIAM Journal on Numerical Analysis*, 16:964–979, 1979.

[85] O. L. Mangasarian and R. R. Meyer, editors. Parallel Optimization. *Mathematical Programming*, volume 42(2), 1988.

[86] O. L. Mangasarian and R. R. Meyer, editors. Parallel Optimization II. *SIAM Journal on Optimization*, volume 1(4), 1991.

[87] B. Martinet. Regularisation d'inequations variationelles par approximations successives. *Revue Française d'Informatique et Recherche Operationelle*, 4:154–159, 1970.

[88] M. McKenna and S. Zenios. An optimal parallel implementation of a quadratic transportation algorithm. In *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–363, Philadelphia, PA, 1990. SIAM.

[89] M. McKenna and S. A. Zenios. An optimal parallel implementation of a quadratic transportation algorithm. Technical report, Thinking Machines Corporation, Cambridge, MA, 1990.

[90] R. R. Meyer and S. A. Zenios, editors. *Parallel Optimization on Novel Computer Architectures*, volume 14 of *Annals of Operations Research*. A.C. Baltzer Scientific Publishing Co., Switzerland, 1988.

[91] D. Miller, J. Pekny, and G. L. Thompson. Solution of large dense transportation problems using a parallel primal algorithm. *Operations Research Letters*, 9(5):319–324, 1990.

[92] J. Mulvey and H. Vladimirou. Evaluation of a parallel hedging algorithm for stochastic network programming. In R. Sharda, B. Golden, E. Wasil, O. Balci, and W. Stewart, editors, *Impact of Recent Computer Advances on Operations Research*, 1989.

[93] J. Mulvey and S. Zenios. GENOS 1.0: A Generalized Network Optimization System. User's Guide. Report 87–12–03, Decision Sciences Department, the Wharton School, University of Pennsylvania, Philadelphia, PA 19104, 1987.

[94] J. M. Mulvey. Pivot strategies for primal-simplex network codes. *Journal of the Association of Computing Machinery*, 25(2), April 1978.

[95] B. Murtagh and M. Saunders. Large-scale linearly constrained optimization. *Mathematical Programming*, 14:41–72, 1978.

[96] B. Narendran, R. DeLeone, and P. Tiwari. An implementation of the ϵ-relaxation algorithm on the CM-5. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 1993.

[97] S. Nielsen and S. Zenios. Massively parallel algorithms for nonlinear stochastic network problems. Report 90–09–08, Decision Sciences Department, The Wharton School, University of Pennsylvania, Philadelphia, PA, 1990.

[98] S. Nielsen and S. Zenios. Solving multistage stochastic network programs. Report 92–08–04, Decision Sciences Department, The Wharton School, University of Pennsylvania, Philadelphia, PA, 1992.

[99] S. Nielsen and S. Zenios. Proximal minimizations with D-functions and the massively parallel solution of linear network programs. *Computational Optimization and Applications*, 1(4):375–398, 1993.

[100] S. Nielsen and S. Zenios. Proximal minimizations with D-functions and the massively parallel solution of linear stochastic network programs. *International Journal of Supercomputing Appplications*, 1994.

[101] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, New York, 1970.

[102] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[103] J. Peters. The network simplex method on a multiprocessor. *Networks*, 20, 1990.

[104] C. Pfefferkorn and J. Tomlin. Design of a linear programming system for the Illiac IV. Technical report, Department of Operations Research, Stanford University, Stanford, California, April 1976.

[105] C. Phillips and S. Zenios. Experiences with large scale network optimization on the Connection Machine. In *The Impact of Recent Computing Advances on Operations Research*, volume 9, pages 169–180. Elsevier Science Publishing, 1989.

[106] M. Pinar and S. Zenios. Parallel decomposition of multicommodity network flows using linear-quadratic penalty functions. *ORSA Journal on Computing*, 4(3):235–249, 1992.

