Parallel Computing 17 (1991) 873–894
North-Holland

Practical aspects and experiences

Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization

Emmanuel D. Chajakis and Stavros A. Zenios

Decision Sciences Department, The Wharton School, University of Pennsylvania, Philadelphia, PA 19104, USA

Received December 1989
Revised May 1990, October 1990, January 1991

Abstract

Chajakis, E.D. and S.A. Zenios, Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization, Parallel Computing 17 (1991) 873–894.

We discuss a parallel dual relaxation algorithm for network optimization. Synchronous and asynchronous implementations of the algorithm are developed on a shared memory multiprocessor, the Alliant FX/8. Alternative designs for parallel computing that tradeoff synchronization delays with computations are proposed. Their performance is analyzed empirically with computational experiments. It is demonstrated that the most effective parallel implementation is an asynchronous one, for which speedup factors of up to 7.2 are achieved on an 8 processor system.

Keywords. Nonlinear network optimization; relaxation algorithms; parallel implementation; partitioning; scheduling; synchronization; shared memory multiprocessor.

1. Introduction

Optimization problems with network constraints and nonlinear costs are used to model several kinds of physical systems. Financial models, hydroelectric power generation, air-traffic control and pipe (e.g. water distribution) network systems are a few examples of real world systems that are modelled as nonlinear network-flow optimization problems. A recent review of network programming applications is given by Dembo et al. [9] with emphasis on the nonlinear models.

The wide range of nonlinear network applications has motivated the development of algorithms and software that can solve progressively larger problems taking advantage of the problem's intrinsic structure. Recent advances in computer technology and new computer architectures become increasingly strong factors influencing the design of algorithms for network-flow problems. The advent of parallel computing prompted the development of parallel algorithms. Such algorithms, when implemented on suitable hardware, can solve network problems with tens of thousands of nodes and hundreds of thousands of arcs in a fraction of the time it takes for the same algorithm on serial machines. Bertsekas and Tsitsiklis [5] provide a textbook treatment of parallel computations including network-flow problems. Zenios [14] provides an overview and an annotated bibliography of the current status of parallel optimization.

Nonlinear network optimization problems with strictly convex and separable objective functions give rise to unconstrained and differentiable dual problems. Such problems are in turn well suited for parallel computation using Jacobi-type algorithms. Dual coordinate ascent algorithms for this class of problems were developed in Bertsekas et al. [6]. It was shown later, Tseng et al. [13], that these algorithms would converge even in an asynchronous implementation (under some rather technical but easy to enforce conditions). Zenios and Mulvey [16] showed how the algorithms could be implemented in parallel without the need to resort to an asynchronous implementation. They provided the first empirical evidence – using a simulated multiprocessing environment – that these algorithms would realize significant speedups. Later, Zenios and Lasken [15] developed data structures for the implementation of the algorithms on a massively parallel Connection Machine CM-1. The SIMD architecture of this machine imposed a Jacobi implementation upon the authors.

In this report we compare directly synchronous and asynchronous implementations of this class of algorithms. Using a shared memory multiprocessor – the Alliant FX/8 – we have the flexibility to implement both synchronous and asynchronous relaxation methods (abbreviated SRM and ARM, respectively). The parallel implementations are evaluated based on computational experiments with the test problems of Tseng [12]. Standard problems of parallel implementation such as partitioning, scheduling and synchronization are addressed, and some solutions are proposed and subsequently implemented. This paper is the first study of relaxation algorithms that compares synchronous with asynchronous implementations. It develops implementation techniques that are applicable to both shared memory and distributed memory MIMD systems.

The remainder of this paper is organized as follows: Section 2 presents the problem and the relaxation algorithm. Both synchronous and asynchronous implementations together with the related partitioning, scheduling and synchronization issues are discussed in Section 3. Section 4 provides details on the computational results. Conclusions are contained in Section 5.

2. Problem definition and the relaxation algorithm

We present in this section the nonlinear network-flow cost minimization problem and the relaxation algorithm. Interested readers may refer to [5, pp. 390–414] and the references therein for a textbook treatment of related material.

Let us define as $G \equiv \{N, E\}$ a directed network where $N = \{i \mid i = 1, \ldots, m\}$ is the set of nodes and $E = \{(i, j) \mid i, j \in N\}$ is the set of arcs or edges. The primal nonlinear network-flow cost minimization problem is defined as follows:

$$\begin{aligned} \underset{x}{\text{Minimize}} \quad & \sum_{(i,j)\in E} f_{ij}(x_{ij}), \\ \text{subject to} \quad & Ax = b, \\ & l \leqslant x \leqslant u, \end{aligned}$$

where

f_{ij}: $\mathbb{R} \to \mathbb{R}$ strictly convex cost function of one variable, assumed here to be differentiable,
$x = (x_{ij}) \in \mathbb{R}^n$, $(i, j) \in E$, vector of flows on the arcs of the network,

A, $m \times n$ node-arc incidence matrix with at most two nonzero entries in each column,
$b = (b_i) \in \mathbb{R}^m$, $i \in N$, vector of supplies and demands at nodes of the network,
$l = (l_{ij}) \in \mathbb{R}^n$, $(i, j) \in E$, lower bounds for flows on arcs,
$u = (u_{ij}) \in \mathbb{R}^n$, $(i, j) \in E$, upper bounds for flows on arcs,
$\pi = (\pi_i) \in \mathbb{R}^m$, $i \in N$, vector of dual prices.

We also define the following for later use:

$$\delta^+(i) = \{(i, j) \mid \forall (i, j) \in E\}, \text{ the set of arcs leaving node } i,$$

$$\delta^-(j) = \{(i, j) \mid \forall (i, j) \in E\}, \text{ the set of arcs entering node } j,$$

$$d_i = \sum_{(i, j) \in \delta^+(i)} x_{ij} - \sum_{(j, i) \in \delta^-(i)} x_{ji} - b_i, \text{ the } \textit{deficit} \text{ at node } i.$$

In order to specify the algorithm we also define complementary slackness conditions that are satisfied by an optimal primal-dual set (x, π):

$$\pi_i - \pi_j \geqslant \left. \frac{\partial f_{ij}(x_{ij})}{\partial x_{ij}} \right|_{x_{ij} = u_{ij}} \Rightarrow x_{ij} = u_{ij}, \tag{2.1}$$

$$\pi_i - \pi_j \leqslant \left. \frac{\partial f_{ij}(x_{ij})}{\partial x_{ij}} \right|_{x_{ij} = l_{ij}} \Rightarrow x_{ij} = l_{ij}, \tag{2.2}$$

$$\pi_i - \pi_j = \left. \frac{\partial f_{ij}(x_{ij})}{\partial x_{ij}} \right|_{x_{ij} = \hat{x}_{ij}}, \quad \text{for } l_{ij} < \hat{x}_{ij} < u_{ij} \Rightarrow x_{ij} = \hat{x}_{ij}. \tag{2.3}$$

2.1. *The relaxation algorithm*

The algorithm can now be stated as follows:

Step 0: (Initialization) Set $k \leftarrow 0$. Give initial dual prices π_i^0 to all nodes $i \in N$. Compute initial flow values x^0 to satisfy complementary slackness conditions (2.1)–(2.3), and initialize deficits d_i^0 for all $i \in N$.

Step 1: (Node selection) Search for a node s such that $|d_s^k| > \epsilon$ where ϵ is a user specified tolerance. If there is no such node than decrease ϵ or STOP if ϵ is sufficiently small.

Step 2: (Price update) Update the dual price $\pi_s^{k+1} = \pi_s^k + \gamma$. The constant γ is chosen such that $|d_s^{k+1}| < \epsilon$. The deficit d_s^{k+1} is computed using the vector of flows x^{k+1} that satisfies the complementary slackness conditions (2.1)–(2.3), using the updated π_s^{k+1} for this calculation. Details of a simple procedure for computing γ are given below.

The tolerance ϵ at Step 1 is reduced iteratively until it reaches a user specified final value. Each sweep over all nodes is called a *minor* iteration. Each set of minor iterations during which there is at least one node s with $|d_s| > \epsilon$ constitutes one *major* iteration. Each new major iteration begins with a new reduced value for ϵ. It is not essential that the deficit is reduced exactly to zero at Step 2 for the algorithm to converge. However, sufficient decrease in deficit must be realized. The one-dimensional search procedure that is used to compute γ in Step 2 is explained next. It is adapted from Tseng [9] for our problems and it produces an acceptable dual price to guarantee convergence; alternative methods for estimating the parameter γ which are also suited for parallel computing are given in Nielsen and Zenios [18]. Other details on the implementation of this algorithm can be found in Zenios and Mulvey [16].

2.2. Price update

The price π_s^{k+1} at node s for the $(k+1)$st iteration is

$$\pi_s^{k+1} = \pi_s^k + \gamma,$$

where γ is calculated as follows:

Case 1 ($d_s^k > 0$). Let $\mu \in (0,1)$ be a user specified parameter. Let n_s be the number of outgoing edges at nodes s that are not at the lower bound and incoming edges at node s that are not at the upper bound.

