Parallel Computing 19 (1993) 1019–1028
North-Holland

PARCO 782

Practical aspects and experiences

Asynchronous implementation of Relaxation and gradient algorithms for convex network flow problems

Didier El Baz *

LAAS du CNRS, 7, avenue du Colonel Roche, 31077 Toulouse Cedex, France

Received 4 November 1992
Revised 16 November 1992

Abstract

We consider the single commodity strictly convex network flow problem. The dual of this problem is unconstrained, differentiable, and well suited for solution via parallel iterative methods. We study the implementation of parallel asynchronous relaxation and gradient algorithms on a transputer network. We present and analyse computational experiments.

Keywords. Network flow problem; parallel implementation; asynchronous iterative methods; multitransputer systems.

1. Introduction

We consider the single commodity strictly convex network flow problem. This problem occurs in many domains: electrical networks, gas or water distribution, financial models, air traffic control. Typically, nonlinear network flow problems require intensive computations (see [29]). As a consequence the introduction of parallelism seems to be very attractive. We concentrate here on the dual problem which is unconstrained, differentiable and well suited for solution via parallel iterative methods. In recent papers (see [6,15, and 16]) we have shown that the structure of the dual problem allows the successful application of parallel asynchronous relaxation and gradient algorithms. In this paper we study the implementation of parallel asynchronous relaxation and gradient algorithms on a transputer network, we present and analyse computational experiments. The reader is also referred to [13] for preliminary results and to [8] for implementation of approximate relaxation algorithms on the Allient FX/8.

Section 2 deals with the single commodity convex network flow problem. Asynchronous relaxation and gradient algorithms are presented in Section 3. Section 4 deals with the

* *Corresponding author.* Email: elbaz@laas.fr

implementation of asynchronous iterative algorithms on a transputer network. Experimental results are given in Section 5.

2. The single commodity convex network flow problem

We consider the single commodity convex network flow problem. Let $G=(N, A)$ be a connected directed graph. N is referred to as the set of nodes, $A \subset N \times N$ is referred to as the set of arcs, and the cardinal number of N is denoted by n. Let $c_{ij}: R \to (-\infty, +\infty]$ be the cost function associated with each arc (i, j). c_{ij} is a function of the flow of the arc (i, j) which is denoted by f_{ij}. Let d be the destination node for network traffic, $b_i \geq 0$ the supply at node $i \in N-\{d\}$, and $b_d = -\sum_{i \in N-\{d\}} b_i$ the demand at d. The problem is to minimize total cost subject to a conservation of flow constraint at each node

$$\min \sum_{(i,j)\in A} c_{ij}(f_{ij}), \tag{2.1}$$

subject to

$$\sum_{(i,j)\in A} f_{ij} - \sum_{(m,i)\in A} f_{mi} = b_i, \quad \forall i \in N.$$

We assume that problem (2.1) has a feasible solution. We also make the following standing assumptions on c_{ij}:

(a) c_{ij} is strictly convex, lower semicontinuous;

(b) the conjugate convex function of c_{ij}, defined by

$$c_{ij}^*(t_{ij}) = \sup_{f_{ij}} \{t_{ij} \cdot f_{ij} - c_{ij}(f_{ij})\}, \tag{2.2}$$

is real valued, i.e. $-\infty < c_{ij}^*(t_{ij}) < +\infty$ for all real t_{ij}.

Assumption (b) implies that $\lim_{|f_{ij}| \to \infty} c_{ij}(f_{ij}) = +\infty$. Therefore the objective function of problem (2.1) has bounded level sets (see [25], Section 8). It follows that there exists an optimal solution for problem (2.1) which must be unique in view of the strict convexity assumed in (a). From the strict convexity of c_{ij}, it follows also that c_{ij}^* is continuously differentiable and its gradient denoted by $\nabla c_{ij}^*(t_{ij})$ is the unique f_{ij} attaining the supremum in (2.2) (see [25], pp. 218, 253).

