Parallel Algorithms for Solving the Convex Minimum Cost Flow Problem
Beraldi, P;Guerriero, F;Musmanno, R
Computational Optimization and Applications; Feb. 2001; 18, 2; SciTech Premium Collection
pg. 175

Computational Optimization and Applications, 18, 175–190, 2001
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

Parallel Algorithms for Solving the Convex Minimum Cost Flow Problem

P. BERALDI AND F. GUERRIERO
Dipartimento di Elettronica, Informatica e Sistemistica, Università della Calabria, 87036 Rende (CS), Italy

R. MUSMANNO
Facoltà di Ingegneria, Università degli Studi di Lecce, 73100 Lecce, Italy

Received January 3, 1996; Accepted August 13, 1999

Abstract. In this paper we deal with the solution of the separable convex cost network flow problem. In particular, we propose a parallel asynchronous version of the ϵ-relaxation method and we prove theoretically its correctness.

We present two implementations of the parallel method for a shared memory multiprocessor system, and we empirically analyze their numerical performance on different test problems. The preliminary numerical results show a good reduction of the execution time of the parallel algorithm with the respect to the sequential counterpart.

Keywords: convex minimum cost flow problem, ϵ-relaxation method, shared memory multiprocessor, asynchronous parallel algorithms, speedup

1. Introduction

In this paper we consider the minimum cost flow problem with convex separable cost function (MCCF, for short). This problem has been widely studied because of its practical and algorithmic importance, and a large number of algorithms have been proposed for its solution.

They can be grouped into two different categories: *exact* methods by which the optimal solution is obtained [1–3] and *approximate* methods that determine a solution which is optimal within a certain user-defined tollerance [4–10].

In this paper we concentrate our attention on the approximate methods. All these methods relax in some way the complementary slackness conditions and they differ in the strategy used to move toward to optimality. Among them, the ϵ-relaxation method [8–10] is one of the most promising approach. It presents the same worst case theoretical complexity of the other approximate methods, but it exhibits better computational performance, at least in conventional computing settings [7, 8]. Furthermore, this method is very appealing since it is potentially well suited to implementation on parallel computing systems.

The paper is organized as follows. In the Section 2, we introduce the MCCF problem and the ϵ-relaxation method. In Section 3, we present a corresponding parallel asynchronous algorithm, which can be viewed as an extension of the method developed by Bertsekas and Tsitsiklis [11] when the cost function is linear. In the same section we formally illustrate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

the correctness of the parallel method, adapting the convergence proof given in [11] to the convex cost function case.

In Section 4, we describe parallel implementations on a shared memory multiprocessor. Finally, in Section 5, we present and discuss some computational results collected on a large set of test problems.

2. The convex network flow problem

Given a directed graph $G = (N, A)$, where N is the set of nodes with $|N| = n$ and A is the set of arcs with $|A| = m$, the MCCF problem can be formulated as follows:

$$\min \sum_{(i,j)\in A} f_{ij}(x_{ij}) \tag{1}$$

$$\text{s.t.} \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = s_i, \quad \forall i \in N \tag{2}$$

where $f_{ij} : R \to (-\infty, +\infty]$, $\forall (i, j) \in A$, is a convex, closed, proper function (extended real-valued, lower semicontinuous, not identically taking the value ∞ [4]); x_{ij}, $\forall (i, j) \in A$, represents the number of units of flow through the arc (i, j) from node i to node j. Furthermore, s_i, $\forall i \in N$, is the supply/demand of node i, depending whether its value is greater/less than zero. We refer to constraints (2) as the *conservation of flow* constraints.

For each function f_{ij}, we denote with l_{ij} and u_{ij}, respectively, the left and right endpoints of the effective domain $C_{ij} = \{\xi \in R \mid f_{ij}(\xi) < \infty\}$.

We make the following assumptions.

Assumption 2.1. The MCCF problem is feasible, that is, there exists at least one flow vector x satisfying the flow conservation constraints (2) and its components x_{ij} belong to C_{ij}, $\forall (i, j) \in A$.

Assumption 2.2. There exists at least one feasible flow vector x such that: $f_{ij}^-(x_{ij}) < \infty$ and $f_{ij}^+(x_{ij}) > \infty$, $\forall (i, j) \in A$, where $f_{ij}^-(x_{ij})$ and $f_{ij}^+(x_{ij})$ denote, respectively, the left and the right directional derivative of f_{ij} at x_{ij}.

The ϵ-relaxation method for solving the MCCF problem can be viewed as a generalization of the method proposed in [13] for the linear minimum cost flow problem. The method is based on the satisfaction of the *ϵ-complementary slackness* conditions.

Let p_i be the price of node $i \in N$. Given a scalar $\epsilon > 0$, we say that a flow-price vector pair (x, p) satisfies the ϵ-complementary slackness conditions (ϵ-CS for short) if and only if

$$f_{ij}^-(x_{ij}) - \epsilon \le p_i - p_j \le f_{ij}^+(x_{ij}) + \epsilon, \quad \forall (i, j) \in A. \tag{3}$$

It can be shown that, if a feasible flow-price vector pair (x, p) satisfies ϵ-CS, then the cost corresponding to x is optimal within a factor proportional to ϵ [8].