1a. For all outgoing arcs $(s, j) \in \delta_s^+$. At the $(k+1)$st iteration the flow on (s, j) must satisfy

$$x_{sj}^k - \frac{d_s^k}{n_s} \leqslant x_{sj}^{k+1} \leqslant x_{sj}^k - \mu \frac{d_s^k}{n_s}.$$

Let $\alpha = x_{sj}^k - d_s^k/n_s$ and $\beta = x_{sj}^k - \mu d_s^k/n_s$.
Then:

$$\gamma_{sj} = \begin{cases} -\infty, & \text{if} \quad \beta < l_{sj}, \\ \theta_{sj}(l_{sj}) & \text{if} \quad \alpha < l_{sj} \leqslant \beta, \\ \theta_{sj}(\alpha) & \text{if} \quad l_{sj} < \alpha. \end{cases}$$

where

$$\theta_{sj}(w) = -\pi_s^k + \pi_j^k + \left. \frac{\partial f_{sj}(x_{sj})}{\partial x_{sj}} \right|_{x_{sj}=w}.$$

Set $\gamma_{\text{out}} = \max_{(s,\, j) \in \delta^+(s)} \{\gamma_{sj}\}$.

1b. For all incoming arcs $(t, s) \in \delta_s^-$. At the $(k+1)$st iteration the flow on (t, s) must satisfy

$$x_{ts}^k + \mu \frac{d_s^k}{n_s} \leqslant x_{ts}^{k+1} \leqslant x_{ts}^k + \frac{d_s^k}{n_s}.$$

Let $\alpha = x_{ts}^k + \mu d_s^k/n_s$ and $\beta = x_{ts}^k + d_s^k/n_s$.
Then:

$$\gamma_{ts} = \begin{cases} -\infty & \text{if} \quad u_{ts} < \alpha, \\ \theta_{ts}(u_{ts}) & \text{if} \quad \alpha < u_{ts} \leqslant \beta, \\ \theta_{ts}(\beta) & \text{if} \quad \beta < u_{ts}. \end{cases}$$

where

$$\theta_{ts}(w) = \pi_t^k - \pi_s^k - \left. \frac{\partial f_{ts}(x_{ts})}{\partial x_{ts}} \right|_{x_{ts}=w}.$$

Set $\gamma_{\text{in}} = \max_{(t,\, s) \in \delta^-(s)} \{\gamma_{ts}\}$.

Finally $\gamma = \max\{\gamma_{\text{out}}, \gamma_{\text{in}}\}$.

Case 2 ($d_s^k < 0$). This case is treated similarly to Case 1, and details are ommitted.

It can be shown, see Bertsekas and Tsitsiklis ([5], pp. 390–406), that with the choice of π_s^{k+1} given by this linesearch the deficit at node s will be reduced to $0 \leqslant d_s^{k+1} \leqslant (1 - \mu/n_s) d_s^k$. This in turn implies that the deficit reduction could be small for very dense problems. However the price updating procedure is very simple and worked well for our test problems. Its use is illustrated in *Fig. 1*.

3. Parallel relaxation methods

From the description of the algorithm in the previous section it is apparent that *one node at a time* is chosen and its price is adjusted by a scalar γ. New flows and associated deficits are computed from the complementary slackness conditions based on the prices of the adjacent nodes. Based on this observation a parallel synchronous relaxation method (SRM) can be proposed: prices of nodes that are not adjacent to each other can be computed simultaneously. Accordingly, an asynchronous relaxation method (ARM) can be proposed if the restriction of non-adjacency of nodes that are proccessed in parallel is dropped. SRM has identical convergence properties as the sequential algorithm. The convergence of ARM under different conditions in the amount of asynchronicity has been established in Bertsekas and El Baz [3] and Tseng et al. [13].

We discuss in this Section the parallel implementation of both synchronous and asynchronous implementations. We address issues of task partitioning, task scheduling and synchronization for both SRM and ARM. All implementations discussed below store price and flow vectors in shared memory and also maintain a vector of deficit values. The algorithm can be implemented using only price information in memory, and this implementation would be closer to the theoretical framework of [3,5]. However, this implementation would require additional calculations in obtaining flow and deficit vectors for a given price vector at each iteration. Using additional memory (i.e. flow and deficit vectors) offers substantial savings in computing time. Of course maintaining data integrity and consistency among the three arrays is a problem that is handled in a different way in each of the parallel implementations.

3.1. Task partitioning

This is the problem of partitioning the algorithm into modules of computation that can be scheduled for concurrent execution on multiple processors. Different partitioning is required for SRM and ARM.

3.1.1. Synchronous relaxation

Each problem must be partitioned with respect to its nodes into sets, each of which contains nodes that are not connected to each other by an arc. In other words, disjoint subsets of nodes must be found. This observation was first made in [16]. The partitioning problem can be formalized as follows:

For a given graph $G \equiv \{N, E\}$ find a set of subsets $\{N_i\}$, $i = 1, \ldots, K$ such that:

$$N = \bigcup_{i=1}^{K} N_i, \quad N_i \bigcap N_j = \emptyset \quad \text{for} \quad i \neq j$$

and $\forall k, l \in N_i$ for all $i \nexists (k, l) \in E$.

The problem just defined is that of coloring the graph G and each of the K disjoint sets is a set of nodes with the same color. The minimal K is the *chromatic number* of the network. A simple heuristic algorithm is employed for the partitioning of the graph into subgraphs that consist of nodes with the same color. The algorithm is a modification of a heuristic given in

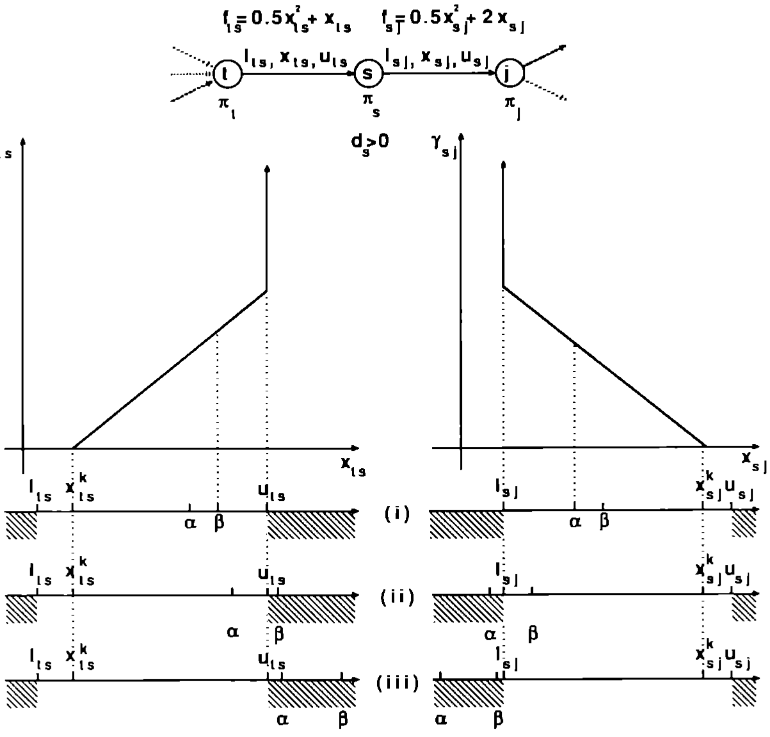

Fig. 1. Illustration of the linesearch procedure for a simple example.
The linesearch procedure is illustrated here for a simple example. The example corresponds to Case 1 ($d_s^k > 0$) of the description in section 2.2. Since $d_s > 0$, the dual price π_s must decrease. The dual price decrements γ_{ls} and γ_{sj} are set: (i) at the extreme of the interval $[\alpha, \beta]$ which is the farthest from the current flow, if both α and β are within the bounds, (ii) at the appropriate bound, if only one of α and β is within the bounds. (iii) They are set to $-\infty$ if none of α and β is within the bounds. The new dual price is then computed as: $\pi_s^{k+1} = \pi_s^k + \max(\gamma_{ls}, \gamma_{sj})$.

Christofides [8]. It is important to note that finding the partitioning that corresponds to the chromatic number of a network is an NP-complete problem. In our implementation it is not important to have the minimal number of disjoint sets of nodes. Therefore it is not advisable to use an algorithm for the minimum coloring.

Upon termination of the algorithm the nodes with the same color are partitioned into P blocks of equal size (where P is usually equal to the number of processors on the parallel computer). Each block becomes a task scheduled for concurrent execution. *Figure 2* shows the block partitioning of a small network that was colored with two colors by our heuristic algorithm. Each of the two groups of nodes with the same color is partitioned into four blocks for concurrent execution by multiple processors. Nodes that belong to a block are shown in black.