A dual problem for (2.1) is given by

$$\min_{p \in R^n} q(p), \tag{2.3}$$

subject to no constraints on the vector $p = \{p_i / i \in N\}$, where q is the dual functional given by

$$q(p) = \sum_{(i,j)\in A} c_{ij}^*(p_i - p_j) - \sum_{i \in N} b_i \cdot p_i. \tag{2.4}$$

We refer to p as a price vector and its components as prices. The ith price, p_i, is a Lagrange multiplier associated with the ith conservation of flow constraint.

The duality between problems (2.1) and (2.3) is explored in great detail in [26]. The necessary and sufficient condition for optimality of a pair (f, p) is given in [25]. A feasible flow vector $f = \{f_{ij} / (i, j) \in A\}$ is optimal for (2.1) and a price vector $p = \{p_i / i \in N\}$ is optimal for (2.3) if and only if for all arcs $(i, j) \in A$,

$p_i - p_j$ is a subgradient of c_{ij} at f_{ij}.

An equivalent condition is

$$f_{ij} = \nabla c_{ij}^*(p_i - p_j), \quad \forall (i, j) \in A.$$

Any one of these equivalent relations is referred to as the complementary slackness condition (see [25], pp. 337–338 and [6]).

Existence of an optimal solution of the dual problem can be guaranteed under the following additional regular feasibility assumption (see [26], p. 360 and p. 329): there exists a feasible flow vector, $f = \{f_{ij}/(i, j) \in A\}$, such that $c'_{ij-}(f_{ij}) < +\infty$ and $c'_{ij+}(f_{ij}) > -\infty$, for all $(i, j) \in A$, where c'_{ij-}, respectively c'_{ij+}, denotes the left, respectively the right, derivative of c_{ij}. We note that the regular feasibility assumption is not overly restrictive. On the other hand the optimal solution of the dual problem is never unique since adding the same constant to all coordinates of a price vector p leaves the dual cost unaffected. We can remove this degree of freedom by constraining the price of one node. We constrain the price of the destination node p_d to be zero. Thus we consider the reduced dual optimal solution set P^* defined by

$$P^* = \left\{ p'/q(p') = \min_p q(p),\ p'_d = 0 \right\}. \tag{2.5}$$

Clearly P^* is nonempty. Consider now $\partial q/\partial p_i \mid_p$, from (2.4) it follows that

$$\left.\frac{\partial q}{\partial p_i}\right|_p = \sum_{(i,j)\in A} \nabla c_{ij}^*(p_i - p_j) - \sum_{(m,i)\in A} \nabla c_{mi}^*(p_m - p_i) - b_i. \tag{2.6}$$

3. Asynchronous iterative algorithms

Since the reduced dual problem is unconstrained and differentiable it is natural to consider algorithmic solution by a descent iterative method. A relaxation method is interesting in this respect since it admits a simple implementation. Given a price vector, p, a node i is selected and its price p_i is changed to a value $\hat{p}_i$ such that the dual cost is minimized at $\hat{p}_i$ with respect to the ith price, all others prices being kept constant (i.e. $\partial q/\partial p_i = 0$). The algorithm proceeds by relaxing the prices of all nodes in cyclic order and repeating the process. We can associate a mapping to this iterative procedure. The so-called relaxation mapping, $F: R^n \to R^n$, is defined by $F_i(p) = \hat{p}_i$, $i = 1, \ldots, n$. A gradient method admits also simple implementation. Given a price vector p all prices p_i are changed to a value $\hat{p}_i = p_i - \alpha.\ \partial q/\partial p_i \mid_p$, and this process is repeated. We define the gradient mapping $F: R^n \to R^n$, with components $F_i(p) = p_i - \alpha.\ \partial q/\partial p_i \mid_p$.