In the sequel, some terminology and computational operations are introduced.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Definition 2.1. Given a flow distribution x, the surplus g_i of a node i is the difference between the supply s_i and the net outflow from i, that is:

$$g_i = \sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} + s_i. \tag{4}$$

Definition 2.2. Given a flow-price vector pair (x, p) satisfying ϵ-CS, the push list L_i of node i, $\forall i \in N$, is defined as follows:

$$\begin{aligned} L_i = &\{(i, j) \mid \epsilon/2 < p_i - p_j - f_{ij}^+(x_{ij}) \leq \epsilon\} \\ &\cup \{(j, i) \mid -\epsilon \leq p_j - p_i - f_{ji}^-(x_{ji}) < -\epsilon/2\}. \end{aligned} \tag{5}$$

Definition 2.3. For each node i, the push list L_i contains unblocked arcs, that is, arcs (i, j) such that:

$$p_i - p_j \geq f_{ij}^+(x_{ij} + \delta), \tag{6}$$

or arcs (j, i) such that:

$$p_j - p_i \leq f_{ji}^-(x_{ji} - \delta), \tag{7}$$

where δ is a given positive scalar.

Definition 2.4. Given the push list L_i of node i and an unblocked arc $a = (i, j)$ [or $a = (j, i)$], the flow margin of the arc a is the supremum of δ for which the relation (6) [or (7)] holds.

The ϵ-relaxation method starts with a flow vector x such that $x_{ij} \in C_{ij}$, $\forall (i, j) \in A$, and a price vector p such that the flow-price vector pair (x, p) satisfies ϵ-CS. At each iteration, a node i with positive surplus (referred in the following to as *active* node) is selected, and one (or more) of the two following basic operations are performed on it.

1. A *price rise*, which consists of increasing the price p_i by the maximum amount that maintains ϵ-CS, whereas the flow vector x and the price p_j, $\forall j \in N - \{i\}$, are left unchanged.
2. A *flow push* along an arc (i, j) [or along an arc (j, i)] that consists of increasing the flow on arc (i, j) [or decreasing the flow on arc (j, i)] by an amount $\delta \in (0, g_i]$, while leaving all other flows as well as the price vector unchanged.

The typical iteration on node i is as follows.

1. (*Scan the push list L_i*)
   If $L_i = \emptyset$ go to 3.
2. (*Decrease the surplus of node i*)
   Choose an arc $a \in L_i$ and perform a δ-flow push along it, where

   $$\delta = \min\{g_i, \text{flow margin of } a\}.$$

   If $g_i = 0$, go to the next iteration; otherwise go to 1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

3. (*Increase the price of node* i)
   Execute a price rise operation on i. Go to the next iteration.

For a feasible problem, the ϵ-relaxation method terminates in a finite number of iterations when $g_i = 0$, $\forall i \in N$, with a flow vector that is optimal within a factor that is proportional to ϵ [8].

3. The parallel ϵ-relaxation method

This section is devoted to the description of a parallel asynchronous version of the ϵ-relaxation method for MCCF problem. The proposed parallel algorithm can be viewed as an extension of the method developed for the case of linear cost function by Bertsekas and Tsitsiklis [11], who have also demonstrated the theoretical convergence properties (theoretical results for the linear case have also been discussed in a simpler form by Li and Zenios in [12]).

The ϵ-relaxation method described in the previous section presents a quite simple structure, that makes it well suited to be implemented on parallel systems.

Perhaps, the easiest way to parallelize the method consists of selecting, at each iteration, several non-adjacent active nodes and performing the two basic operations of the sequential method (price rise and flow push) concurrently. However, such an implementation could be not very efficient, since the number of non-adjacent active nodes could be relatively small in the course of the algorithm. Thus, it might not be possible to take full advantage of the parallelization.

A much more efficient version can be obtained by allowing simultaneous iterations on adjacent active nodes. In this case, it may happen that two processors could update simultaneously the flow value along the same arc. For this reason, in order to guarantee the correct termination of the parallel method, some synchronization mechanisms are needed.

In the following, we describe the details of this parallel asynchronous version and we prove its theoretical convergence properties.

In order to design a parallel method valid for both shared and distributed memory systems, we refer to a theoretical computational model that consists of a set of N_P processors, each with its own local memory, that communicate through a global memory or an interconnection network.

We assume that the computation proceeds in *supersteps*, each consisting of an *input* phase, a *computational* phase, and an *output* phase.

In the input phase, a processor can receive information from other processors (i.e., it can read data from the shared memory, or receive data sent to it from other processors); in the computational phase, it performs local computation; in the output phase, it communicates data to other processors (i.e., by sending them through communication links or writing them into the global memory).

For the sake of simplicity, in the sequel, we assume that each node i is assigned to a separate processor P_i (i.e., the number of available processors N_P is equal to the number of nodes n).

This makes no theoretical difference if we consider the most general case when $N_P < n$ (in this case, more nodes have to be assigned to each processor in some way).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

The parallel asynchronous scheme for the ϵ-relaxation method can be formally described as follows.

Each processor P_i executes the typical iteration at node i. The input and output phases involve communication with the *adjacent* processors P_j, $\forall j \in F_i \cup B_i$, where

$$F_i = \{j \mid (i, j) \in A\},$$

and

$$B_i = \{j \mid (j, i) \in A\}.$$

At any time t, each processor P_i holds the following values:

- $p_i(t)$: the price of node i;
- $p_j(i, t)$: the price of node $j \in F_i \cup B_i$ communicated by P_j at some earlier time;
- $x_{ij}(i, t)$: the estimate of the flow on the arc (i, j), $j \in F_i$, available at processor P_i at time t;
- $x_{ji}(i, t)$: the estimate of the flow on the arc (j, i), $j \in B_i$, available at processor P_i at time t.