3.1.2. Asynchronous relaxation

In the ARM it is possible to iterate on the prices of adjacent nodes concurrently. This implies, however, that the prices on which a calculation is based are changing while the calculation is performed. The convergence of the algorithm, see Bertsekas and El Baz [3] and Tseng et al. [13], is preserved under some mild conditions. Hence, in implementing ARM we do not necessarily have to partition the set of nodes into subsets of non-adjacent nodes. However, the partitioning (i.e. graph coloring) scheme introduced for SRM can be achieved very efficiently. We therefore partition the network into sets of nodes with the same color. Each

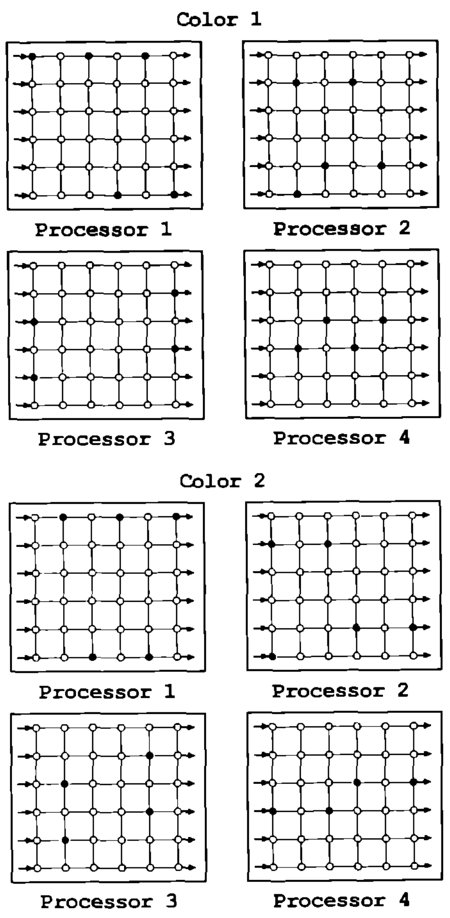

Fig. 2. Partitioning of each group of nodes with the same color into four blocks for concurrent SRM execution.

color is subdivided into P blocks of equal size. Then blocks of nodes with all the different colors are combined to form P blocks of mixed color. Each block becomes a task scheduled for parallel execution. Grouping nodes of different color together implies that adjacent nodes may be iterated upon concurrently. The task partitioning for ARM is illustrated in *Fig. 3*. Nodes that belong to a block are shown in black. Arcs between nodes in the same block are denoted

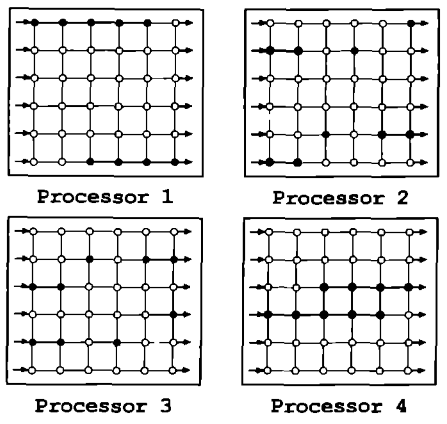

Fig. 3. Partitioning of the nodes into four blocks for concurrent ARM execution.

by bold lines. Notice that some adjacent nodes belong to different blocks and thus are scheduled to be processed in parallel.

3.2. Task scheduling

The two different modes of task scheduling are (1) static and (2) dynamic. In *static* task scheduling each processor is assigned the same group of nodes in each iteration. When groups of different nodes are assigned to a processor in different iterations then the mode of task scheduling is *dynamic*. In both SRM and ARM, we use a static mode of task scheduling.

A dynamic task scheduling scheme that would, presumably, result in improved load balancing was initially implemented. According to the dynamic scheme, after each minor iteration k, only nodes with $|d_s^k| > \epsilon$ would be assigned to blocks for processing. This scheme avoids creating blocks with several nodes satisfying primal feasibility conditions and hence not providing any work for the algorithm. However, the overhead of the scheduling routine adds to the total execution time, and eliminates any gains due to load balancing. Thus the dynamic scheme was abandoned. Techniques for improving load balancing with static scheduling are described in the next two sections.

3.2.1. Synchronous relaxation

The sets of nodes that belong to the same color are partitioned into as many blocks as there are processors and all blocks contain equal numbers of nodes. Each of the blocks is then scheduled to a processor that is the same throughout the execution, and processed in parallel. This is repeated until all sets are exhausted, and that marks the end of a *minor* iteration.

The partitioning of the sets into blocks with equal numbers of nodes ensures load balancing for problems in which the degrees of all nodes are (approximately) equal to each other. In problems where the degrees of the nodes vary significantly, load balancing can deteriorate if blocks are constructed selecting nodes randomly. Deterioration of load balancing can be prevented if blocks are constructed selecting nodes in such way that the total degrees of the blocks are approximately equal to each other. The coloring algorithm we use (coloring nodes in decreasing degree order) enables us to implement a load balancing scheme. After coloring nodes of the same color are sorted by their degrees in descending order. If blocks are constructed selecting a node with a high degree and a node with a low degree alternatingly, the total degrees of the nodes in all blocks will be approximately equal. We call this alternating selection *node shuffling*. The improvement of load balancing with node shuffling can be significant as shown in *Fig. 4*. In the *Figure* the average ratio of maximal to minimal block size (size is understood to be the sum of the node degrees in the block) is shown for our test problems.

3.2.2. Asynchronous relaxation

Nodes that are scheduled for concurrent execution could be adjacent in contradiction to the synchronous case. Therefore, blocks can be constructed with arbitrary node selection. However, in our implementation we try to have equal representation of colors in each block. To achieve this, blocks are constructed in the following way: nodes are colored and shuffled and blocks are built as in the synchronous case. One block from each color is then chosen and these K blocks are merged to form a larger block. The larger blocks are scheduled for concurrent processing.

3.3. Task synchronization

A theoretical analysis of several synchronization issues for both the asynchronous and the synchronous relaxation methods can be found in [5,13]. In this Section we discuss practical

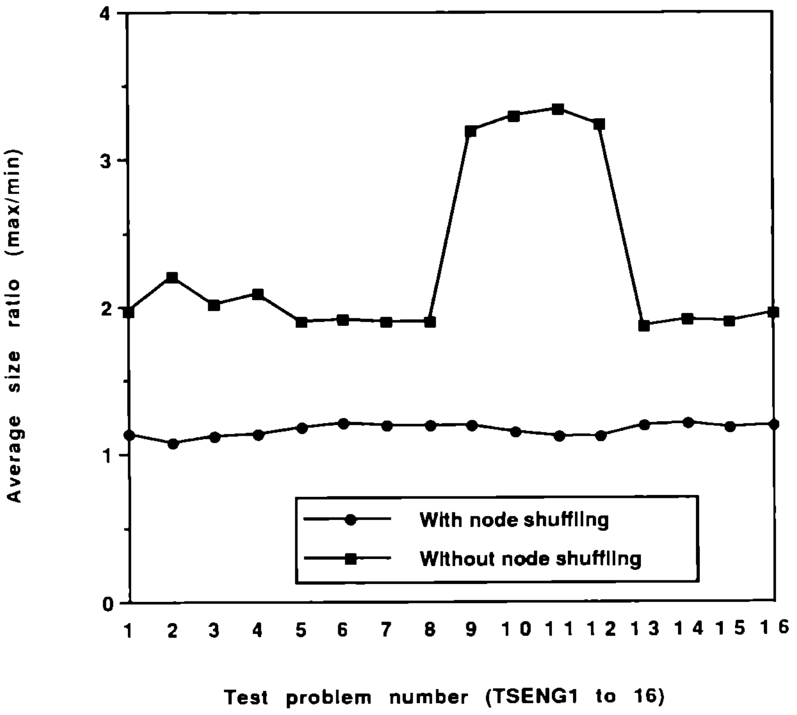

Fig. 4. Average ratio of maximal to minimal block size. (The size of the block is defined in the context of this figure as the sum of the degrees of all nodes in the block.)

approaches to synchronization problems that are encountered during the development of both asynchronous and synchronous implementations.

3.3.1. Synchronous relaxation

During task partitioning the nodes of a network are grouped into K sets, as many as the colors that the coloring heuristic yields. Each set is partitioned into P blocks that are scheduled for concurrent execution by the P processors. As a *phase* of the parallel computation we define one sweep of all the nodes in each of the blocks that are scheduled to be processed concurrently. A synchronization barrier is required to ensure that all nodes with the same color are operated upon, before the algorithm can proceed to the next color. Such a barrier is trivially enforced on the Alliant FX/8 with the CNCALL and NOSYNC compiler directives within the loop that iterates over the colors.

In SRM no memory access conflicts should occur during each phase of the parallel computation. The partitioning and scheduling schemes that we use ensure that there are no common arcs between nodes that have been scheduled for concurrent processing. *Figure 5* illustrates node i of color 2 and nodes j, k, l and m of color 1 adjacent to i, during a phase in which color 1 is scheduled for processing. Apparently there is no conflict in the updating of prices π_s, $s \in \{j, k, l, m\}$. The updating of the flows x_{li}, x_{mi}, x_{ij} and x_{ik} according to conditions (2.1)–(2.3) is also performed with no conflicts because the price at node i does not change in this phase. However, a memory access conflict may occur if two or more processors update deficit d_i simultaneously. Two different strategies have been employed to prevent conflicts when deficits are updated:

(i) Deficits of all nodes are computed at the end of a phase. This implementation is shown later to be inefficient, but we refer to it because it is instructive.