Both relaxation and gradient algorithms are well suited for parallel implementation. The prices p_i, $i = 1, \ldots, n$, can be updated concurrently by several processors. From (2.6) we conclude that we need only local information (i.e. prices of adjacent nodes) to update a price. Parallel relaxation and gradient algorithms are carried out according to a particular order and need synchronization. Since synchronous operation requires some overhead and idle time due to synchronization may be nonnegligible it is interesting to consider iterative schemes whereby computations are carried out concurrently without any order neither synchronization, namely asynchronous iterative methods. In brief, an asynchronous iterative algorithm relative to the mapping F from R^n onto itself is a sequence $\{p(k)\}$ of vectors of R^n defined as follows (see [7], Section 6.1).

We assume that there is a set of times $T = \{0, 1, 2, \ldots\}$ at which one or more components p_i of p are updated by some processor. Let T^i be the set of times at which p_i is updated, we have for $i = 1, \ldots, n$:

$$p_i(k+1) = F_i\big(p_1(\tau_1^i(k)), \ldots, p_n(\tau_n^i(k))\big), \quad \forall k \in T^i,$$
$$p_i(k+1) = p_i(k), \quad \forall k \notin T^i, \tag{3.1}$$

where F_i is the ith component of the mapping F, and for $i = 1, \ldots, n$:
the set T^i is infinite,

$$0 \leq \tau_j^i(k) \leq k, \quad j = 1, \ldots, n, \quad \forall k \in T^i,$$

if $\{k_t\}$ is a sequence of elements of T^i that tends to infinity, then $\lim_{t \to \infty} \tau_j^i(k_t) = +\infty$ for every j.

We note that the restrictions imposed on asynchronous iterative methods are very weak: no component of the iterate vector is abandoned forever and more and more recent values of the components have to be used as the computation progresses. The advantages of asynchronous iterative algorithms are implementation simplicity and computation flexibility. Since there are no synchronization overhead, neither idle time due to synchronization, one may also hope that asynchronous iterations converge faster than synchronous iterative methods. For further details about asynchronous iterative algorithms the reader is referred to [3,5,7, and 21].

Convergence of asynchronous iterative algorithms has been established for many problems (see [3–7,9,11,14–17, and 20–24]). Particular attention must be paid to the Asynchronous Convergence Theorem of Bertsekas and Tsitsiklis (see [7], p. 431). This theorem is an original and general result, it is also a powerful aid in showing convergence of asynchronous iterative algorithms.

In the particular case of network flow problems the author has shown that the structure of the dual problem allows the successful application of asynchronous relaxation methods (see [6]) and asynchronous gradient algorithms (see [16]).

4. Implementation of asynchronous iterations

We have implemented a relaxation method (R), asynchronous relaxation methods (AR), a gradient method (G), and asynchronous gradient methods (AG) on a multitransputer system. The machine consists of a network of 5 transputers T800 with some local memory on a B008 board of Inmos. Transputer on slot 0 of the B008 board is used to begin the application and to receive the results, the 4 other transputers are dedicated to the computations.

We have considered grid network flow problems. For each problem, there is only one nonzero traffic input, say $b_1 = 1$, and the arc costs are $c_{ij}(f_{ij}) = \frac{1}{4} \cdot f_{ij}^4$. A problem with 24 nodes and 37 arcs (called size 24) is shown in *Fig. 1*. We have chosen the same starting point for the different problems: the subsolution $p_i = 0$, $\forall i \in N$.

For R and AR, algorithm partitioning is made according to block red-black decomposition of the grid. The rows are coloured alternately in red and black. Sets of adjacent rows are assigned to the processors. In what follows, a task corresponds to the updating of the variables of a row. So each processor has two sets of tasks, a red one and a black one. Processors perform successively the red tasks and the black tasks.

For G and AG, sets of adjacent rows are assigned to the processors.

We have used a pipeline network of processors with bidirectional links. This topology seems naturally well suited to grid network flow problems when few processors are available.

