Journal of Computing and Information Technology - CIT 1, 1993, 2, 99-110

Parallel Iterative Algorithms for Solving Path Problems

Robert Manger

Department of Mathematics, University of Zagreb,

Path problems are a family of optimization and enumeration problems involving the determination of paths in directed graphs. In this paper few parallel iterative algorithms for solving path problems are developed. More precisely, the considered algorithms are parallelized counterparts of the classical iterative methods, such as Jacobi or Gauss-Seidel. The underlying model of computation is a tightly coupled multiprocessor. It is shown that the algorithms obtain the required solution after a finite number of iterations. For each algorithm, the computational complexity of a single iteration is estimated, and an upper bound for the number of iterations is established. On the basis of these results the algorithms are compared.

Introduction

Path problems are a family of related problems involving the determination of paths in a directed graph. The best-known example is the shortest path problem, stated as follows: in a graph whose arcs are given lengths determine a shortest path between two nodes (the path length is obtained by summing the corresponding arc lengths). A similar problem is to find a most reliable path between two nodes (where arc reliabilities are given, and the path reliability is defined as the product of the arc reliabilities). In addition to these optimization problems, some enumeration problems are also encountered, e.g. listing all paths from one node to another. A number of additional examples can be found in (Carre 1979, Manger 1990, Rote 1990).

There are many ways how to solve path problems. The "algebraic" approach tries to find a general formulation for the whole family of problems, by introducing a suitable abstract algebraic structure. Solving a particular problem is then reduced to computing in an appropriate concrete instance of the structure. Path problems are generally solved by algorithms that are applicable to any instance of the structure.

Few variants of the algebraic approach have been proposed (Carre 1979, Kung et al. 1987, Rote 1990, Zimmermann 1981), which differ in their definition of the structure. A good tradeoff between elegance and generality has been obtained in (Carre 1979), by introducing a structure whose instances are called "path algebras". The theory from (Carre 1979) relies heavily on establishing an analogy between path problems and standard linear algebra. Consequently, algorithms for solving path problems have been derived as counterparts of the well known methods for solving linear systems.

The aim of this paper is to develop general parallel iterative algorithms for solving path problems. In fact, the paper will deal with parallel versions of certain general sequential algorithms from (Carre 1979), which are in turn counterparts of classical iterative methods, such as Jacobi or Gauss-Seidel. As the model of parallel computing, a tightly coupled multiprocessor will be used (Quinn 1987). For each of the described algorithms, it will be shown that the algorithm obtains the required solution after a finite number of iterations. Also, the computational complexity of a single iteration will be estimated, and an upper bound for the number of iterations will be established.

The paper is organized as follows. First, some preliminaries and definitions are listed, regarding path algebras and parallel computing. Then comes an explanation how path algebras are used to formulate and solve path problems. The next section describes an obvious parallelization of the Jacobi method. It is, however, more difficult to parallelize the Gauss-Seidel method, which seems to be inherently sequential. Yet, in the farther three sections three algorithms are developed, which can be regarded as parallel versions of the Gauss-Seidel method. The final section gives some concluding remarks.

Definitions and Preliminaries

In the first part of this section we list some definitions and results from (Carre 1979). We start by defining a *path algebra*, as a set P equipped with two binary operations $\vee$ and $\circ$ which have the following properties.

- The operation $\vee$ is idempotent, commutative, and associative.
- The operation $\circ$ is associative, left-distributive over $\vee$, and right-distributive over $\vee$.
- There exist a zero element $\phi \in P$ and a unit element $e \in P$ such that for all $a \in P$: $\phi \vee a = a, \phi \circ a = \phi = a \circ \phi, e \circ a = a = a \circ e$.

Two concrete examples of the structure will be given in the next section. Additional examples can be found in (Carre 1979, Manger 1990, Rote 1990). For $a, b \in P$, the elements $a \vee b$ and $a \circ b$ are respectively called the *join* and the *product* of a and b.

We further consider matrices and vectors over a path algebra P. Let $M_n(P)$ denote the set of all $n \times n$ matrices whose entries belong to P. Similarly, let $V_n(P)$ be the set of all vectors of length n whose elements belong to P. We define the join and the product for matrices and vectors, by analogy with the sum and the product in ordinary linear algebra. For instance, given two matrices $A, B \in M_n(P)$, $A = [a_{ij}]$, $B = [b_{ij}]$, we put $A \vee B = [a_{ij} \vee b_{ij}]$, $A \circ B = [\bigvee_{k=1}^{n} a_{ik} \circ b_{kj}]$. It is easy to check that $M_n(P)$ with these operations is itself a path algebra. The zero element of $M_n(P)$ is the matrix Φ whose all entries are ϕ. The unit element of $M_n(P)$ is the matrix E whose diagonal entries are e and all other entries are ϕ. We also introduce the zero vector of $V_n(P)$: it is the vector whose all elements are ϕ.

Next, let us consider an ordering of a path algebra P. Let $\preceq$ denote a binary relation on P defined as follows. For $a, b \in P$: $a \preceq b$ if $a \vee b = b$. It is easy to show that $\preceq$ is indeed an ordering of P, and that the zero element ϕ is the least element of P with respect to $\preceq$. Also, for all $a, b \in P$: $a \vee b \succeq a$ and $a \vee b \succeq b$. Finally, both operations $\vee$ and $\circ$ are isotone for $\preceq$. The ordering $\preceq$ can be extended to $M_n(P)$ and $V_n(P)$.

Now we list some conventions that must be taken into account when interpreting expressions over a path algebra P. If the order of operations is not explicitly regulated by parentheses, then $\circ$ takes the precedence over $\vee$. Compound joins of the form $\bigvee_{i \in S} a_i$ are assumed to be ϕ if the index set S is empty.