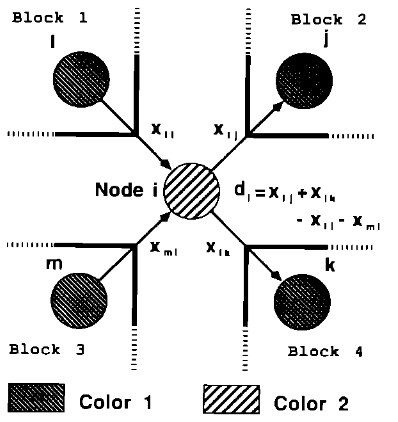

Fig. 5. Deficit updating at node *i* under SRM partitioning, scheduling and synchronization schemes.

(ii) Deficits are updated during the execution of a phase. For nodes of a color different than the one that is scheduled for the current phase, deficit updating becomes a *locked* operation; in other words, processors are forced to update sequentially instead of concurrently the deficit that results from a flow change. Locking the updating of deficits becomes possible through the use of appropriate lock monitors (see [7]).

The flow diagrams for the SRM implementation with calculation of deficits at the end of each phase and the SRM with locked deficit updating are shown in *Fig. 6* and *Fig. 7*, respectively. Each of the above deficit updating strategies induces inefficiencies in the implementation. First, when all deficits are computed at the end of a phase many unneccesary computations are performed since in a sparse network only a small proportion of the nodes will need to have their deficits updated after each phase. Second, when deficits are updated during a phase, unnecessary computations are avoided, but inefficiencies arise due to the locks. Overhead is also incurred when the locking and unlocking routines are used. All the above inefficiencies combined amount to a substantial synchronization penalty associated with SRM.

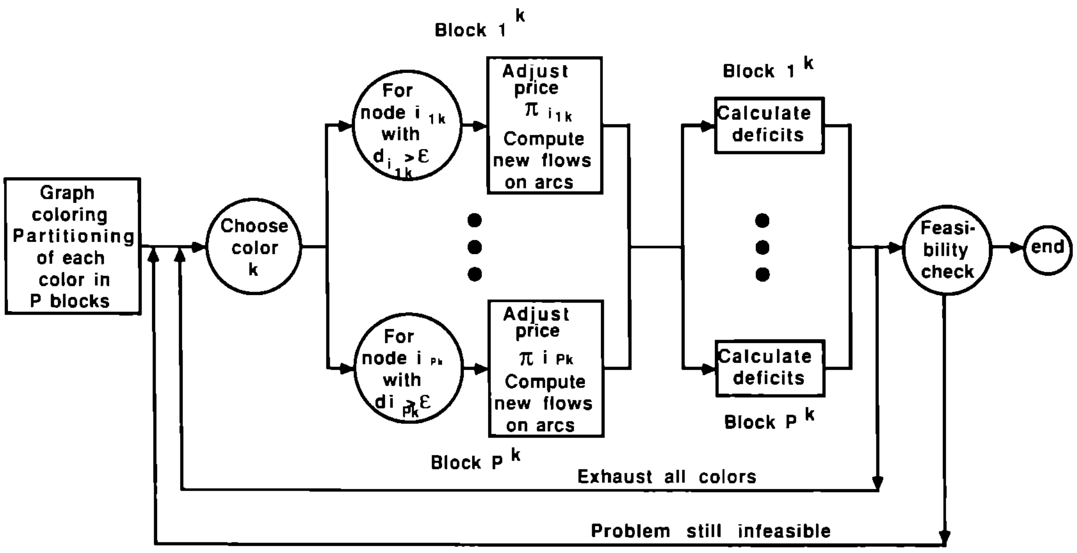
Block 1^k
For node i_{1k} with $d_{i_{1k}} > \varepsilon$
Adjust price $\pi_{i_{1k}}$ Compute new flows on arcs
Block 1^k
Calculate deficits
Graph coloring Partitioning of each color in P blocks
Choose color k
Feasi-bility check
end
For node i_{Pk} with $d_{i_{Pk}} > \varepsilon$
Adjust price $\pi_{i_{Pk}}$ Compute new flows on arcs
Calculate deficits
Block P^k
Block P^k
Exhaust all colors
Problem still infeasible

Fig. 6. Flow diagram of the SRM implementation with calculation of deficits at the end of each phase.

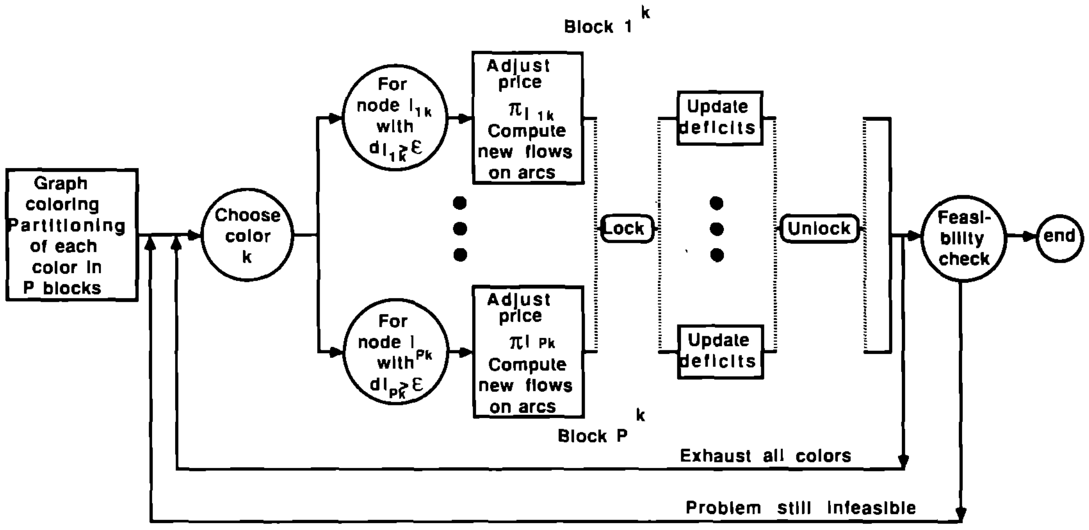
Block 1^k
Graph coloring Partitioning of each color in P blocks
Choose color k
For node i_{1k} with $d_{i_{1k}} \geq \varepsilon$
Adjust price $\pi_{i_{1k}}$ Compute new flows on arcs
For node i_{P_k} with $d_{i_{P_k}} \geq \varepsilon$
Adjust price $\pi_{i_{P_k}}$ Compute new flows on arcs
Block P^k
Lock
Update deficits
Update deficits
Unlock
Feasibility check
end
Exhaust all colors
Problem still infeasible

Fig. 7. Flow diagram of the SRM implementation with locked deficit updating.

A third implementation is possible if deficits are updated without locking. The flow diagram for this implementation is shown in *Fig. 8*. This implementation is, strictly speaking, not synchronous because memory access conflicts occur when deficits are updated, and the data integrity of the deficit vector cannot be guaranteed. The deficit values of nodes that had their prices adjusted are computed by the corresponding processor using accurate flow values. Nodes with different color may have their deficits updated by multiple processors working on adjacent nodes; these deficit values are not necessarily accurate. However, when the algorithm proceeds to iterate over nodes of different color the (possibly inaccurate) deficit values will be used to compute the first price-adjustment step, and following the first step the deficit of the node will be computed correctly from the latest accurate price information. Under this implementation

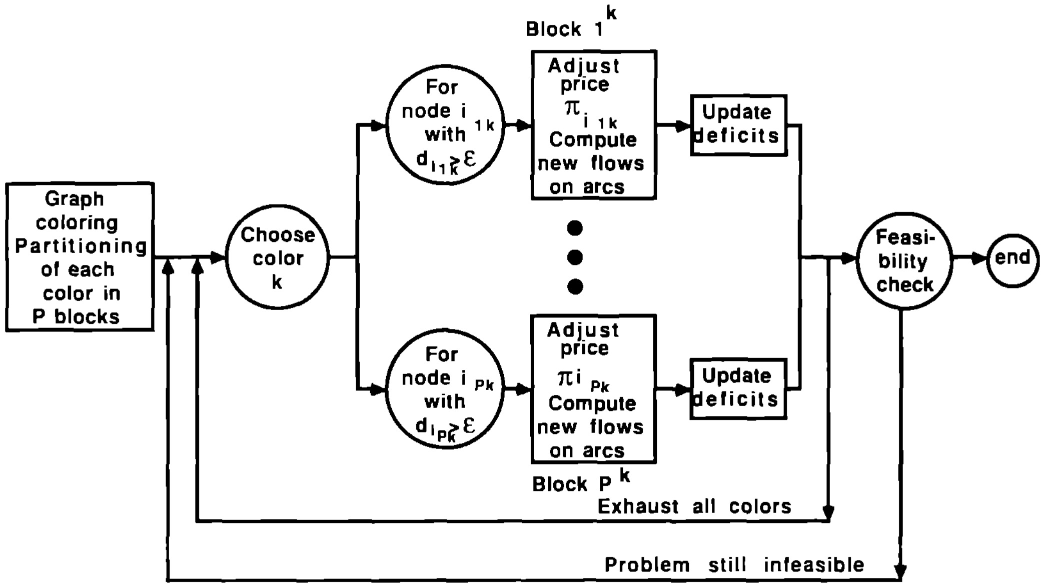
Block 1^k
Graph coloring Partitioning of each color in P blocks
Choose color k
For node i_{1k} with $d_{i_{1k}} \geq \varepsilon$
Adjust price $\pi_{i_{1k}}$ Compute new flows on arcs
Update deficits
For node i_{P_k} with $d_{i_{P_k}} \geq \varepsilon$
Adjust price $\pi_{i_{P_k}}$ Compute new flows on arcs
Update deficits
Block P^k
Feasibility check
end
Exhaust all colors
Problem still infeasible

Fig. 8. Flow diagram of the SRM implementation with unlocked deficit updating.