We assume that the price and the flow values can change only at an increasing sequence of times $t_0, t_1, \ldots, t_m$, with $t_m \to \infty$. At each time t, the processor P_i can execute one of the following three phases.

1. *Computational Phase.* P_i computes the surplus $g_i(t)$:

   $$g_i(t) = \sum_{\{j:(j,i)\in A\}} x_{ji}(i, t) - \sum_{\{j:(i,j)\in A\}} x_{ij}(i, t) + s_i.$$

   If $g_i(t) > 0$, then the typical iteration is executed and the following values

   $$p_i(t), \quad x_{ij}(i, t), \ j \in F_i, \quad x_{ji}(i, t), \ j \in B_i$$

   are updated.
2. *Output Phase.* The values of $p_i(t)$, $x_{ij}(t)$, $x_{ji}(t)$, computed during the computational phase, are communicated to the adjacent processors P_j, $j \in F_i \cup B_i$.
3. *Input Phase.* P_i receives the price $p_i(t')$ and the arc flow $x_{ij}(j, t')$ or $x_{ji}(j, t')$, computed by processor P_j, $j \in F_i \cup B_i$, at some earlier time $t' < t$.
   On the basis of this information, P_i updates $p_j(i, t)$ and $x_{ij}(i, t)$ if $j \in F_i$, ($x_{ji}(i, t)$, if $j \in B_i$).
   If $p_j(t') \geq p_j(i, t)$, then $p_j(i, t) = p_j(t')$.
   In addition, if $j \in F_i$, the value of $x_{ij}(i, t)$ is replaced by $x_{ij}(j, t')$ if

   $$p_i(t) < p_j(t') + f_{ij}^{+}(x_{ij}(j, t')) + \epsilon \quad \text{and} \quad x_{ij}(j, t') < x_{ij}(i, t).$$

   In the case of $j \in B_i$, the value of $x_{ji}(i, t)$ is replaced by $x_{ji}(j, t')$ if

   $$p_j(t') \geq p_i(t) + f_{ji}^{-}(x_{ji}(j, t')) - \epsilon \quad \text{and} \quad x_{ji}(j, t') > x_{ji}(i, t).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Let T^i be the set of times for which the computational phase is executed by processor P_i, and let $T^i(j)$ be the set of times when P_i receives new data from adjacent processors P_j, $j \in F_i \cup B_i$.

We make the following assumptions.

Assumption 3.1. T^i and $T^i(j)$ have an infinite number of elements for all processors P_i and P_j, $j \in F_i \cup B_i$.

Assumption 3.2. Old information is eventually purged from the system, that is, given any time t_k, there exists $t_m \geq t_k$ such that the computing time of the price and flow information obtained at any node after t_m (i.e, the time t' in the input phase) exceeds t_k.

Assumption 3.3. For each processor P_i, the initial arc flow $x_{ij}(i, t_0)$, $j \in F_i$, and $x_{ji}(i, t_0)$, $j \in B_i$, satisfy ϵ-CS together with $p_i(t_0)$ and $p_j(i, t_0)$, $j \in F_i \cup B_i$. Furthermore,

$$\begin{aligned} p_i(t_0) &\geq p_i(j, t_0) \quad \forall j \in F_i \cup B_i, \\ x_{ij}(i, t_0) &\geq x_{ij}(j, t_0) \quad \forall j \in F_i. \end{aligned}$$

The sketch of the typical iteration of the parallel method, the updating rules and the initial conditions imply the following properties.

1. The price sequence is monotonically nondecreasing in t and

$$p_i(t) \geq p_i(j, t''), \forall j \in F_i \cup B_i, t'' \leq t \tag{8}$$

2. ϵ-CS are locally satisfied for each node i:

$$\begin{aligned} f_{ij}^-(x_{ij}(i,t)) - \epsilon \leq p_i(t) - p_j(i,t) \leq f_{ij}^+(x_{ij}(i,t)) + \epsilon \quad \forall (i,j), \quad j \in F_i \\ f_{ji}^-(x_{ji}(i,t)) - \epsilon \leq p_j(i,t) - p_i(t) \leq f_{ji}^+(x_{ji}(i,t)) + \epsilon \quad \forall (j,i), \quad j \in B_i \end{aligned} \tag{9}$$

3. Processor P_i stores an estimate of arc flow $x_{ij}(i, t)$ which is greater than or equal to the value stored at processor P_j, that is:

$$x_{ij}(i,t) \geq x_{ij}(j,t), \quad \forall j \in F_i, \quad \forall t \geq t_0. \tag{10}$$

4. There exists a node which is never processed. This follows from the fact that the surplus of a node, once nonnegative, remains nonnegative and from (10) we obtain:

$$\sum_{i \in N} g_i(t) \leq 0, \quad \forall t \geq t_0. \tag{11}$$

This implies that at any time t, there is at least one node i with negative surplus if there is a node with positive surplus. At this node i, processor P_i must not have executed any iteration up to t, and, therefore, the price $p_i(t)$ must be equal to the initial price $p_i(t_0)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

We say that the parallel asynchronous version of the ϵ-relaxation method terminates if there is a time t_k such that, for all $t \geq t_k$, we have:

$$g_i(t) = 0 \quad \forall i \in N, \tag{12}$$

$$x_{ij}(i, t) = x_{ij}(j, t) \quad \forall (i, j) \in A, \tag{13}$$

$$p_j(t) = p_j(i, t) \quad \forall j \in F_i \cup B_i. \tag{14}$$

Now we are ready to show the correctness of the parallel algorithm. Our proof of convergence follows the same approach proposed for the linear case by Bertsekas and Tsitsiklis [11], by taking into account, however, the different type of cost function (convex instead of linear).