Further, we introduce the class of stable matrices over a path algebra P. The powers of a matrix $A \in M_n(P)$ are defined by: $A^0 = E$, $A^1 = A$, $A^2 = A \circ A, \ldots, A^k = A^{k-1} \circ A$ $(k = 2, 3, \ldots)$. The matrix A is said to be *stable* if for some non-negative integer q, $\bigvee_{k=0}^{q} A^k = \bigvee_{k=0}^{q+1} A^k$. The least q with this property is called the *stability index* of A. The join $A^* = \bigvee_{k=0}^{q} A^k$ is called the *strong closure* of A, and the similar expression $\widehat{A} = A^* \circ A = A \circ A^* = \bigvee_{k=1}^{q+1} A^k$ is called the *weak closure* of A. The matrix A is said to be nilpotent if for some positive integer q, $A^q = \Phi$. Obviously, a nilpotent matrix is always stable.

Finally, let us consider a vector equation over a path algebra P of the following form:

$$\mathbf{y} = A \circ \mathbf{y} \vee \mathbf{b}. \qquad (1)$$

Here, $\mathbf{y} \in V_n(P)$ is the unknown, while $A \in M_n(P)$ and $\mathbf{b} \in V_n(P)$ are specified, A being stable. It can easily be checked that the equation (1) has a *least* solution (with respect to $\preceq$), which can be expressed as $\mathbf{y} = A^* \circ \mathbf{b}$. Moreover, if A is nilpotent, then $\mathbf{y} = A^* \circ \mathbf{b}$ is a *unique* solution of (1).

In the remaining part of this section we explain our model of parallel computing. We assume that a *tightly coupled multiprocessor* (Quinn 1987) stands at our disposal, i.e. a computer built of many processors sharing a common memory. The processors are working asynchronously. Each processor can perform any computational operation with data of any type. In order to do so, a processor must first read the operands from the memory, then compute the result, and finally write the result back into the memory.

Concurrent reads and writes of the same data item are allowed. Possible conflicts are resolved through the *serialization principle* (Gottlieb et al. 1983), which states that the effect of conflicting reads and writes is the same as if they occured in some (unspecified) sequential order. Note that the serialization principle introduces a dose of non-determinism in our model.

In order to enable comparison amongst various methods for computing in a path algebra P, we introduce the concept of *computational complexity*. This is the performance time of a given algorithm, under the assumption that one computational operation with elements of P (i.e. $\vee$, $\circ$, equality test) takes one unit of time and all other operations take no time. Also, it is assumed that all parallel **for** loops are optimally scheduled (so that the required performance time of the algorithm is minimised). Only non-preemptive schedules (Quinn 1987) are valid.

Solving Path Problems

It has been demonstrated (Carre 1979, Manger 1990, Rote 1990) that various path problems, posed on a directed graph G with n nodes, can be reduced to computing the strong or weak closure of a stable matrix $A \in M_n(P)$ over a suitable path algebra P. We are going to illustrate this idea by two examples.

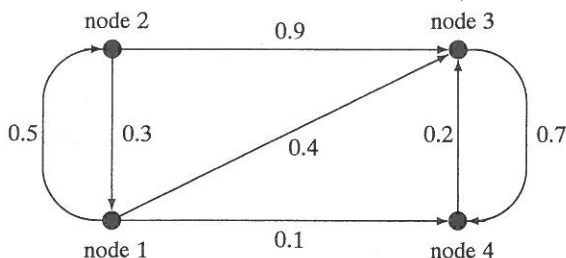

Figure 1: A directed graph G with arc labels given

Let us consider the graph G in Figure 1. Let us interpret the arc labels as "lengths". Then we may want to solve the *shortest distance problem*, i.e. we may want to determine the length of a shortest path connecting any pair of nodes. In this case, the corresponding matrix A and the resulting strong and weak closures A^* and $\widehat{A}$, respectively, are given by

$$A = \begin{bmatrix} \infty & 0.5 & 0.4 & 0.1 \\ 0.3 & \infty & 0.9 & \infty \\ \infty & \infty & \infty & 0.7 \\ \infty & \infty & 0.2 & \infty \end{bmatrix}$$
$$A^* = \begin{bmatrix} 0 & 0.5 & 0.3 & 0.1 \\ 0.3 & 0 & 0.6 & 0.4 \\ \infty & \infty & 0 & 0.7 \\ \infty & \infty & 0.2 & 0 \end{bmatrix}$$
$$\widehat{A} = \begin{bmatrix} 0.8 & 0.5 & 0.3 & 0.1 \\ 0.3 & 0.8 & 0.6 & 0.4 \\ \infty & \infty & 0.9 & 0.7 \\ \infty & \infty & 0.2 & 0.9 \end{bmatrix}$$

The (i, j)-th entry of A is the length of the arc connecting the i-th and the j-th node in G. The (i, j)-th entry of both A^* and $\widehat{A}$ is the length of a shortest path between the i-th and the j-th node. The difference between A^* and $\widehat{A}$ is as follows. A^* assumes the existence of a trivial zero-length path (having no arcs), which connects any node to itself. $\widehat{A}$, on the other hand, takes into account only non-trivial paths. In all matrices above, symbols ∞ denote that the corresponding arcs/paths do not exist. P is here the set of real numbers extended with ∞; the role of the operation $\vee$ takes the standard minimum, and $\circ$ is the conventional summation. The ordering $\preceq$ is in fact the standard $\geq$.