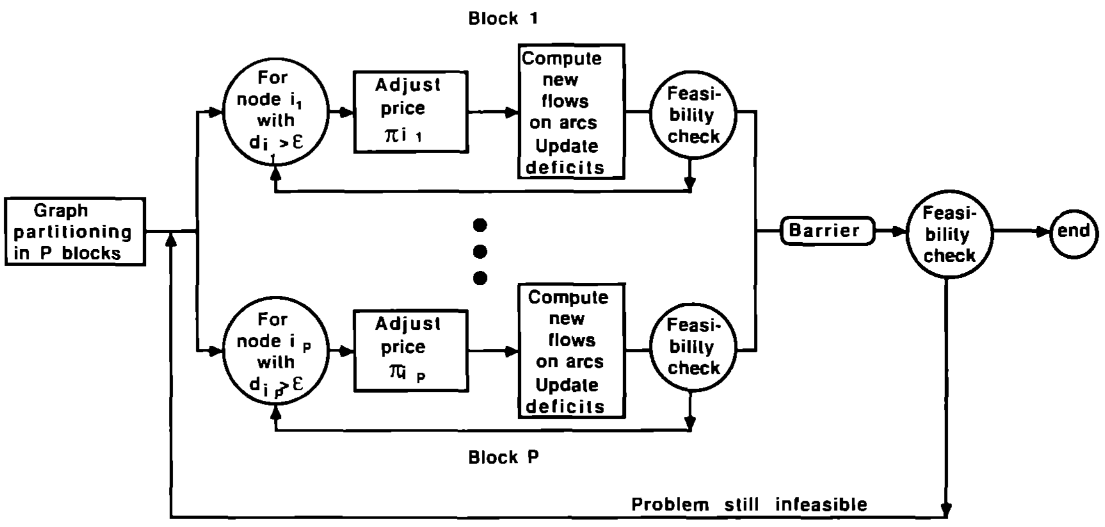
Block 1
For node i_1 with $d_i > \varepsilon$
Adjust price $\pi_{i\,1}$
Compute new flows on arcs Update deficits
Feasibility check
Graph partitioning in P blocks
Barrier
Feasibility check
end
For node i_P with $d_{i\,P} > \varepsilon$
Adjust price $\pi_{i\,P}$
Compute new flows on arcs Update deficits
Feasibility check
Block P
Problem still infeasible

Fig. 9. Flow diagram of the asynchronous implementation.

the problem of data integrity due to the absence of locks is a practical one and does not affect the asymptotic convergence of the algorithm.

It is very interesting to observe – see *Table 4* – that the number of iterations of the parallel unlocked implementation differs by less than 1% from the iterations of the sequential implementation. Hence, for the relatively sparce problems we solve here, and that are typically encountered in practice, the problem of maintaining data integrity is of no significance.

3.3.2. Asynchronous relaxation

The asynchronous method that we have implemented is described below. The ARM partitioning and scheduling schemes have been used. In this case all network nodes are grouped into P blocks that are scheduled for concurrent execution by all processors. Each minor iteration now consists of only one phase. The flow diagram for ARM is shown in *Fig. 9.* An

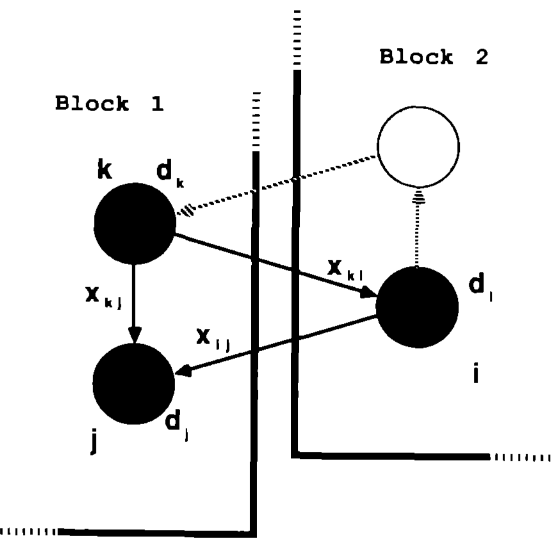

Fig. 10. Computation of flows and updating of deficits with ARM.

example of how ARM partitioning and scheduling induce asynchronicity is shown in *Fig. 10.* In this *Figure* parts of two blocks of nodes that are scheduled for concurrent processing are shown. Nodes k and j that are grouped in the same block can have a common arc (k, j). Node i belongs to a different block and can be connected to k and j with arcs (k, i) and (i, j), respectively. Price updating of nodes k and j, and the computation of the flow x_{kj} that results from each of the new prices are done sequentially. Hence, no conflicts occur during computations of flows on arcs that connect nodes in the same block. However, updating of the price of node i can be done concurrently with that for nodes k and j. This has as a result potential memory access conflicts during computation of flows x_{ij} and x_{ki} on arcs that connect nodes that belong to different blocks and during updating of the deficits on nodes i, j and k. As in the case of the unlocked synchronous algorithm our implementation ensures that these memory access conflicts do not affect the asymptotic convergence properties of the algorithm. They are instead a nuisance of practical importance, and may result into an increase in the number of iterations. The implementation uses the (possibly inaccurate) deficit values to compute the first price adjustment step for a given node. At that point the deficit is recalculated by estimating (by the processor assigned to the particular node) new flows based on the new local price and all available adjacent prices. The result is a new deficit value that accurately reflects the price information available to the processor. Of course the price read from memory for adjacent nodes may have been changed in the meantime by other processors, and hence it may be outdated. Differences in the price values are, however, permitted in the asynchronous algorithm [4].

The asynchronous implementation can be computationally more efficient than the previously described ones because most of the synchronization restrictions of SRM do not apply. In ARM it is not required that all processors finish computations for one phase before they start processing the next phase. Therefore, each processor can iterate independently on its scheduled block of nodes without the need to use synchronization points that induce overhead and occasionally deteriorate load balancing. Also, no operations require locking which is another source of delay. For these reasons the synchronization penalty for ARM is very small and high speedup factors can be achieved. However, the solution path followed by the asynchronous relaxation can be very different from that of the synchronous relaxation depending on the degree of asynchronicity. From the results in *Table 5* we observe that the asynchronous algorithm requires, for most problem instances, more iterations than its synchronous counterpart.

3.3.3. Finite termination

While the asynchronous algorithm converges asymptotically, we want in practice to terminate the algorithm when the primal feasibility error $\| d^k \|$ is sufficiently small for some finite k. Some results on the finite termination of asynchronous algorithms for fixed point problems are given in Bertsekas and Tsitsiklis [5, 4]. If finite termination criteria are not developed properly it is possible that the algorithm will terminate with a solution that is not within the required tolerance, or may not terminate at all. The former case was observed in the early stages of this research and an example of the latter case is given in [4]. We describe here our termination detection procedure.

Since processors iterate independently on their scheduled subgraphs, they would terminate computation when their subgraphs satisfy locally primal feasibility and complementary slackness conditions. However, after a processor terminates computation, prices in its subgraph may be changed by other processors that have not terminated yet. Therefore, the solution may not satisfy optimality conditions upon termination of all processors. A custom designed barrier was implemented to resolve the problem of premature termination. Each processor sends a termination message to all other processors when optimality conditions are satisfied in its

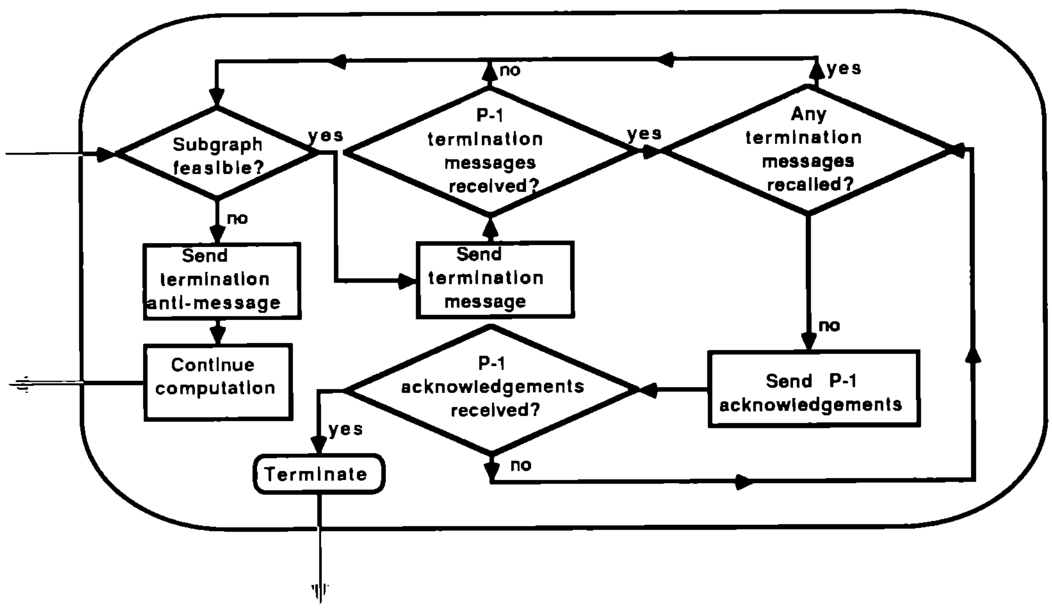
Subgraph feasible?
yes
no
Send termination anti-message
Continue computation
P-1 termination messages received?
no
yes
Send termination message
Any termination messages recalled?
yes
no
Send P-1 acknowledgements
P-1 acknowledgements received?
yes
no
Terminate