Proposition 3.1. *If the problem is feasible and Assumptions* 3.1–3.3 *hold, the algorithm terminates.*

Proof: Suppose no iterations are executed at any node after some time t^*. Then Eq. (12) must hold for large enough t. Because no iterations occur after t^*, Assumption 3.1, Eq. (8), and the updating rules defined in the input phase imply Eq. (14). Furthermore, after t^*, no flow estimates can change unless if new data are available. Note that the updating rules, Eq. (10), Assumptions 3.1 and 3.2 imply the consistency of the arc flow values as in Eq. (13).

We assume now that iterations are executed indefinately. In this case, for every t, there is a time $t' > t$ and a node i such that $g_i(t') > 0$. But this is impossible, since we observe that the number of price increases and the number of δ-flow pushes performed by the parallel asynchronous algorithm are bounded. This fact can be demonstrated by following the same approach used in [8] to show the correctness of the sequential method. □

4. Parallel implementations on a shared memory multiprocessor

In this section we describe different parallel asynchronous implementations of the ϵ-relaxation method designed for a shared memory multiprocessor.

A key issue of the proposed parallel method is related to the partitioning and allocation of the workload among the available processors, in such a way to guarantee a good load balancing.

In our case, the computational workload depends on the number of active nodes, since the method terminates when the surplus of all nodes is reduced to zero.

In principle, it is possible to consider two different allocation strategies: static and dynamic.

In the first case, the set of nodes is partitioned into N_P blocks of equal size, each containing the same number of active nodes, using a procedure executed only once, at the beginning of the algorithm. Each processor extracts nodes only from its private subset; consequently, there is a reduction of the synchronitazion overhead due to access to shared data, generally, performed through a specific mechanism such as lock. On the other hand, the main drawback of this strategy is the impossibility to guaranteeing a priori, during the execution of the algorithm, a good load balancing among the available processors. Indeed, following flow

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

push operations other nodes become active and there is no way to ensure that their number remains roughly the same in each block.

This limitation can be overcome by considering a dynamic node allocation strategy. The active nodes are stored into a FIFO queue L (i.e. nodes are extracted from the top of L and inserted at the bottom), shared among all processors. The main drawback of this strategy is due to the synchronization overhead: a lock is used in order to guarantee that a node could not be simultaneously selected by more than one processor. Empirical computational studies have revealed that, in the case of the ϵ-relaxation method, the dynamic allocation outperforms the static one [15]. For this reason, all the implementations presented in the sequel are based on dynamic allocation strategies.

A first parallel implementation of the ϵ-relaxation method can be easily derived from the parallel scheme introduced in the previous section.

Each processor stores into its local memory a private flow-price vector pair on the basis of which it executes the typical iteration of the method. Processors exchange information through the shared memory in which the global flow-price pair, the queue L of active nodes and a boolean variable *flag* are stored. At the beginning, each processor reads from the shared memory an *initial* flow-price vector pair computed in such a way that ϵ-CS are satisfied. The computation starts with the extraction of a node i from the queue L and proceeds with the execution of the basic operations on the extracted node (computational phase).

Once the surplus of node i is reduced to zero, the processor writes the new flow and price values into the shared memory (output phase) and warns the adjacent processors of their availability, by using the *flag*. Then, each processor gets the new data and, eventually, updates the local flow-price vector pair according to the rules defined in the input phase.

The main drawback of the proposed parallel algorithm, when implemented on a shared memory multiprocessor, is that the shared memory is not exploited in the most efficient way: data are copied from the main memory into the local ones, and viceversa.

More efficient implementations can be defined by using the main memory in a more appropriate way, by avoiding the use of local copies and maintaining only a global flow-price vector pair, shared among all processors.

Following this approach, we have considered two different parallel implementations of the asynchronous ϵ-relaxation method, that differ in the way of organizing the queue of active nodes. More specifically, our implementations are as follows.

- *Single Queue Implementation*. The active nodes are stored in a single queue L shared among all processors. Each processors P_i selects the active node i from L, computes the surplus of the node, stores its value in a local variable σ and executes the basic operations until σ is reduced to zero. This implementation allows the eventual simultaneous selection of adjacent nodes by two processors. This means that the flow values along the arcs betweeen the two nodes could be updated in a non-predeterminated order and, consequently, the value of σ could be not consistent with the current flow distribution.

  For this reason, in order to guarantee the correct termination of the algorithm, when the queue L is found empty by all the processors, it is necessary to compute again the surplus of all the nodes (check phase) and, eventually, restart the computation if the optimality conditions are not satisfied at some nodes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

- *Multiple Queues Implementation*. In this implementation, the active nodes are organized in multiple queues, that is, there is a separate queue for each processor. Each processor extracts nodes from its own queue and uses a heuristic procedure for choosing the queue into which eventually insert nodes that become active, after a δ-flow push operation. The queue chosen is the one with the minimum current value of the number of nodes already inserted into the queue. The heuristic is easy to implement and ensures a good load balancing among the processors.

  The multiple queues implementation guarantees much less contention for queue access than the case of a single queue (we reduce the probability that more processors attempt to simultaneously insert a node into the same queue).