Let us again consider the same graph G in Figure 1, but now interpret the arc labels as "reliabilities". Suppose that we want to solve the *maximum reliability problem*, i.e. we want to determine the reliability of a most reliable path connecting any pair of nodes. Then the suitable matrix A and the corresponding closures A^* and $\widehat{A}$ are given by

$$A = \begin{bmatrix} 0 & 0.5 & 0.4 & 0.1 \\ 0.3 & 0 & 0.9 & 0 \\ 0 & 0 & 0 & 0.7 \\ 0 & 0 & 0.2 & 0 \end{bmatrix}$$

$$A^* = \begin{bmatrix} 1 & 0.5 & 0.45 & 0.315 \\ 0.3 & 1 & 0.9 & 0.63 \\ 0 & 0 & 1 & 0.7 \\ 0 & 0 & 0.2 & 1 \end{bmatrix}$$

$$\widehat{A} = \begin{bmatrix} 0.15 & 0.5 & 0.45 & 0.315 \\ 0.3 & 0.15 & 0.9 & 0.63 \\ 0 & 0 & 0.14 & 0.7 \\ 0 & 0 & 0.2 & 0.14 \end{bmatrix}$$

The meaning of any matrix entry is similar as before. P is now the set of real numbers falling into the range $[0,1]$, with the standard maximum serving as $\vee$ and the standard multiplication as $\circ$. The ordering $\preceq$ now conforms with $\leq$.

In order to solve path problems, we consider algorithms that evaluate (some or all) elements of A^* or $\widehat{A}$, where $A \in M_n(P)$ is a stable matrix, and P is an unspecified (usually arbitrary) path algebra. Thus our algorithms in principle solve an "abstract" path problem. Direct evaluation of the whole closure matrix according to its definition would be tedious. More efficient algorithms are based on the idea that the needed entries can be obtained indirectly, by solving suitably chosen equations. For instance, the equation (1) with an appropriate vector $\mathbf{b}$ can be used to compute any column of A^* and $\widehat{A}$ respectively.

The algorithms for solving path problems considered in this paper are in fact algorithms that compute the least solution of the equation (1) in an arbitrary path algebra.

The Jacobi Algorithm

The equation (1) has the form that suggests iterating as a suitable method of solving. An initial vector $\mathbf{y}^{(0)} \in V_n(P)$ is chosen, and the sequence of vectors

$$\mathbf{y}^{(k)} = A \circ \mathbf{y}^{(k-1)} \vee \mathbf{b} \quad (k = 1, 2, \ldots) \qquad (2)$$

is computed. We hope that after a finite number of iterations the least solution of (1) will be reached; i.e. for some k_0 and any $k \geq k_0$, $\mathbf{y}^{(k)}$ will be equal to $A^* \circ \mathbf{b}$. The elements of $\mathbf{y}^{(k)}$ do not depend one onto another, so they can be computed in parallel. Using these ideas, the following simple Jacobi-like algorithm is obtained.

Algorithm 1

(* *P is any path algebra. m processors are available. Input: a stable matrix $A = [a_{ij}] \in M_n(P)$, and two vectors $\mathbf{b} = [b_i]$ and $\mathbf{y}^{(0)} = [y_i^{(0)}]$ both $\in V_n(P)$.* *)

$k := 0$;
repeat
  $s :=$ *true* (* *s is a Boolean variable* *) ;
  $k := k + 1$;
  for all $i \in \{1, 2, \ldots, n\}$ **do in parallel**
    begin
      $y_i^{(k)} := (\bigvee_{j=1}^{n} a_{ij} \circ y_j^{(k-1)}) \vee b_i$
        (* *sequentially* *) ;
      if $y_i^{(k)} \neq y_i^{(k-1)}$ **then**
        $s :=$ *false*
    end
until $s =$ *true*

(* *Output: the vector* $[y_i^{(k)}]$. *)

Note that Algorithm 1 includes the sequential Jacobi method (Carre 1979) as a special case (when $m = 1$). The sequence of vectors $\mathbf{y}^{(k)}$ $(k = 1, 2, \ldots)$ depends on the input data, but not on m. Therefore the number of iterations in the parallel case is the same as in the sequential case, but the time needed for one iteration is hopefully shorter. The properties of Algorithm 1 are summarized in the next theorem.

Theorem 1 *Let the vector $\mathbf{y}^{(0)}$ in Algorithm 1 be chosen so that $\mathbf{y}^{(0)} \preceq A^* \circ \mathbf{b}$ (for instance $\mathbf{y}^{(0)}$ = zero vector or $\mathbf{y}^{(0)} = \mathbf{b}$). Then Algorithm 1 correctly computes the least solution of the equation (1). If q is the stability index of A, then the number of iterations (i.e. the number of steps of the* **repeat** ... **until** *loop) is not greater than $q + 2$. Specially for $\mathbf{y}^{(0)} = \mathbf{b}$, the number of iterations is not greater than $q + 1$. The computational complexity of a single iteration is $\lceil n/m \rceil (2n + 1)$.*

Proof. By iterating (2) we get

$$\mathbf{y}^{(k)} = A^k \circ \mathbf{y}^{(0)} \vee (E \vee A \vee A^2 \vee \ldots \vee A^{k-1}) \circ \mathbf{b}. \qquad (3)$$

Hence, by the definition of the stability index,

$$\mathbf{y}^{(k)} = A^k \circ \mathbf{y}^{(0)} \vee A^* \circ \mathbf{b} \succeq A^* \circ \mathbf{b} \quad \text{if } k \geq q + 1. \qquad (4)$$

The imposed condition $\mathbf{y}^{(0)} \preceq A^* \circ \mathbf{b}$ implies that

$$\begin{aligned} A^k \circ \mathbf{y}^{(0)} &\preceq A^k \circ A^* \circ \mathbf{b} \\ &= (A^k \vee A^{k+1} \vee A^{k+2} \vee \ldots) \circ \mathbf{b} \\ &\preceq A^* \circ \mathbf{b}. \end{aligned}$$

Combining the inequality above and (3) we obtain

$$\mathbf{y}^{(k)} \preceq A^*\circ\mathbf{b} \vee (E\vee A\vee A^2\vee\ldots\vee A^{k-1})\circ\mathbf{b} \preceq A^*\circ\mathbf{b} \vee A^*\circ\mathbf{b} = A^*\circ\mathbf{b}. \qquad (5)$$

From (4) and (5) if follows that $\mathbf{y}^{(k)} = A^*\circ\mathbf{b}$ if $k \geq q+1$. Hence the algorithm stops after at most $q+2$ iterations, and it correctly computes the least solution of the equation (1).