Fig. 11. Schematic diagram of the termination barrier for ARM.

subgraph. In turn, it receives termination messages from other processors. If termination messages have been received from all $P-1$ processors, the processor sends acknowledgements of receipt of their termination messages to all $P-1$ processors. Then it checks if acknowledgements for the reception of its termination message have been received from all $P-1$ processors. If yes, it terminates. If no, it checks for termination anti-messages (i.e., signals that cancel any previously sent termination messages). However, if termination messages have not been received from all $P-1$ processors or if any termination anti-messages have been received, the processor checks feasibility conditions again. If they are violated, a termination anti-message is sent and computation is resumed. A schematic description of how the barrier functions is shown in *Fig. 11*. This barrier has proved to be effective in preventing premature termination. Occasionally, however, due to communication delays in transmitting anti-messages it may detect false termination conditions. To resolve this, primal feasibility is checked after the end of computation. If it is violated, computation is resumed. Again, however, it is possible that upon termination the deficit will be above the target tolerance, ϵ. Hence, before restarting the algorithm the tolerance is reset to $\epsilon^{\text{new}} \leftarrow n_1\epsilon^{\text{old}}$. For sufficiently small value of n_1 the algorithm will terminate with $\| d^k \| < \epsilon^{\text{old}}$ even if $d_i^k \geq \epsilon^{\text{new}}$ for some nodes. Although difficulties with termination detection were observed in our work, they are expected to be more significant in distributed memory environments due to longer communication delays. Empirical observations on this problem are deferred until Section 4.4.

4. Computational results

We implemented both the synchronous and asynchronous algorithms on a vector multiprocessor. The algorithm was implemented in Fortran 77 on the Alliant FX/8 with up to 8 computational elements (CE), at ACRF at Argonne National Laboratory. The program was compiled with compiler options (– Ogvc) that produced a globally vectorized and concurrent code. In addition, compiler directives were used to implement the parallel schemes of Section 3. In this Section we present computational results with the parallel implementations. The

Table 1
Characteristics of the test problems

Problem	No. of colors	Nodes	Arcs	Coloring time (Alliant CPU secs)
TSENG1	5	500× 500	5063	0.26
TSENG2	6	750× 750	7611	0.48
TSENG3	6	1000×1000	10126	0.74
TSENG4	6	1250×1250	12665	1.08
TSENG5	2	500× 500	10051	0.22
TSENG6	2	750× 750	15086	0.37
TSENG7	2	1000×1000	20134	0.56
TSENG8	2	1250×1250	25153	0.80
TSENG9	10	500× 500	10000	0.33
TSENG10	10	750× 750	15000	0.53
TSENG11	10	1000×1000	20000	0.76
TSENG12	10	1250×1250	25000	1.04
TSENG13	2	500× 500	10068	0.22
TSENG14	2	750× 750	15061	0.38
TSENG15	2	1000×1000	20127	0.56
TSENG16	2	1250×1250	25149	0.78

objective of our experiments is twofold. First we want to compare the parallel performance of all implementations. Second, we want to establish the efficiency of the parallel implementation in solving very large problems.

4.1. Test problems

The test problems were generated using a modification of the NETGEN generator [11]. They are generated using the parameters specified in Tseng [12] – the test problem specifications appeared in an early working paper version but not in the published article and Ref. [17] specifies the test problems. *Table 1* shows the characteristics of the test problems and for each one of them the CPU time in seconds for coloring and partitioning sets into blocks on the Alliant FX/8. In this *Table* TSENG1 to TSENG8 are uncapacitated transportation problems, TSENG9 to TSENG12 are capacitated transportation problems and TSENG13 to TSENG16 are capacitated transportation problems. All the computational results presented here are obtained using $\epsilon = 10^{-3}(\Sigma_{i=1}^{m} b_i)/m$.

4.2. Synchronous relaxation

The speedup curves for the SRM implementations on the Alliant FX/8 are shown in *Figs. 12* and *13*. The speedup factor of the implementation with computation of deficits at the end of each phase ranges from 5.89 to 7.34. *Figure 12* shows speedup factors for problems TSENG4 and TSENG5 with increasing number of CE. The high speedup factors observed are due to the fact that the computation of the deficits at the end of each phase parallelizes very well and requires a relatively large proportion of the total time. Here the synchronization penalty results mainly from computing all the deficits, whether they change or not, and that reflects on the large solution times. The implementation in which deficits of all nodes are updated during each phase and locks are used where memory access conflicts may occur achieves speedup factors in the range of 3.34 to 5.71. This can be observed better in *Fig. 13*, where the speedup curves for TSENG2 and TSENG4 are plotted. However, in the latter implementation a significant drop in solution times, compared to the former, is realized as shown in *Tables 2* and *3* of the Appendix.

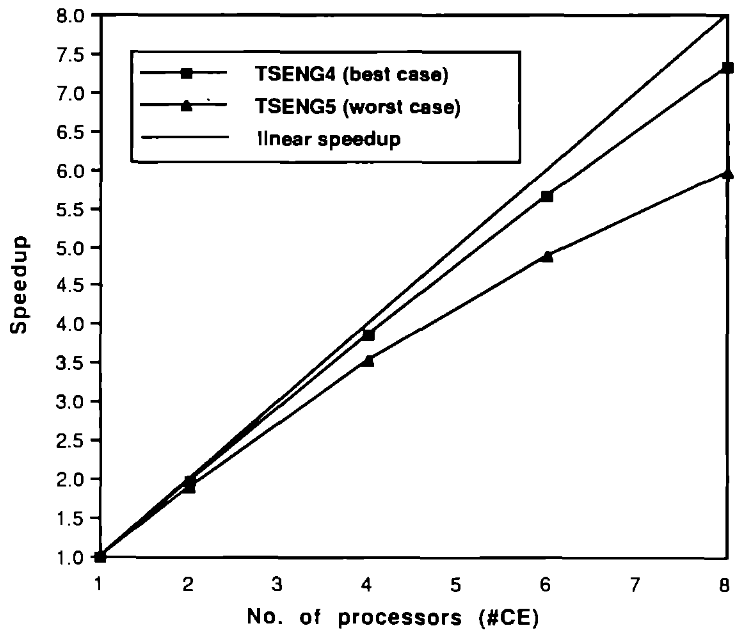

Fig. 12. Speedup curves for the SRM with computation of the deficits after each phase.

Solution times for SRM with unlocked deficit updating are even lower compared to SRM with locked deficit updating and speedup factors are in the range 3.41–6.16. The number of iterations is different in each run as opposed to the synchronous or, equivalently, the sequential implementations, that converge in exactly the same number of iterations in each run. This reflects the asynchronicity that has been introduced in this implementation. However, both the

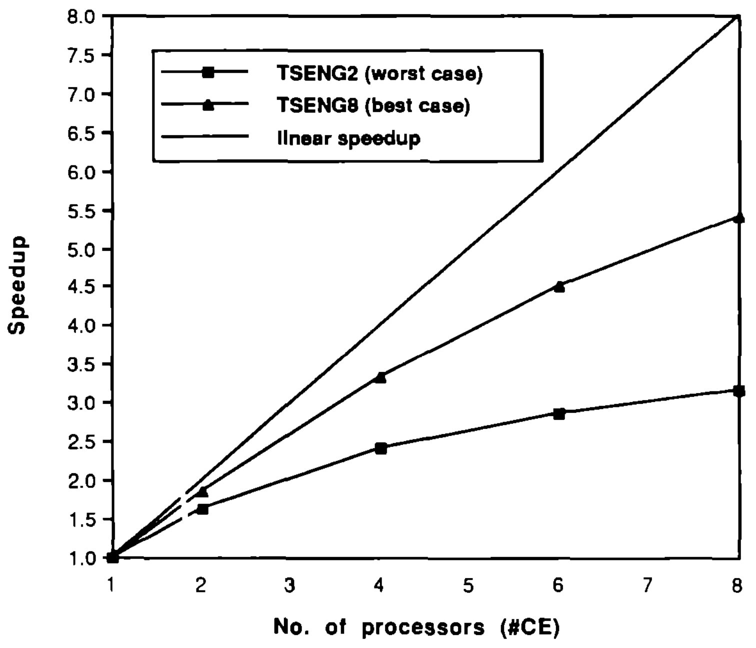

Fig. 13. Speedup curves for the SRM with locked deficit updating.