  In this case, we note that the termination condition is detected in a slight different way from the single queue implementation. When a processor finds its queue empty, it switches in an idle state and, eventually, reawakens when a node is added to its queue. When the idle condition is reached by all processors (this situation is detected by using specific procedures, see for more details [15], a check phase is performed, in order to verify that the optimality conditions are satisfied by all the nodes.

We observe that both the proposed parallel implementations find the optimal solution in a finite number of iterations (the updating rules introduced in the algorithm guarantee that the ϵ-CS are satisfied at each iteration).

It is worth observing that the procedure used in our parallel algorithms for the current updating of price and flow vector resembles the computational scheme used in the parallel relaxation algorithms of Chajakis and Zenios [14]. However, our implementations are substantially different from the relaxation method, not only for the fact that Chajakis and Zenios have examined the case of strictly convex cost function (quadratic in the numerical experiments). We cite, for example, that Chajakis and Zenios adopted a static node allocation procedure to split the workload among the processors, whereas we use a dynamic allocation, and, consequently, any asynchronous updating operation has been re-designed (in some sense, our method is much more "chaotic").

5. Computational experiments

It is well known that the theoretical and the practical performance of the ϵ-relaxation method can be improved by using the ϵ-scaling technique, which was first introduced for the linear minimum cost flow problem in [16] and [17].

The key idea of the ϵ-scaling technique is to apply the ϵ-relaxation method several times, starting with a large value of ϵ and reduce it up to a final value corresponding to the desiderable degree of solution accuracy.

Each application of the algorithm, called the scaling phase, provides good initial prices and flows for the next phase.

In our implementations, the sequence $\{\epsilon^{(k)}\}$ is defined by

$$\epsilon^{(k)} = \theta\epsilon^{(k-1)}, k = 1, 2, \ldots$$

where $\epsilon^{(0)}$ (starting ϵ value) and $\theta \in (0, 1)$ (scaling factor) are chosen by the user.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

For our testing, we have considered convex linear/quadratic problems with cost function defined as follows:

$$f_{ij}(x_{ij}) = \begin{cases} a_{ij}x_{ij} + b_{ij}x_{ij}^2 & \text{if } 0 \leq x_{ij} \leq u_{ij} \\ \infty & \text{otherwise} \end{cases}$$

and we have chosen $\theta = 0.5$ and $\epsilon^{(0)} = \max_{\forall(i,j)\in A} a_{ij} + 2b_{ij}u_{ij}$

We have considered scaled versions of the algorithm. All the issues introduced for the unscaled versions can be also used for the corresponding scaled counterparts, without loss of efficiency.

We choose to terminate the algorithms at the scaling phase $\bar{k}$ such that $\epsilon^{(\bar{k})} \leq 10^{-10}$.

The computational experiments have been carried out on two different sets of test problems, for which the percentage of arcs with quadratic costs is equal to fifty percent of the total number of arcs, and the remaining arcs have linear cost. All the problems have been generated by using the public domain NETGEN generator [18].

The first set (referred to as *medium scale* test problems) consists of twelve test problems, belonging to the suite designed by Klingman and Mote [18]. The corresponding classification number and the main characteristics are reported in Table 1.

The second set (referred to as *large scale* test problems) consists of eight larger problems, having 5,000 and 10,000 nodes, with different number of arcs (Table 2).

For all test problems the arc cost is chosen randomly, according to a uniform distribution, within the range [1, 100], and the arc capacity in the range [1, 1000].

The parallel algorithms have been implemented and tested by using an Origin 2000, a multiprocessor consisting of 4 nodes, each with a memory of 128 MB. Each node consists of two processors R10000 at 195 MHz, with a 4 MB cache memory and a hub device, which carries out duties similarly to a bus in a bus-based system. The nodes are connected by two routers.

Table 1. Medium scale test problems.

Problem	Nodes	Arcs	Sources	Sinks	Tsurplus
101	5,000	25,000	2,500	2,500	250,000
102	5,000	25,000	2,500	2,500	2,500,000
103	5,000	25,000	2,500	2,500	6,250,000
107	5,000	37,500	2,500	2,500	375,000
108	5,000	50,000	2,500	2,500	500,000
109	5,000	75,000	2,500	2,500	750,000
111	5,000	35,500	2,500	2,500	250,000
112	5,000	50,000	2,500	2,500	250,000
113	5,000	75,000	2,500	2,500	250,000
123	5,000	25,000	500	500	250,000
124	5,000	25,000	1,000	1,000	250,000
125	5,000	25,000	1,500	1,500	250,000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Table 2. Large scale test problems.

Problem	Nodes	Arcs	Sources	Sinks	Tsurplus
L1	5,000	200,000	2,500	2,500	500,000
L2	10,000	200,000	5,000	5,000	1,000,000
L3	5,000	400,000	2,500	2,500	500,000
L4	10,000	400,000	5,000	5,000	1,000,000
L5	5,000	600,000	2,500	2,500	500,000
L6	10,000	600,000	5,000	5,000	1,000,000
L7	5,000	800,000	2,500	2,500	500,000
L8	10,000	800,000	5,000	5,000	1,000,000

The main feature of this system is that the hardware allows the physical distributed memory of the system to be shared, just as in a bus-based system, but since each hub is connected to its local memory, the bandwith is proportional to the number of nodes, and so, there is no inherent limit to the number of processors that can be effectively used in the system. On the other hand, the main drawback of this parallel system is related to the access time to memory. Indeed, it is no longer uniform: it varies depending on how far away the memory being accessed is in the system. So, while two processors in each node have quick access to their local memory through their hub, accessing remote memories by additional hubs adds an extra time overhead.