The sharper bound on the number of iterations for $\mathbf{y}^{(0)} = \mathbf{b}$ is obtained from (3), by noting that in this case

$$\mathbf{y}^{(k)} = (E \vee A \vee A^2 \vee \ldots \vee A^{k-1} \vee A^k) \circ \mathbf{b} = A^* \circ \mathbf{b} \quad \text{if } k \geq q.$$

In accordance with the definition of computational complexity, the time needed to compute any $y_i^{(k)}$ is exactly $(2n+1)$. The optimal schedule obviously assigns $\lceil n/m \rceil$ steps to one processor. Hence the computational complexity of one iteration is $\lceil n/m \rceil (2n+1)$. □

Theorem 1 establishes a relationship between the total execution time of Algorithm 1 and the stability index q of the matrix A. So an interesting question arises: how big could q really be? According to (Carre 1979), q can be visualized as the maximum order (number of arcs) of a path which still contributes to the solution of the corresponding path problem. For the majority of problems (including shortest distance and maximum reliability), only those paths that have no cycles are relevant. Thus q usually can not be greater than $n-1$. Moreover, if our graph has some special structure, q can even be considerably smaller than $n-1$. This is for instance true for layered graphs, where the order of a path is proportional to the number of layers, not the total number of nodes.

A Non-Deterministic Gauss-Seidel Algorithm

Algorithm 1 computes an element $y_i^{(k)}$ of the k-th vector by using only the elements $y_j^{(k-1)}$ of the $(k-1)$-st vector. Suppose that at the moment of evaluating $y_i^{(k)}$ some $y_j^{(k)}$ are already available. Then we could substitute those $y_j^{(k)}$ in the expression for $y_i^{(k)}$, in place of the corresponding $y_j^{(k-1)}$. This substitution is intuitively acceptable, for we hope that in this way we would reduce the number of iterations required.

By Gauss-Seidel algorithms we mean modifications of Algorithm 1, where $y_i^{(k)}$ depends on some $y_j^{(k)}$. In this section we consider the simplest of such modifications, which is based on using a single vector $\mathbf{y}$ in place of $\mathbf{y}^{(k)}$ $(k = 0, 1, 2, \ldots)$.

Algorithm 2
(* *P is any path algebra. m processors are available. Input: a stable matrix $A = [a_{ij}] \in M_n(P)$, a vector $\mathbf{b} = [b_i] \in V_n(P)$, and another vector $\mathbf{y} = [y_i] \in V_n(P)$ —i.e. its initial value.* *)
repeat
  $s :=$ *true* (* *s is a Boolean variable* *) ;
  for all $i \in \{1, 2, \ldots, n\}$ **do in parallel**
    begin
      $\bar{y}_i := (\bigvee_{j=1}^{n} a_{ij}\circ y_j) \vee b_i$
        (* *sequentially* *) ;
      if $\bar{y}_i \neq y_i$ **then**
        begin
          $y_i := \bar{y}_i$;
          $s :=$ *false*
        end
    end
until $s =$ *true*
(* *Output: the final value of the vector* $[y_i]$. *)

The values y_i, computed by Algorithm 2 in a particular iteration, are not uniquely determined. They depend, for instance, on the schedule assigning the steps of the parallel **for** loop to the available processors. But in spite of this nondeterminism, the correct result will be obtained, as guaranteed by the following theorem.

Theorem 2 *Let the chosen initial value of the vector $\mathbf{y}$ in Algorithm 2 be the zero vector or $\mathbf{b}$. Then Algorithm 2 correctly computes the least solution of the equation (1). Also, the number of iterations required by Algorithm 2 is not greater than the number required by Algorithm 1 (for the same input data). The computational complexity of a single iteration in Algorithm 2 is $\lceil n/m \rceil (2n+1)$.*

Proof. The computational complexity of one iteration is estimated analogously as in the proof of Theorem 1.

In order to prove the remaining claims, it is necessary to compare the performance of Algorithms 2 and 1. We consider one execution of Algorithm 2 for the given input data using the given number of processors. Also, we consider the execution of Algorithm 1 for the same input data (the number of processors is irrelevant). We introduce the following notation for $k = 0, 1, 2, \ldots$:

$\mathbf{y}^{(k)} = [y_i^{(k)}]$ ... the vector obtained in the k-th iteration of Algorithm 1,

$\widetilde{\mathbf{y}}^{(k)} = [\widetilde{y}_i^{(k)}]$ ... the value of the vector $\mathbf{y}$ in Algorithm 2, immediately after the k-th iteration.

It will be shown that

$$\mathbf{y}^{(k)} \preceq \widetilde{\mathbf{y}}^{(k)} \preceq A^* \circ \mathbf{b} \quad (k = 0, 1, 2, \ldots).$$

Theorem 1 guarantees the existence of an integer k_0, such that $\mathbf{y}^{(k)} = A^* \circ \mathbf{b}$ for $k \geq k_0$. Supposing that the inequality above is valid, it follows that for $k \geq k_0$ also $\widetilde{\mathbf{y}}^{(k)} = A^* \circ \mathbf{b}$. Hence Algorithm 2 terminates, and obtains the correct least solution of the equation (1). Also, the number of iterations is not greater than for Algorithm 1.

Let us write $A^* \circ \mathbf{b} = [(A^* \circ \mathbf{b})_i]$. It is left to be shown that the inequality

$$y_i^{(k)} \preceq \widetilde{y}_i^{(k)} \preceq (A^* \circ \mathbf{b})_i \tag{6}$$

indeed holds for all $i = 1, 2, \ldots, n$, $k = 0, 1, \ldots$ The proof is carried out by double mathematical induction: on k and on the fictious time instant when Algorithm 2 updates y_i (according to the serialization principle). The induction on k is the "outer" induction, and the induction on time is the "inner" (nested) induction.