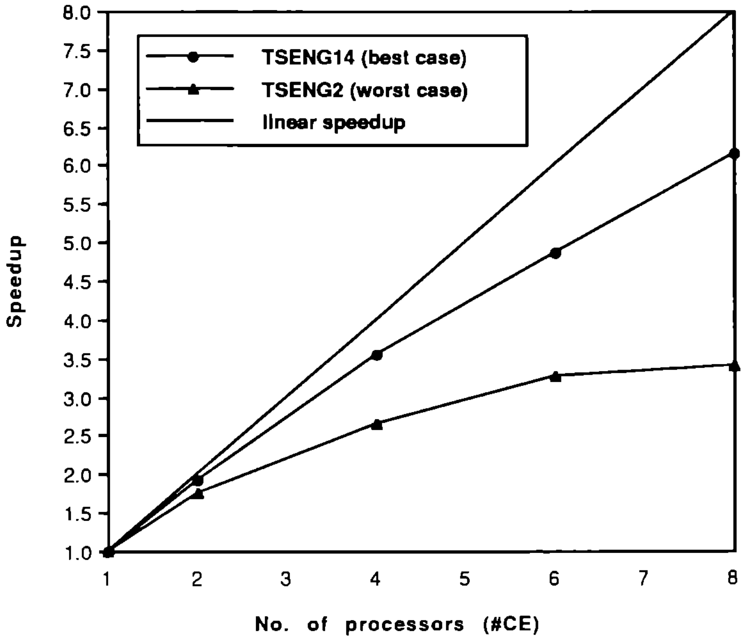

Fig. 14. Speedup curves for the SRM with unlocked deficit updating.

difference in the number of iterations between implementations and the variation in the number of iterations among different runs are small. *Figure 14* shows the speedup curves for TSENG2 and TSENG14. Detailed experimental results are contained in *Table 4* of the Appendix.

4.3. Asynchronous relaxation

Solution times for the asynchronous implementation are lower than that for any of the previous implementations. Speedup factors are higher and in the range of 3.04 to 7.17. *Figure 15* shows the speedup curves for TSENG2 and TSENG13 (speedup ratios of 3.04 and 7.17, respectively). The reason for this efficiency improvement is the removal of any synchronization points. The total number of iterations is higher for all but three of the test problems with the asynchronous than with the synchronous implementation because ARM works with possibly outdated estimates of the prices and deficits at each iteration. To get an estimate of the variance in solution times of the asynchronous algorithm we solved problem TSENG1 ten times. The average observed time was 3.27 sec with a standard deviation 0.07. The small value of the standard deviation is attributed to the tightly coupled parallelism of the Alliant FX/8. It is expected that the variance in solution times could be substantially higher for loosely coupled architectures. ARM achieves the same objective value as SRM for the same final tolerance ϵ and upon termination, primal and dual feasibility and complementary slackness conditions are satisfied. *Table 5* of the Appendix contains detailed experimental results of the asynchronous implementation. *Figure 16* illustrates a comparison of the solution times for the SRM and ARM implementations.

4.4. Termination detection

To avoid some of the inefficiencies associated with finite termination detection, we implemented two termination criteria. First, processors would terminate local computations when

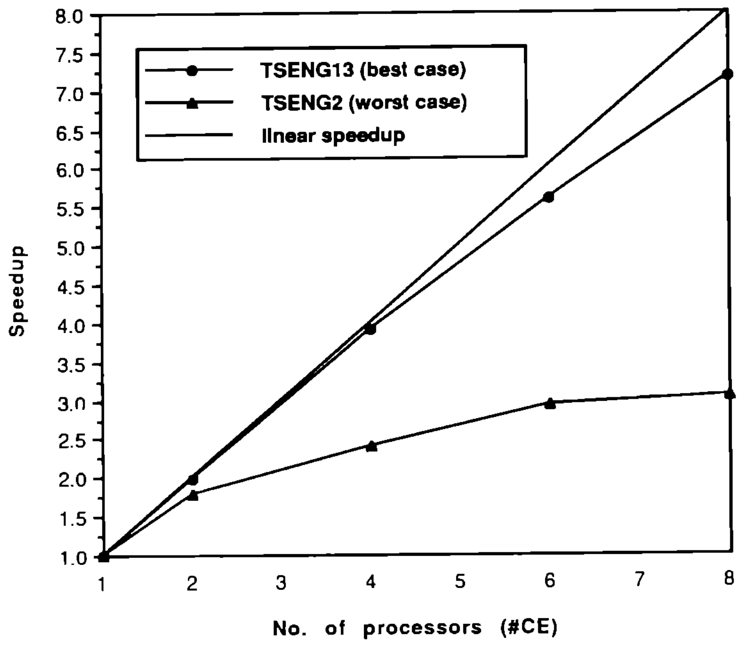

Fig. 15. Speedup curves for the asynchronous relaxation method.

$d^k < \epsilon_{\text{local}} = \alpha\epsilon$ for a final tolerance ϵ and $\alpha \in (0,1)$, and then would spin in front of the barrier as described in Section 3.3.3 until they receive a message to resume computing or until all processors have terminated. Upon termination of all processors the deficit vector was checked once more and if any components would exceed ϵ then we would reset $\epsilon_{\text{local}} \leftarrow n_1\epsilon_{\text{local}}$ and

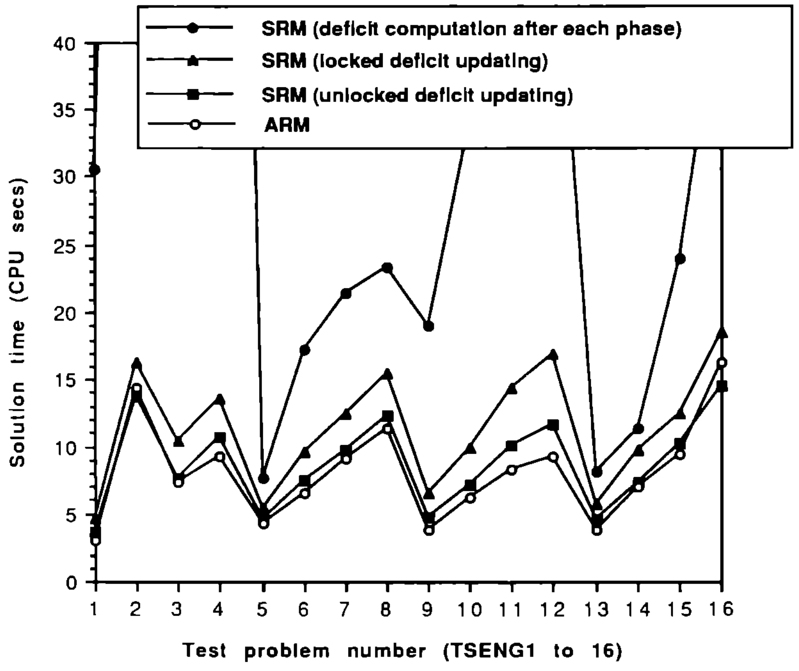

Fig. 16. Comparison of solution times among implementations on the Alliant FX/8.

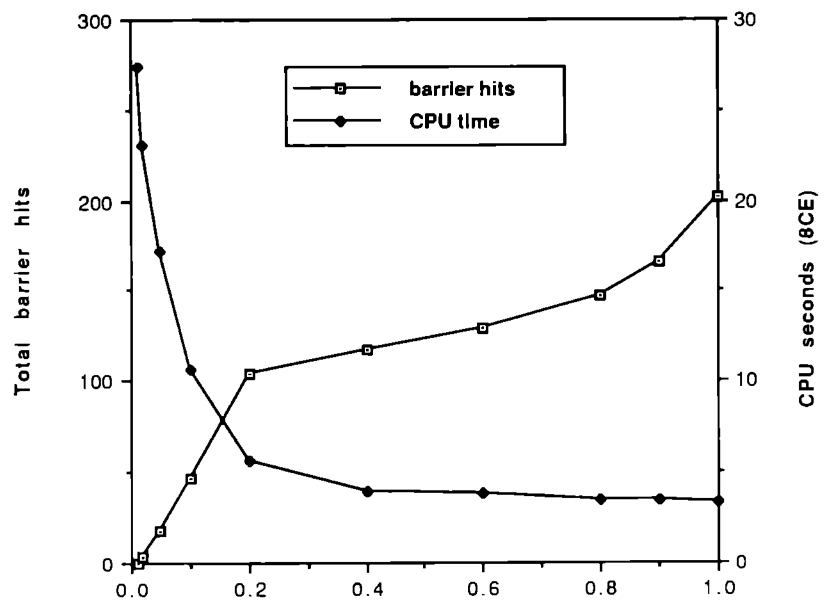

Fig. 17. Total number of barrier hits and total solution time for varying values of α.

resume calculations. The program would only infrequently need to resume calculations at this step. In no occasion did the program have to restart more than once. This observation is attributed to the very short memory access delays of the shared memory architecture.