[107] L. C. Polymenakos and D. P. Bertsekas. Parallel shortest path auction algorithms. Report LIDS-P-2151, LIDS, MIT, Cambridge, MA, 1993.

[108] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Communications of the Association for Computing Machinery*, 29, 1986.

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

[110] R. T. Rockafellar. Augmented Lagrangians and applications of the proximal point algorithm in convex programming. *Mathematics of Operations Research*, 1:97–116, 1976.

[111] R. T. Rockafellar. Monotone operators and the proximal point algorithm. *SIAM Journal on Control and Optimization*, 14:877–898, 1976.

[112] R. T. Rockafellar. *Network Flows and Monotropic Optimization.* John Wiley, New York, 1984.

[113] J. Rosen, editor. *Supercomputing in Large Scale Optimization.* Annals of Operations Research. A.C. Baltzer Scientific Publishing Co., Switzerland, 1990.

[114] S. A. Savari and D. P. Bertsekas. Finite termination of asynchronous iterative algorithms. LIDS report, MIT, Cambridge, 1994.

[115] G. Schultz and R. Meyer. A structured interior point method. *SIAM Journal on Optimization*, 1(4):583–602, November 1991.

[116] Sequent Computer Systems. *Symmetry^{TM} Technical Summary*, 1987.

[117] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[118] M. Teboulle. Entropic proximal mappings with applications to nonlinear programming. Working paper, Department of Mathematics and Statistics, University of Maryland, 1990.

[119] P. Tseng. Dual ascent methods for problems with strictly convex costs and linear constraints: a unified approach. *SIAM Journal on Control and Optimization*, 28:214–242, 1990.

[120] P. Tseng and D. P. Bertsekas. On the convergence of the exponential multiplier method for convex programming. *Mathematical Programming*, 60:1–19, 1993.

[121] J. Wein and S. Zenios. Massively parallel auction algorithms for the assignment problem. In *3rd Symposium on the Frontiers of Massively Parallel Computations*, pages 90–99, 1990.

[122] J. Wein and S. Zenios. On the massively parallel solution of the assignment problem. *Journal of Parallel and Distributed Computing*, 13:221–236, 1991.

[123] H. Zaki. A comparison of two algorithms for the assignment problem. Technical report ORL 90-002, Dept. of Mech. and Ind. Eng., Univ. of Illinois, Champaign-Urbana, IL, 1990.

[124] S. Zenios and Y. Censor. Massively parallel row-action algorithms for some nonlinear transportation problems. *SIAM Journal on Optimization*, 1:373–400, 1991.

[125] S. Zenios and M. Pinar. Parallel block-partitioning of truncated newton for nonlinear network optimization. *SIAM Journal on Scientific and Statistical Computing*, 13(5):1173–1193, 1992.

[126] S. Zenios, R. Qi, and E. Chajakis. A comparative study of parallel dual coordinate ascent implementations for nonlinear network optimization. In T. Coleman and Y. Yi, editors, *Large Scale Numerical Optimization*, pages 238–255. SIAM, 1990.

[127] S. A. Zenios and R. A. Lasken. Nonlinear network optimization on a massively parallel Connection Machine. *Annals of Operations Research*, 14:147–163, 1988.

[128] S. A. Zenios and J. M. Mulvey. Nonlinear network programming on vector supercomputers: a study on the CRAY X-MP. *Operations Research*, 34:667–682, Sept.-Oct. 1986.

[129] S. A. Zenios and J. M. Mulvey. A distributed algorithm for convex network optimization problems. *Parallel Computing*, 6:43–56, 1988.

[130] S. A. Zenios and J. M. Mulvey. Vectorization and multitasking of nonlinear network programming algorithms. *Mathematical Programming*, 42:449–470, 1988.

[131] S. A. Zenios and S. Nielsen. Massively parallel algorithms for singly constrained nonlinear programs. Report 90-03-01, Decision Sciences Department, Wharton School, University of Pennsylvania, Philadelphia, PA, 1990.