The outer induction basis: it is obvious that

$$y_i^{(0)} = \widetilde{y}_i^{(0)} \preceq b_i \preceq (A^* \circ \mathbf{b})_i \ (i=1, 2, \ldots, n),$$

since in both algorithms we use the same initial vector which is equal to the zero vector or **b**.

The outer induction hypothesis: for some value of k,

$$y_i^{(k-1)} \preceq \widetilde{y}_i^{(k-1)} \preceq (A^* \circ \mathbf{b})_i \ \ (i=1, 2, \ldots, n).$$

The outer induction step is proved by the inner induction. Suppose that $t_1^{(k)} < t_2^{(k)} < \cdots < t_{\nu(k)}^{(k)}$ $(1 \leq \nu(k) \leq n)$ are all distinct instants when the updatings of variables y_i in the k-th iteration take place.

The inner induction basis: let y_i be any variable being updated at the instant $t_1^{(k)}$. The processor performing the updating must previously compute the new value for y_i which depends on all y_j. Prior to any computational operation, the processor must fetch the corresponding operands. Hence the values of all y_j are read before $t_1^{(k)}$, so they have not yet been updated. Consequently,

$$\widetilde{y}_i^{(k)} = \Big(\bigvee_{j=1}^{n} a_{ij} \circ \widetilde{y}_j^{(k-1)}\Big) \vee b_i. \tag{7}$$

Combining the outer induction hypothesis with (7) we obtain

$$\begin{aligned} \widetilde{y}_i^{(k)} &\succeq \Big(\bigvee_{j=1}^{n} a_{ij} \circ y_j^{(k-1)}\Big) \vee b_i \\ &= \ldots \text{ Algorithm 1 } \ldots = y_i^{(k)}, \end{aligned} \tag{8}$$

and also

$$\begin{aligned} \widetilde{y}_i^{(k)} &\preceq \Big(\bigvee_{j=1}^{n} a_{ij} \circ (A^* \circ \mathbf{b})_j\Big) \vee b_i \\ &= ((A \circ A^* \vee E) \circ \mathbf{b})_i = (A^* \circ \mathbf{b})_i. \end{aligned} \tag{9}$$

From (8) and (9) it follows that the required inequality (6) is valid for all i such that y_i is updated at the instant $t_1^{(k)}$.

The inner induction hypothesis: for all i such that y_i is updated before $t_r^{(k)}$,

$$y_i^{(k)} \preceq \widetilde{y}_i^{(k)} \preceq (A^* \circ \mathbf{b})_i.$$

The inner induction step: let y_i be any variable which is updated at the instant $t_r^{(k)}$. The processor performing the updating must previously compute the new value for y_i which depends on all y_j. Prior to any computational operation, the processor must fetch the corresponding operands. Some of the fetched y_j have already been updated in the k-th iteration, and some have not. Let S_1 be the set of indices j such that y_j has been updated, and let S_2 be the

set of j such that y_j has not been updated. Then

$$\widetilde{y}_i^{(k)} = \Big(\bigvee_{j\in S_1} a_{ij}\circ\widetilde{y}_j^{(k)}\Big)\vee\Big(\bigvee_{j\in S_2} a_{ij}\circ\widetilde{y}_j^{(k-1)}\Big)\vee b_i. \quad (10)$$

Since all y_j are read before $t_r^{(k)}$, the inner induction hypothesis can be applied to $\widetilde{y}_j^{(k)}$ (for $j\in S_1$). The outer induction hypothesis can be applied to $\widetilde{y}_j^{(k-1)}$ (for $j\in S_2$). Combining the inner and outer induction hypothesis with (10) we obtain

$$\widetilde{y}_i^{(k)} \succeq \Big(\bigvee_{j\in S_1} a_{ij}\circ y_j^{(k)}\Big)\vee\Big(\bigvee_{j\in S_2} a_{ij}\circ y_j^{(k-1)}\Big)\vee b_i. \quad (11)$$

It is evident from (3) that $\mathbf{y}^{(k)}\succeq\mathbf{y}^{(k-1)}$ if $\mathbf{y}^{(0)}=$ zero vector or $\mathbf{y}^{(0)}=\mathbf{b}$. Combining this inequality with (11) we get

$$\begin{aligned}\widetilde{y}_i^{(k)} &\succeq \Big(\bigvee_{j\in S_1} a_{ij}\circ y_j^{(k-1)}\Big)\vee\\ &\quad\vee\Big(\bigvee_{j\in S_2} a_{ij}\circ y_j^{(k-1)}\Big)\vee b_i\\ &= \ldots \text{Algorithm 1} \ldots = y_i^{(k)}. \quad (12)\end{aligned}$$

Also by applying the inner and outer induction hypothesis to (10) we obtain

$$\begin{aligned}\widetilde{y}_i^{(k)} &\preceq \Big(\bigvee_{j\in S_1} a_{ij}\circ(A^*\circ\mathbf{b})_j\Big)\vee\\ &\quad\vee\Big(\bigvee_{j\in S_2} a_{ij}\circ(A^*\circ\mathbf{b})_j\Big)\vee b_i\\ &= ((A\circ A^*\vee E)\circ\mathbf{b})_i=(A^*\circ\mathbf{b})_i. \quad (13)\end{aligned}$$

From (12) and (13) it follows that the required inequality (6) holds for all i such that y_i is updated at the instant $t_r^{(k)}$.

This completes the proof of the theorem. □

A Deterministic Gauss-Seidel Algorithm

If a small number of processors ($m << n$) is available, then in Algorithm 1 the same processor computes more elements $y_i^{(k)}$ of the vector $\mathbf{y}^{(k)}$. Imagine that $\mathbf{y}^{(k)}$ is split into $\leq m$ segments which are approximately equal in length. By modifying Algorithm 1 we can achieve that one processor computes exactly one of those segments, i.e. a sequence of consecutive elements $y_i^{(k)}$. When evaluating the expression for the next $y_i^{(k)}$, the processor can use all the previously computed $y_j^{(k)}$ from the same segment, and substitute them in place of the corresponding $y_j^{(k-1)}$. We obtain the following modification which as well belongs to the family of Gauss-Seidel algorithms.