The operating system used is IRIX 6.4, whereas the compiler is the f77.

The performance of the parallel implementations has been evaluated by measuring the average execution times, obtained over 5 runs, as a function of the number of processors.

Tables 3, 4 and Tables 5, 6 report the results for the single queue (*SQ* for short) and the multiple queues implementation (*MQ* for short), respectively.

Table 3. Average execution time (in secs) required by the *SQ* implementation for the medium scale test problem.

Problem	Seq	2-Proc	4-Proc	8-Proc
101	333.69	230.13	128.34	66.47
102	456.97	302.63	159.78	85.26
103	525.69	318.60	174.07	95.75
107	411.41	304.75	158.23	90.22
108	496.10	359.49	187.92	107.15
109	509.36	363.83	191.49	106.34
111	354.29	266.38	157.46	80.16
112	367.10	269.93	152.96	82.13
113	431.45	303.20	167.24	91.22
123	249.16	190.20	103.39	57.81
124	291.54	217.57	115.69	62.43
125	301.72	226.86	115.60	63.79

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Table 4. Average execution time (in secs) required by the *MQ* implementation for the medium scale test problem.

Problem	Seq	2-Proc	4-Proc	8-Proc
101	333.69	222.46	119.18	65.17
102	456.97	295.61	151.82	82.78
103	525.69	311.68	153.71	89.25
107	411.41	293.86	152.37	85.53
108	496.10	346.92	181.72	100.83
109	509.36	351.28	184.55	102.28
111	354.29	256.73	150.76	76.69
112	367.10	262.21	146.84	78.27
113	431.45	294.72	161.48	87.62
123	249.16	184.56	99.27	55.12
124	291.54	211.26	110.85	61.38
125	301.72	209.53	110.93	62.60

Table 5. Average execution time (in secs) required by the *SQ* implementation for the large scale test problem.

Problem	Seq	2-Proc	4-Proc	8-Proc
L1	585.67	385.31	196.53	117.60
L2	1743.24	1131.97	579.15	347.26
L3	793.93	508.93	254.46	155.67
L4	1862.67	1164.17	585.75	349.47
L5	998.67	608.94	310.15	183.91
L6	2314.22	1377.51	712.07	408.15
L7	1374.26	808.39	411.46	233.32
L8	3299.17	1896.07	970.34	548.95

Table 6. Average execution time (in secs) required by the *MQ* implementation for the large scale test problem.

Problem	Seq	2-Proc	4-Proc	8-Proc
L1	585.67	370.68	187.71	99.77
L2	1743.24	1089.52	544.76	290.06
L3	793.93	484.10	244.29	129.52
L4	1862.67	1089.28	564.44	299.95
L5	998.67	567.43	294.59	157.77
L6	2314.22	1285.96	670.79	343.36
L7	1374.26	750.96	392.64	199.75
L8	3299.17	1745.59	906.36	447.04

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

In order to evaluate the performance of the proposed parallel version of the ϵ-relaxation method, we have measured the speedup value computed as the average sequential execution time over the average multiple-processors execution time (see figures 1–4).

We observe that the speedup values are not proportional to the number of processors used. More specifically, the average speedup values are 1.49, 2.82, 5,02 on 2, 4 and 8 processors, respectively for the *SQ* implementation, and 1.56, 2.98, 5.55 on 2, 4 and 8 processors for the *MQ* implementation.

This numerical behaviour can be explained by different reasons: (a) during the last scaling phases, the number of nodes with surplus greater than the user-defined threshold decreases. Thus, some processors can remain in an idle state and, consequently, we have a loss of efficiency; (b) the non-uniform access to memories. This affects the performance of the method, especially when the number of processors is increased; (c) the synchronitazion overhead due to access with locking of the common data structure. It penalizes, in particular, the *SQ* implementation as confirmed by comparing the results obtained with the *MQ* implementation (see Tables 4 and 6). In this case, each processor extracts nodes from its private queue and eventually inserts nodes into another queue, chosen by using a heuristic procedure. In the *SQ* implementation the locking data access is used by all processors for the same queue.

Other interesting considerations can be drawn by observing that the performance of the parallel implementations strongly depends on the characteristics of the test problems.

In particular, we note that better speedup is achieved for the test problems with higher values of total surplus (see, problems 101, 102 and 103 of figures 1 and 2 and problems L7 and L8 of figures 3 and 4). This behavior can be explained by observing that the higher the

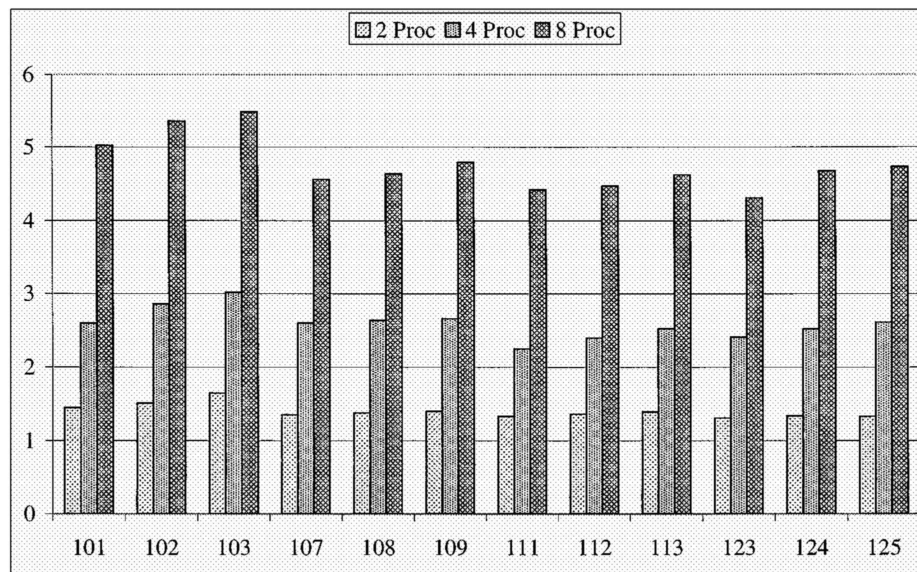

Figure 1. Speedup values of the *SQ* implementation for the medium scale problems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