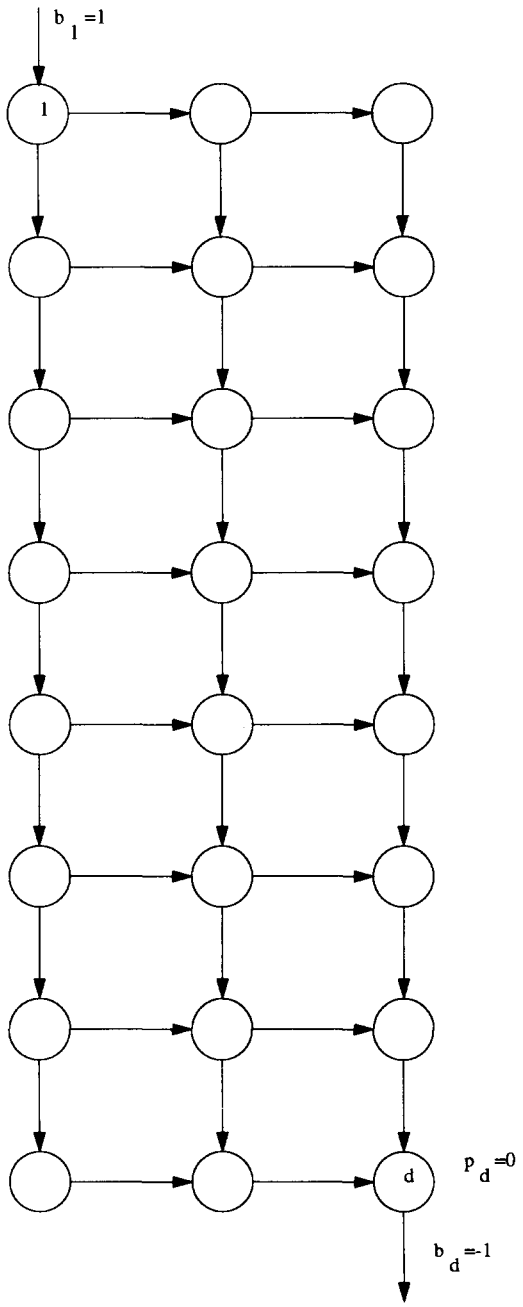

Fig. 1. Problem of size 24.

However the implementation proposed here can be extended without difficulties to other network of processors topologies.

Asynchronous iterative methods (AR and AG), are implemented as follows. Two concurrent processes run simultaneously on each transputer: a low level priority process, the so-called computation process, performs updatings and transmits the results to adjacent processors, a high level priority process, the so-called buffer process, bufferises data sent by

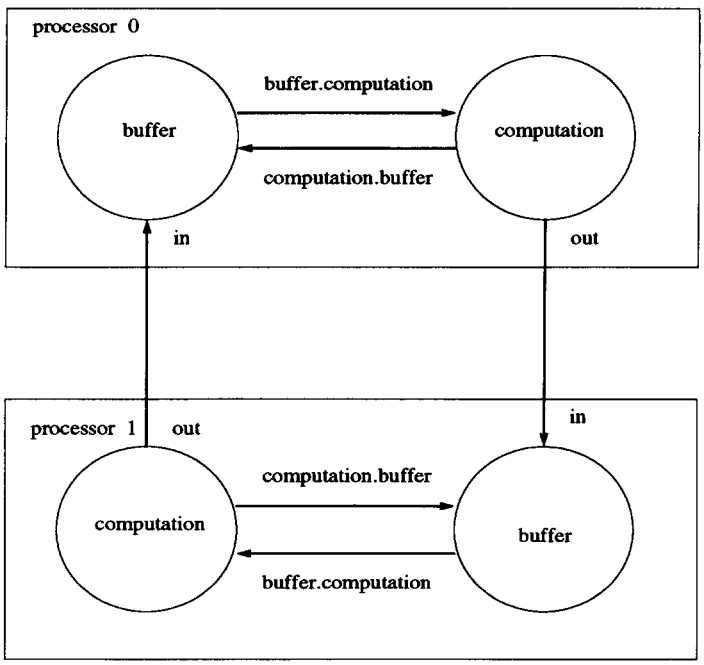

Fig. 2. Processes running on two transputers.