Algorithm 3
(* *P is any path algebra. m processors are available. Input: a stable matrix $A=[a_{ij}]\in M_n(P)$, and two vectors* $\mathbf{b}=[b_i]$ *and* $\mathbf{y}^{(0)}=[y_i^{(0)}]$ *both* $\in V_n(P)$. *)

```
k := 0 ;
l := ⌈n/m⌉   (* length of a segment *) ;
m̄ := ⌈n/l⌉   (* number of segments ≤ m *) ;
repeat
    s := true   (* s is a Boolean variable *) ;
    k := k + 1 ;
    for all r ∈ {1, 2, ..., m̄} do in parallel
        for i := (r − 1)l + 1 to min{rl, n} do
            begin
                y_i^(k) := (⋁_{j=1}^{(r−1)l} a_ij∘y_j^(k−1))∨
                           (⋁_{j=(r−1)l+1}^{i−1} a_ij∘y_j^(k))∨
                           (⋁_{j=i}^{n} a_ij∘y_j^(k−1)) ∨ b_i
                           (* sequentially *) ;
                if y_i^(k) ≠ y_i^(k−1) then
                    s := false
            end
until s = true
```

(* *Output: the vector* $[y_i^{(k)}]$. *)

Note that Algorithm 3 is "deterministic", i.e. the computed sequence of vectors $\mathbf{y}^{(k)}$ ($k=1,2,\ldots$) is uniquely determined by the input data and by the number of processors available. The advantage in respect to Algorithm 2 is less communication with the common memory. Namely, all $y_i^{(k)}$ from the same segment of the vector $\mathbf{y}^{(k)}$ can temporarily be kept in local registers, until the whole segment is computed. Additional characteristics of Algorithm 3 are listed in the next theorem.

Theorem 3 *Let the chosen vector* $\mathbf{y}^{(0)}$ *in Algorithm 3 be the zero vector or* $\mathbf{b}$. *Then Algorithm 3 correctly computes the least solution of the equation (1). Also, the number of iterations required by Algorithm 3 is not greater*

than the number required by Algorithm 1 (for the same input data). The computational complexity of a single iteration in Algorithm 3 is $\lceil n/m \rceil (2n+1)$.

Proof. Similar as for Theorem 2. □

We stress that the sequence of vectors $\mathbf{y}^{(k)}$ $(k = 1, 2, \ldots)$ really depends on the number of processors m. For $m = 1$, Algorithm 3 becomes the usual sequential Gauss-Seidel method (Carre 1979). For $m = n$, Algorithm 3 does the same as Algorithm 1, i.e. it turns into the parallel Jacobi algorithm. With the increase of m, the number of iterations probably also increases, but the time needed for one iteration decreases.

In the remainder of this section we observe some properties of the sequential algorithm which are going to be applied in the next section. The results are summarized in the following lemma.

Lemma 1 *The sequential Gauss-Seidel method (i.e. Algorithm 3 for* $m = 1$*) is considered. Let the vector* $\mathbf{y}^{(0)}$ *be the zero vector or* $\mathbf{b}$. *Let us introduce the following notation:*

$$w_r^{(k)} := \Big(\bigvee_{j=r}^{n} a_{rj} \circ y_j^{(k)}\Big) \vee b_r \quad (r = 1, 2, \ldots, n, k = 0, 1, 2, \ldots),$$

$$z_r^{(k)} := \bigvee_{j=1}^{r-1} a_{rj} \circ y_j^{(k)} \quad (r = 1, 2, \ldots, n, k = 0, 1, 2, \ldots).$$

Then

1. $y_i^{(k)} \succeq y_i^{(k-1)} \quad (i = 1, 2, \ldots, n, k = 1, 2, \ldots),$

2. $w_r^{(k)} = w_r^{(k-1)} \vee \bigvee_{j=r}^{n} a_{rj} \circ y_j^{(k)} \quad (r = 1, 2, \ldots, n, k = 1, 2, \ldots),$

3. $z_r^{(k)} = z_r^{(k-1)} \vee \bigvee_{j=1}^{r-1} a_{rj} \circ y_j^{(k)} \quad (r = 1, 2, \ldots, n, k = 1, 2, \ldots).$

Proof. First we prove 1. The matrix A can be expressed as $A = L \vee U$ where $L \in M_n(P)$ is strictly lower triangular and $U \in M_n(P)$ is upper triangular. Consequently,

$$\mathbf{y}^{(k)} = L \circ \mathbf{y}^{(k)} \vee U \circ \mathbf{y}^{(k-1)} \vee \mathbf{b} \quad (k = 1, 2, \ldots). \quad (14)$$

The strictly lower triangular matrix L is nilpotent (i.e. $L^n = \Phi$), and therefore also stable. According to the previously cited results, (14) has a unique solution

$$\mathbf{y}^{(k)} = L^* \circ U \circ \mathbf{y}^{(k-1)} \vee L^* \circ \mathbf{b} \quad (k = 1, 2, \ldots). \quad (15)$$

By iterating (15) we obtain

$$\mathbf{y}^{(k)} = (L^* \circ U)^k \circ \mathbf{y}^{(0)} \vee (E \vee (L^* \circ U) \vee \vee (L^* \circ U)^2 \vee \ldots \vee (L^* \circ U)^{k-1}) \circ L^* \circ \mathbf{b} \quad (k = 1, 2, \ldots). \quad (16)$$

If $\mathbf{y}^{(0)} =$ zero vector, then (16) can be written as

$$\mathbf{y}^{(k)} = \mathbf{y}^{(k-1)} \vee (L^* \circ U)^{k-1} \circ L^* \circ \mathbf{b} \quad (k = 1, 2, \ldots).$$

Similarly, for $\mathbf{y}^{(0)} = \mathbf{b}$, (16) becomes

$$\mathbf{y}^{(k)} = \mathbf{y}^{(k-1)} \vee (L^* \circ U)^{k-1} \circ \circ (L \vee L^2 \vee \ldots) \circ \mathbf{b} \vee (L^* \circ U)^k \circ \mathbf{b} \quad (k = 1, 2, \ldots).$$

Hence in both cases, $\mathbf{y}^{(k)} \succeq \mathbf{y}^{(k-1)}$ $(k = 1, 2, \ldots)$.