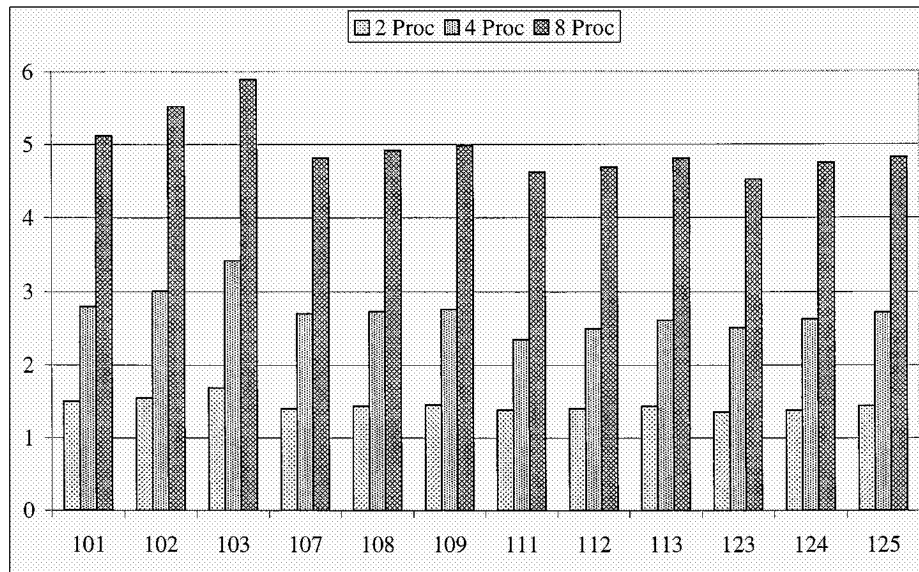

Figure 2. Speedup values of the *MQ* implementation for the medium scale problems.

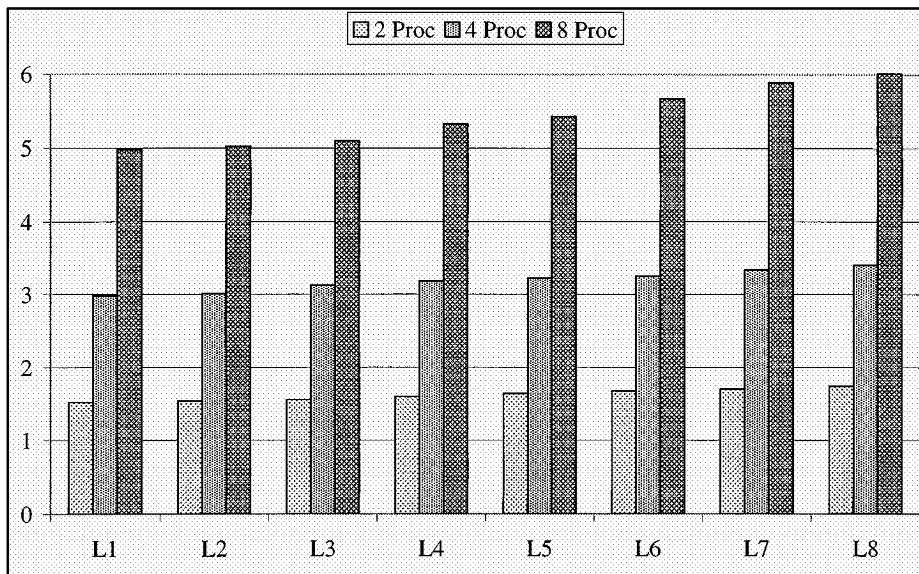

Figure 3. Speedup values of the *SQ* implementation for the large scale test problems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

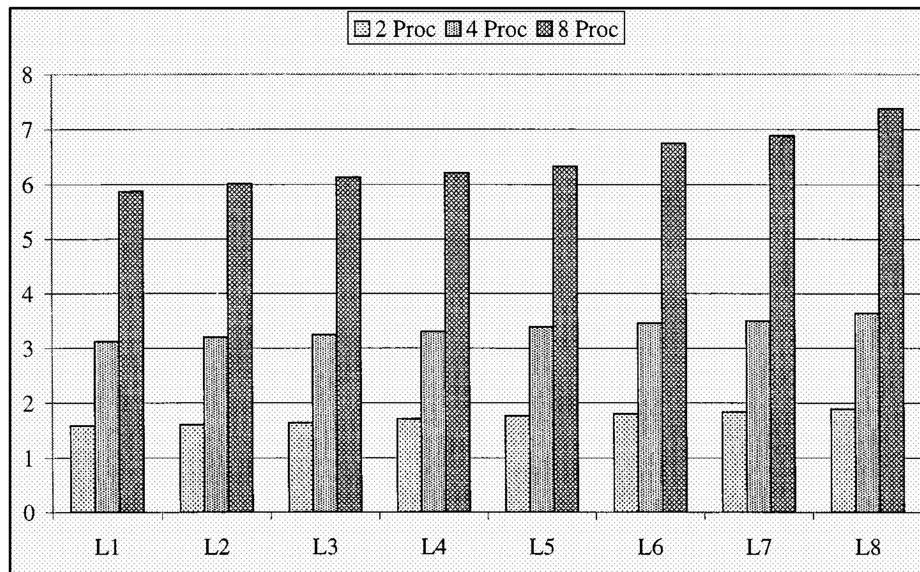

Figure 4. Speedup values of the *MQ* implementation for the large scale test problems.

surplus of a node, the larger the number of basic operations performed in order to obtain an optimal flow distribution.