adjacent processors and transmits the data to the process computation when required. The use of a buffer process in each processor permits to obtain asynchronous operation. The processes running on each processor and the communication channels are shown in *Fig. 2*, in the case of a network of two transputers, in that case, the implementation of asynchronous iterative algorithms can be briefly described as follows with the Occam formalism:

```
PROC asynchronous(VAL INT i, CHAN in, out)
  CHAN buffer.computation, computation.buffer:
  PROC buffer( )
    SEQ
      ... buffer.initialisation
      WHILE loop
        ALT
          ... receipt.of.the.data.sent.by.a.neighbor
          ... service.of.computation.process
  :
  PROC computation( )
    SEQ
      ... initialisation
      ... start
      WHILE cycle
        SEQ
          ... consult.buffer.for.new.data
          ... updating
          ... transmission.of.the.updates.to.the.neighbors
```

```
  :
  PRI PAR
    buffer( )
    computation( )
:
PLACED PAR i = 0 FOR 2
  PROCESSOR i
    asynchronous(i, in, out)
```

In the process computation, the process start is waiting for a keyboard character which is transmitted via the upstream transputers. The computations are then started and the character is transmitted to the downstream transputer. The process computation iterates on the basis of the most recent data available in the buffer, in the beginning of each new updating. The updated prices are transmitted to the buffer processes of adjacent processors.

In the process buffer, the process service.of.computation.process transmits the different data received (keyboard character, results of an iteration, final results) according to the control messages sent by the process computation. The buffer process is idle while it is waiting for messages. All the cpu time is then allocated to the process computation, because the scheduler of the Transputer is designed so that the idle processes do not consume cpu time (see [19] and its references). The process buffer which has very fast elementary processes has a higher level of priority than the process computation which consumes more time. Hence the transmission of data from a process computation of a processor to a process buffer of a neighbor is very fast and the transmitter is not delayed.

The reader is also referred to [10] and [18] for different types of implementation of asynchronous iterative algorithms on transputer networks.

5. Computational experience

We present now computational experiments carried out on the transputer network. AR and AG are implemented on 2 and 4 transputers (they are denoted by AR2, AR4, AG2 and AG4, respectively). We have chosen a stepsize $\alpha = 10^{-3}$ for the gradient methods and relaxation steps are stopped when $\partial q/\partial p_i \leq 5.10^{-3}$. *Tables 2* and *5* show for the different iterative methods the solution time in seconds for which all conservation of flow constraints are satisfied with an error less than 10^{-1}. The corresponding speedups are shown in *Tables 3* and *6*. The respective numbers of updatings for the different processors are shown in *Tables 1* and *4*.

Task scheduling is made according to static mode. The same number of tasks is allocated to the different processors in the case of AR2 and AR4 except for AR4 in the special case of problem of size 30, for which the number of tasks allocated to processors 0, 1, 2, 3, is respectively 3, 2, 2, and 3. *Tables 2* and *3* show that an asynchronous implementation can speedup efficiently a relaxation method. When the nonlinear function cannot be minimized analytically with respect to each price, asynchronous relaxation algorithms lead to indeterministic load unbalancing. In the particular case of AR4, we note that processors which have two neighbors, like processors 1 and 2, do less updatings than processors which have one neighbor. A processor which has two neighbors receives more messages and has more computation to do since the receipt of a price update moves the local optimum. The good speedup of AR4 in the case of problem of size 30 can be explained by the fact that processors which have two neighbors (i.e. 1 and 2) have less tasks than processors which have one neighbor. As a comparison, when the number of tasks associated to processors 0, 1, 2, and 3 is

Table 1
Numbers of updatings for R, AR2, and AR4

Algorithm	R	AR2	AR2	AR4	AR4	AR4	AR4
processor	0	0	1	0	1	2	3
Size							
24	7786	9149	7826	38082	8698	7174	33323
30	23304	21531	22967	43215	19344	23312	38071
36	46018	35585	48485	119415	36074	42453	107772