Next we prove 2. For $1 \leq r \leq n$ and $k \geq 1$ we compute:

$$w_r^{(k-1)} \vee \bigvee_{j=r}^{n} a_{rj} \circ y_j^{(k)} = \Big(\bigvee_{j=r}^{n} a_{rj} \circ (y_j^{(k-1)} \vee y_j^{(k)})\Big) \vee b_r = \ldots \text{ claim 1 } \ldots = \Big(\bigvee_{j=r}^{n} a_{rj} \circ y_j^{(k)}\Big) \vee b_r = w_r^{(k)}.$$

Claim 3 is proved similarly. □

Another Deterministic Gauss-Seidel Algorithm

Lemma 1 reveals an extraordinary idea how to parallelize the sequential Gauss-Seidel method. Let us consider the k-th iteration of the method. Instead of trying to compute the elements $y_i^{(k)}$ in parallel, we could compute the values $w_r^{(k)}$ and $z_r^{(k)}$ in parallel, using the formulas 2 and 3 from Lemma 1. Then any $y_i^{(k)}$ can be computed in just one additional operation, as $y_i^{(k)} := w_i^{(k-1)} \vee z_i^{(k)}$. We obtain Algorithm 4, whose correctness is guaranteed by Theorem 4.

Algorithm 4
(* *P is any path algebra. m processors are available. Input: a stable matrix* $A=[a_{ij}]\in M_n(P)$, *a vector* $\mathbf{b}=[b_i]\in V_n(P)$, *and another vector* $\mathbf{y} = [y_i] \in V_n(P)$ *—i.e. its initial value.* *)

```
[w_r] := b ;
[z_r] := zero vector ;
for i := 1 to n do
   for all r ∈ {1, 2, ..., n} do in parallel
      if r ≤ i then
         w_r := w_r ∨ a_ri∘y_i
      else
         z_r := z_r ∨ a_ri∘y_i ;
repeat
   s := true  (* s is a Boolean variable *) ;
   for i := 1 to n do
      begin
         ȳ_i := w_i ∨ z_i ;
         if ȳ_i ≠ y_i then
            begin
               y_i := ȳ_i ;
               s := false ;
               for all r ∈ {1, 2, ..., n} do
                        in parallel
                  if r ≤ i then
                     w_r := w_r ∨ a_ri∘y_i
                  else
                     z_r := z_r ∨ a_ri∘y_i
            end
      end
until s = true
```
(* *Output: the final value of the vector* $[y_i]$. *)

Theorem 4 *Let the chosen initial value of the vector* $\mathbf{y}$ *in Algorithm 4 be the zero vector or* $\mathbf{b}$. *Then Algorithm 4 correctly computes the least solution of the equation (1). Also, the number of iterations required by Algorithm 4 is not greater than the number required by Algorithm 3 (for the same input data using any number of processors). The worst-case computational complexity of a single iteration in Algorithm 4 is* $2n(\lceil n/m \rceil + 1)$.

Proof. The performance of Algorithm 4 is compared with the performance of the sequential Gauss-Seidel method (i.e. Algorithm 3 for $m = 1$). We consider the execution of Algorithm 4 for the given input data using the given number of processors. Also, we consider the execution of the sequential Gauss-Seidel method for the same input data. Let y_i, w_i, z_i ($i = 1, 2, \ldots, n$) be the variables in Algorithm 4, and let $y_i^{(k)}$, $w_i^{(k)}$, $z_i^{(k)}$ ($i = 1, 2, \ldots, n$, $k = 0, 1, 2 \ldots$) be the values in the sequential Gauss-Seidel method (using the notation introduced in Lemma 1). Then the following equalities hold.

- At the beginning of the k-th iteration of Algorithm 4,
$$w_r = w_r^{(k-1)},\ z_r = z_r^{(k-1)}\ (r = 1, 2, \ldots, n).$$
- At the moment of computing y_i ($1 \le i \le n$) during the k-th iteration of Algorithm 4,
$$w_i = w_i^{(k-1)},\ z_i = z_i^{(k)} (\Rightarrow y_i = y_i^{(k)}).$$
- At the end of the k-th iteration of Algorithm 4,
$$w_r = w_r^{(k)},\ z_r = z_r^{(k)}\ (r = 1, 2, \ldots, n).$$

These equalities are easily checked by mathematical induction. Indeed, during the k-th iteration of Algorithm 4, the variables w_r and z_r ($r = 1, 2, \ldots, n$) are gradually updated, so that the final result corresponds to formulas 2 and 3 from Lemma 1. At the moment of computing y_i, the value of z_i has already been fully updated (so z_i is already equal to $z_i^{(k)}$), and the updating of w_i has not yet started (so w_i is still equal to $w_i^{(k-1)}$).