Quite often, however, processors that were spinning idle in front of the termination barrier would be instructed to restart. The maximum number of barrier hits would depend on the value of α (recall that a processor would hit the barrier when the local deficit norm was less than $\alpha\epsilon$). *Figure 17* illustrates the total number of barrier hits and the total solution time for varying values of α. For our relatively sparse test problems and the tightly coupled parallel architecture the value $\alpha = 1$ appears to be optimal. It would be interesting to design an experiment to investigate optimal values of α for different levels of asynchronicity and communication delays.

5. Conclusions

We discussed in this paper parallel implementations of a nonlinear network optimization relaxation algorithm. Theoretical analysis and simulations indicated that the algorithm's intrinsic structure makes it a genuinely parallel optimization method. The results of our study on a shared memory architecture confirm these early hypotheses. The relaxation algorithm is a typical example of how the performance of an optimization method can be enhanced due to advanced computer architectures. This study also provides the first empirical verification that asynchronous algorithms can outperform their synchronous counterparts for nonlinear network-flow problems. This observation has recently been confirmed by El Baz [10]. The additional work required by the asynchronous algorithm is completed in less time due to the increased efficiency of the parallel implementation.

Acknowledgements

This research was partially supported by NSF grants CCR-8811135 and ECS-8718971 and AFOSR grant 89-0145. Computing resources were made available through the ACRF of Argonne National Laboratory. Part of the computational experiments were conducted while the first author was attending the Summer Institute in Parallel Processing, held at Argonne National Laboratory in September 1989.

APPENDIX

Table 2
Results for the SRM with computation of the deficits after each phase

Problem	Iterations total/minor	CPU time		Speedup T1/T8
		1 proc.	8 proc.	
TSENG1	18744/336	216.74	30.57	7.09
TSENG2	61273/891	1039.88	146.51	7.10
TSENG3	50483/259	427.50	60.29	7.09
TSENG4	43225/586	1145.02	155.97	7.34
TSENG5	22843/114	45.65	7.75	5.89
TSENG6	31304/276	105.16	17.24	6.10
TSENG7	42299/239	128.95	21.45	6.01
TSENG8	57484/169	140.36	23.32	6.02
TSENG9	20589/56	122.96	19.04	6.46
TSENG10	30395/64	225.21	33.86	6.65
TSENG11	40554/125	428.39	62.79	6.82
TSENG12	48692/87	355.31	54.26	6.55
TSENG13	22069/135	48.64	8.10	6.00
TSENG14	36979/101	68.80	11.42	6.02
TSENG15	41294/302	149.60	24.04	6.22
TSENG16	58816/511	279.33	44.10	6.33

Table 3
Results for the SRM with locked deficit updating

Problem	Iteration total/minor	CPU time		Speedup T1/T8
		1 proc.	8 proc.	
TSENG1	18741/336	18.92	4.61	4.10
TSENG2	60470/817	54.62	16.34	3.34
TSENG3	50481/257	47.03	10.37	4.54
TSENG4	43225/586	51.62	13.65	3.78
TSENG5	22843/114	31.52	5.52	5.71
TSENG6	31304/276	45.06	9.54	4.72
TSENG7	42299/239	60.39	12.54	4.82
TSENG8	57484/169	83.16	15.54	5.35
TSENG9	20589/56	29.13	6.61	4.41
TSENG10	30325/70	44.96	9.96	4.51
TSENG11	40844/108	61.22	14.37	4.26
TSENG12	49196/66	74.93	16.89	4.44
TSENG13	22068/134	31.14	5.83	5.34
TSENG14	36979/101	51.11	9.77	5.23
TSENG15	41294/302	59.25	12.56	4.72
TSENG16	58598/480	83.02	18.54	4.48

Table 4
Results for the SRM with unlocked deficit updating

Problem	Iteration		CPU time		Speedup T1/T8
	1 proc	8 proc	1 proc.	8 proc.	
TSENG1	18741/336	18753/338	16.37	3.69	4.44
TSENG2	60570/817	59926/757	47.12	13.81	3.41
TSENG3	50481/257	50514/262	39.61	7.69	5.15
TSENG4	43225/586	43211/586	45.91	10.79	4.25
TSENG5	22843/114	22912/114	28.01	4.70	5.96
TSENG6	31304/276	31300/274	40.17	7.49	5.36
TSENG7	42299/239	42249/228	53.52	9.78	5.47
TSENG8	57484/169	57537/185	72.87	12.34	5.91
TSENG9	20589/56	20597/52	24.45	4.81	5.08
TSENG10	30325/70	30318/70	37.68	7.26	5.19
TSENG11	40844/108	40852/107	51.22	10.10	5.07
TSENG12	49196/66	49177/65	62.23	11.73	5.31
TSENG13	22068/134	22051/131	27.93	4.71	5.93
TSENG14	36979/101	36901/101	44.89	7.29	6.16
TSENG15	41294/302	41263/302	52.77	10.31	5.12
TSENG16	58598/480	58550/479	74.89	14.51	5.16

Table 5
Results for the asynchronous relaxation method

Problem	Iterations		CPU time		Speedup T1/T8
	1 proc	8 proc	1 proc	8 proc	
TSENG1	18492/306	18956/317	15.63	3.01	5.19
TSENG2	59844/792	62874/1972	44.03	14.47	3.04
TSENG3	50508/254	53971/640	38.26	7.31	5.23
TSENG4	41698/500	42452/528	43.14	9.24	4.67
TSENG5	22322/97	23847/289	27.60	4.33	6.37
TSENG6	30969/289	31325/404	39.51	6.61	5.98
TSENG7	41960/234	43237/458	52.74	9.15	5.76
TSENG8	56328/157	57311/392	71.77	11.41	6.29
TSENG9	20452/57	22680/259	24.34	3.88	6.27
TSENG10	30632/75	33647/439	37.58	6.20	6.06
TSENG11	40738/116	43127/444	50.95	8.34	6.11
TSENG12	47982/67	50884/384	61.54	9.33	6.60
TSENG13	21185/135	20976/193	27.66	3.86	7.17
TSENG14	36407/116	38374/242	44.36	6.99	6.35
TSENG15	41456/296	42007/616	52.40	9.40	5.57
TSENG16	58748/510	60217/1055	73.88	16.39	4.51

References

[1] D.P. Ahfeld, R.S. Dembo, J.M. Mulvey and S.A. Zenios, Nonlinear programming on generalized networks, *ACM Trans. Math. Software* 13 (4) (December 1987) 350–367.

[2] Alliant Computer Systems Corporation, Acton, MA, *FX/FORTRAN Programmer's handbook*, May 1985.

[3] D.P. Bertsekas and D. El Baz, Distributed asynchronous relaxation methods for convex network flow problems, *SIAM J. Control Optimiz.* 25 (1) (January 1987) 74–85.

[4] D.P. Bertsekas and J.N. Tsitsiklis, Convergence rate and termination of asynchronous iterative algorithms, in: *Proc. 1989 Internat. Conf. on Supercomputing*, Crete Greece, June 1989, 461–470.

[5] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods* (Prentice-Hall, Englewood Cliffs, 1989).

[6] D.P. Bertsekas, P.A. Hosein and P. Tseng, Relaxation methods for network flow problems with convex arc costs, *SIAM J. Control Optimiz.* 25 (1) (January 1987) 74–85.

[7] J. Boyle, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson and R. Stevens, *Portable Programs for Parallel Processors* (Holt, Rinehart and Winston, New York, 1987)

[8] N. Christofides, *Graph Theory: An Algorithmic Approach* (Academic Press, New York, 1975).

[9] R.S. Dembo, J.M. Mulvey and S.A. Zenios, Large scale nonlinear network models and their applications, *Oper. Res.* 37 (1989) 353–372.

[10] D. El Baz, A computational experience with distributed asynchronous iterative methods for convex network flow problems, in: *Proc. 28th Conf. on Decision and Control,* Tampa, Florida, December 1989.

[11] D. Klingman, A. Napier and J. Stutz, Netgen – a program for generating large scale (un)capacitated assignment, transportation and minimum cost flow network problems, *Manage. Sci.* 20 (1974) 814–822.

[12] P. Tseng, Dual coordinate ascent for problems with strictly convex costs and linear constraints: a unified approach, *SIAM J. Control Optimiz.* 28 (1) (January 1990) 214–242.

[13] P. Tseng, D.P. Bertsekas and J.N. Tsitsiklis, Partially asynchronous parallel algorithms for network flow and other problems, *SIAM J. Control Optimiz.* 28 (3) (May 1990) 678–710.

[14] S.A. Zenios, Parallel numerical optimization: current status and an annotated bibliography, *ORSA J. Comput.* 1 (1) (1989) 20–43.

[17] S.A. Zenios and Y. Censor, Parallel row-action algorithms for some nonlinear transportation problems, *SIAM J. Optimization* 1 (3) (Aug. 1991) 373–400.

[15] S.A. Zenios and R.A. Lasken, Nonlinear network optimization on a massively parallel connection machine, *Ann. Oper. Res.* 14 (1988) 147–165.

[16] S.A. Zenios and J.M. Mulvey, A Distributed algorithm for convex network optimization problems, *Parallel Comput.* 6 (1988) 45–56.

[18] S. Nielsen and S.A. Zenios, Massively parallel algorithms for singly constrained nonlinear programs, ORSA J. Comput., to appear.