Table 2
Times (s) of R, AR2, and AR4

Algorithm	R	AR2	AR4
Size			
24	1251.88	724.74	533.20
30	6354.23	3385.68	1867.80
36	15391.60	8441.54	5288.61

Table 3
Speedups of AR2 and AR4

Algorithm	AR2	AR4
Size		
24	1.72	2.35
30	1.88	3.40
36	1.82	2.91

Table 4
Numbers of updatings for G, AG2, and AG4

Algorithm	G	AG2	AG2	AG4	AG4	AG4	AG4
processor	0	0	1	0	1	2	3
Size							
24	15710	15640	15670	17602	14396	14408	17192
36	31153	31012	31066	32945	29362	29369	32983
48	51666	51426	51552	53734	49381	49402	53801

Table 5
Times (s) of G, AG2, and AG4

Algorithm	G	AG2	AG4
Size			
24	241.71	120.94	62.42
36	737.70	366.46	187.03
48	1652.37	817.89	414.87

Table 6
Speedups of AG2 and AG4

Algorithm	AG2	AG4
size		
24	1.99	3.87
36	2.01	3.94
48	2.02	3.98

respectively 3, 3, 2 and 2, the number of updatings is respectively 49869, 16406, 30964, and 125683, the computation time is 2098,43 s and the speedup is 3.03. An equal number of tasks for the different processors does not guarantee to obtain the best speedup. Finally we note that the delays $k - \tau_j^i(k)$ may be great for AR4.

Tables 2 and *5* point out that for medium and large scale convex network flow problems, G is faster than R and AR. We conclude that AR will not be very efficient for convex network flow problems even with massive parallelism unless there is special structure that makes each price relaxation particularly easy as in the quadratic case, where $c_{ij}(f_{ij}) = a \cdot f_{ij}^2$ (see [28]). *Tables 5* and *6* point out that an asynchronous implementation speeds up very efficiently G. Task scheduling was also made according to a static mode. The same number of tasks is allocated to the different processors in the case of AG2 and AG4. We note also that the speedups are better for AG than for AR. There is deterministic load balancing in the particular case of asynchronous gradient algorithms since we compute essentially a gradient at each iteration. The speedups of AG2 and AG4 increase with the granularity of tasks since there is deterministic load balancing and the ratio computation time over communication time increases with the granularity of tasks, it varies from 600 to 1300 for the asynchronous gradient algorithms implemented on 4 transputers. The ratio varies from 2000 to 20000 for the asynchronous relaxation methods. For a given problem a transputer which implements G stores about twice as much data in its memory as a transputer which implements AG2, so the percentage of data stored in the fast memory of the transputer is greater for AG2 than for G. This gives a first element to explain the speedups of AG2. AG2 is more efficient than AG4 because for a given problem the granularity of tasks is greater for AG2 than for AG4. The delays $k - \tau_j^i(k)$ are small for the AG methods, as an example they do not exceed 5 during the 20 first iterations of AG4 with problem of size 24.

References

[1] G. Authie, Contribution à l'optimisation de flots dans les réseaux, Un multiprocesseur expérimental pour l'étude des itérations asynchrones, Thèse de Doctorat d'Etat, UPS Toulouse, 1987.

[2] R.H. Barlow and D.J. Evans, Synchronous and asynchronous iterative parallel algorithms for linear systems, *Comput. J.* 25 (1982) 56–60.

[3] G.M. Baudet, Asynchronous iterative methods for multiprocessors, *J. Assoc. Comput. Mach.* 2 (1978) 226–244.

[4] D.P. Bertsekas, Distributed dynamic programming, *IEEE Trans. Auto. Contr.*, AC-27 (1982) 610–616.

[5] D.P. Bertsekas, Distributed asynchronous computation of fixed points, *Math. Programming* 27 (1983) 107–120.