Furthermore, when we increase the number of source nodes (see problems 123, 124 and 125 of figures 1 and 2) the efficiency of the parallel algorithms improves, since a higher computational workload is assigned to each processor.

Finally, we note that the best results have been obtained for the large-scale test problems (average speedup of 6.41 on 8 processors, 3.35 on 4 and 1.65 on 2). This underscores the importance of the parallelism for solving efficiently very large test problems, otherwise unmanageable, from a computational point of view, when using conventional platform.

6. Conclusions

In this paper we have presented a parallel ϵ-relaxation method for solving the separable convex cost network flow problem.

The method exploits the idea of extracting simultaneously and in a non-predetermined order several active nodes and executing asynchronously the basic operations of the sequential method.

The finite convergence of the method has been discussed with reference to a theoretical computing model which includes both the shared and the distributed memory parallel computing systems.

Two versions of the method have been implemented on a non-uniform memory access parallel computer composed of 8 processors. The two versions differ only in the number of queues used for storing the active nodes (one queue versus multiple queues).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

The results collected on a different variety of standard test problems show that, with respect to the sequential counterpart, the multiple queue implementation is the most preferable and significant speedup values are obtained for the large scale problems, which are generally hard to solve by using the sequential method.

References

1. R.R. Meyer, "Two-segment separable programming," Management Science, vol. 25, pp. 285–295, 1979.
2. J. Eckstein and M. Fukushima, "Some reformulations and applications of the alternating direction method of multiplies," in Large-scale Optimization: State of the Art, W.W. Hager, D.W. Hearn, and P.M. Pardalos (Eds.), Kluwer Scientific, 1994, pp. 119–138.
3. P.V. Kamesan and R.R. Meyer, "Multipoint methods for separable nonlinear networks," Math. Programming Study, vol. 22, pp. 185–205, 1984.
4. R.T. Rockafellar, Network Flows and Monotropic Optimization, John Wiley and Sons, 1984.
5. D.P. Bertsekas, P.A. Hosein, and P. Tseng, "Relaxation methods for network flow problems with convex arc costs," SIAM J. Control and Optimization, vol. 25, pp. 1219–1243, 1987.
6. A.V. Karzanov and S.T. McCormick, "Polynomial methods for separable convex optimization in unimodular linear spaces with applications to circulations and co-circulations in networks," UBC Report, 94-MSC-001, 1993.
7. S.T. McCormick and L. Liu, "An experimental implementation of the dual cancel and tighten algorithm for minimum cost network flow," in Network Flows and Matching, D.S. Johnson and C.S McGeoch (Eds.), American Mathematical Society DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1993, vol. 12, pp. 247–266.
8. L.C. Polymenakos, "ϵ-Relaxation and auction algorithms for the convex cost network flow problem," Ph.D. Thesis, Electrical Engineering and Computer Science Department, M.I.T., Cambridge, MA, 1995.
9. R. De Leone, R.R. Meyer, and A. Zakarian, "An ϵ-relaxation algorithm for convex network flow problems," Computer Sciences Department, University of Wisconsin, Madison, WI, Technical Report, 1995.
10. D.P. Bertsekas, L.C. Polymenakos, and P. Tseng, "An ϵ-relaxation method for convex network optimization problems," SIAM J. on Optimization, vol. 7, pp. 853–870, 1997.
11. D.P. Bertsekas and J.N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice-Hall: Englewood Cliffs, N.J., 1989.
12. X. Li and S. Zenios, "Data-level parallel solution of min-cost network flow problems using ϵ-relaxations," European Journal of Operational Research, vol. 79, pp. 474–488, 1994.
13. D.P. Bertsekas and P. Tseng, "Relaxation method for minimum cost ordinary and generalized network flow problems," Operations Research, vol. 36, pp. 93–114, 1988.
14. E. Chajakis and S. Zenios, "Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization," Parallel Computing, vol. 17, pp. 873–894, 1991.
15. P. Beraldi, F. Guerriero, and R. Musmanno, "Parallel algorithms for solving the convex minimum cost flow problem," Department of Electronics, Informatics and Systems, University of Calabria, Technical Report PARCOLAB No. 8/96, Dec., 1996.
16. A.V. Goldberg, "Efficient graph algorithms for sequential and parallel computers," Laboratory for Computer Science, M.I.T., Cambridge, MA, Technical Report TR-374, 1987.
17. A.V. Goldberg and R.E. Tarjan, "Solving minimum cost flow problems by successive approximation," Mathematics of Operations Research, vol. 15, pp. 430–466, 1990.
18. D. Klingman and J. Mote, "Computational analysis of large-scale pure networks," presented at the Joint National Meeting of ORSA/TIMS New Orleans, 1987.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com