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

[7] D.P. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation, Numerical Methods* (Prentice Hall, Englewood Cliffs, NJ, 1989).

[8] E. Chajakis and S.A. Zenios, Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization, *Parallel Comput.* 17 (1991) 873–894.

[9] D. Chazan and W. Miranker, Chaotic relaxation, *Linear Algebra Appl.* 2 (1969) 199–222.

[10] D. Conforti, R. Grandinetti, R. Musmano, M. Cannataro, G. Sezzano and D. Talia, A model of efficient asynchronous parallel algorithms on multicomputers systems, *Parallel Comput.* 18 (1992) 31–45.

[11] R. De Leone and O.L. Mangasarian, Asynchronous parallel successive overrelaxation for the symmetric linear complementarity problem, *Math. Prog.* B 42 (1988) 347–361.

[12] D. El Baz, Mise en oeuvre d'algorithmes itératifs distribués asynchrones sur un réseau de Transputers, *Lettre du Transputer et des Calculateurs Distribués* 3 (1989) 31–40.

[13] D. El Baz, A computational experience with distributed asynchronous iterative methods for convex network flow problems, *Proc. 28th IEEE Conf. on Decision and Control* (Tampa, FL, 1989) 590–591.

[14] D. El Baz, M-functions, and parallel asynchronous algorithms, *SIAM J. Numerical Anal.* 27 (1990) 136–140.

[15] D. El Baz, Asynchronous iterative algorithms for convex network flow problems, *Proc. European Control Conf.* (Grenoble, France, 1991) 2397–2402.

[16] D. El Baz, Distributed asynchronous gradient algorithms for convex network flow problems, to appear in *Proc. 31 st IEEE Conf. on Decision and Control* (Tucson, AZ, 1992) 1638–1642.
[17] M.N. El Tarazi, Some convergence results for asynchronous algorithms, *Numerisch Math.* 39 (1982) 325–340.
[18] L. Giraud and P. Spiteri, Résolution parallèle de problèmes aux limites non linéaires, *MMAN* 25 (1991) 579–606.
[19] C. Jesshope, Parallel processing, the transputer and the future, *Microprocessors and Microsystems* 13 (1989) 33–37.
[20] S. Li and T. Basar, Asymptotic agreement and convergence of asynchronous stochastic algorithms, *IEEE Trans. Auto. Contr.* AC-32 (1987) 612–618.
[21] J.C. Miellou, Algorithmes de relaxation chaotique à retards, *RAIRO* R1 (1975) 55–82.
[22] J.C. Miellou, Itérations chaotiques à retards, étude de la convergence dans le cas d'espaces partiellement ordonnés, *C.R.A.S. Paris* 280 (1975) 233–236.
[23] J.C. Miellou, Asynchronous iterations and order intervals, in: M. Cosnard ed., *Parallel Algorithms and Architectures* (North-Holland, Amsterdam, 1986) 85–96.
[24] J.C. Miellou and P. Spiteri, Un critère de convergence pour des méthodes générales de point fixe, *R.A.I.R.O. MMAN* 19 (1985) 645–669.
[25] R.T. Rockafellar, *Convex Analysis* (Princeton University Press, Princeton, NJ, 1970).
[26] R.T. Rockafellar, *Network Flows and Monotropic Optimization* (Wiley, New York, 1984).
[27] P. Tseng, D.P. Bertsekas and J.N. Tsitsiklis, Partially asynchronous parallel algorithms for network flow and other problems, *SIAM J. Control and Optimization* 28 (1990) 678–710.
[28] S. Zenios and R. Lasken, The Connection Machines CM-1 and CM-2: Solving nonlinear network problems, *Proc. Internat. Conf. on Supercomputing* (St Malo, France, 1988) 648–658.
[29] S. Zenios and J. Mulvey, A distributed algorithm for convex network optimization problems, *Parallel Comput.* 6 (1988) 45–56.