To be true, in the case when $y_i^{(k)} = y_i^{(k-1)}$ Algorithm 4 skips the updating phase depending on $y_i^{(k)}$. In spite of that, the previous equalities

hold, i.e. all values are the same as if the updatings had been performed. Indeed, in this case we may write for $w_r^{(k)}$ $(r \leq i)$

$$
\begin{aligned}
w_r^{(k)} &= \ldots \text{Lemma 1} \ldots \\
&= w_r^{(k-1)} \vee \bigvee_{j=r}^{n} a_{rj} \circ y_j^{(k)} \\
&= \left(\bigvee_{\substack{j=r \\ j \neq i}}^{n} a_{rj} \circ y_j^{(k-1)} \right) \vee b_r \vee \\
&\quad \vee \left(\bigvee_{\substack{j=r \\ j \neq i}}^{n} a_{rj} \circ y_j^{(k)} \right) \vee \\
&\quad \vee a_{ri} \circ (y_i^{(k-1)} \vee y_i^{(k)}) \\
&= \ldots \text{since the operation } \vee \text{ is idempotent} \ldots \\
&= \left(\bigvee_{\substack{j=r \\ j \neq i}}^{n} a_{rj} \circ y_j^{(k-1)} \right) \vee b_r \vee \\
&\quad \vee \left(\bigvee_{\substack{j=r \\ j \neq i}}^{n} a_{rj} \circ y_j^{(k)} \right) \vee a_{ri} \circ y_i^{(k-1)} \\
&= w_r^{(k-1)} \vee \bigvee_{\substack{j=r \\ j \neq i}}^{n} a_{rj} \circ y_j^{(k)}.
\end{aligned}
$$

A similar deduction is applicable to $z_r^{(k)}$ $(r > i)$. We conclude that Algorithm 4 does the same as the sequential Gauss-Seidel method. By Theorem 3 it follows that Algorithm 4 is correct.

To prove the claim about the number of iterations, it is necessary to compare the performance of Algorithms 4 and 3. We consider the execution of Algorithm 4 for the given input data using the given number of processors. Also, we consider the execution of Algorithm 3 for the same input data, using the number of processors which is not necessarily the same as in Algorithm 4. Our intention is to show that

$$
\begin{aligned}
&(\mathbf{y}^{(k)} \text{ in Algorithm 3}) \preceq \\
&\quad \preceq (\mathbf{y} \text{ in Algorithm 4 after the } k\text{-th iteration}) \\
&\quad \preceq A^* \circ \mathbf{b} \quad (k = 0, 1, 2, \ldots).
\end{aligned}
$$

From this point, the reasoning is similar to the proof of Theorem 2. A double mathematical induction is used. Also, claim 1 from Lemma 1 is required.

Finally, we estimate the computational complexity of one iteration. The optimal schedule for the nested **for** loop obviously assigns $\lceil n/m \rceil$ steps to one processor. Therefore the time needed to execute the whole nested loop is $2\lceil n/m \rceil$. The computing of $\bar{y}_i$ and the comparison of $\bar{y}_i$ with y_i increase this time by 2. Since the outer **for** loop is sequential, the complexity of the whole iteration could be n times greater. It is easy to check that this worst case can be achieved. □

The vector $\mathbf{y}$ computed by Algorithm 4 during the k-th iteration is uniquely determined by the input data, so it does not depend on the number of available processors. In this sense, Algorithm 4 is even more "deterministic" than Algorithm 3.

Note that among the considered parallel Gauss-Seidel algorithms, only Algorithm 4 completely follows the sequential Gauss-Seidel method (i.e. after the k-th iteration the same vector $\mathbf{y}$ is obtained as by the sequential method). That's why Algorithm 4 requires a smaller number of iterations than Algorithm 3. On the other hand, the computational complexity of one iteration in Algorithm 4 is somewhat greater than in the remaining parallel algorithms. However, for $m << n$ the difference is negligible, especially as an upper bound is used which is perhaps too pessimistic. Therefore we believe that Algorithm 4 is at least as fast as Algorithm 3, if not even faster.

Concluding Remarks

The parallelizing techniques used in this paper have also appeared in the context of standard linear algebra. Therefore our algorithms for solving path problems can be viewed as being analogous to some numerical iterative methods. The counterparts of Algorithms 1, 2, and 3 are found for instance in (Barlow and Evans 1982, Bertsekas and Tsitsiklis 1989, Konrad and Wallach 1977, Ortega and Voigt 1985, Quinn 1987). The idea used in Algorithm 4 originates from (Konrad and Wallach 1977), but our algorithm is simpler and more elegant than its counterpart, due to special properties of a path algebra. Theorems 1–4 are substantially different

from the corresponding results of numerical linear algebra, because they start from different assumptions and rely on properties of a different algebraic structure. As distinguished from numerical algorithms which are approximative, our algorithms give the exact solution after a finite number of iterations.

In addition to this paper, there are many others that also deal with parallel solution of path problems. Some of them are (Deo et al. 1980, Dey and Srimani 1989, Gayraud and Authie 1992, Kung et al. 1987, Sinha et al. 1986, Takaoka 1989). Further references can be found in (Bertsekas and Tsitsiklis 1989, McHugh 1990, Quinn 1987). However, nearly all of those other papers treat only one particular problem, usually the shortest path problem. Thanks to the used algebraic approach, our results are more general, i.e. they can be applied to a wide variety of problems, not only shortest paths. Another characteristic of the other papers is that they mostly concentrate on the "all pairs" variant of the considered problem, and consequently they use algorithms which can be described as counterparts of Gaussian elimination. This paper studies the "single destination" (or "single source") variant, and it applies iteration. To be precise, there are few works that tackle to some extent parallel iterative algorithms for the shortest path problem, e.g. (Deo et al. 1980, Bertsekas and Tsitsiklis 1989). But still, none of them, for instance, employs the same idea as our Algorithm 4 to parallelize the Gauss-Seidel type of iteration.

Our future plan is to implement the described algorithms on the 4-processor machine HP Apollo DN10000. The main purpose of this implementation would be to experimentally compare and rank Algorithms 2, 3 and 4. We do not expect any substantial difficulties. Namely, the only form of parallelism encountered are relatively simple parallel **for** loops, which can be scheduled optimally or nearly optimally by dynamic scheduling (Manger 1993). Serializable concurrent access to the common memory can be emulated by the use of semaphores (Quinn 1987), which is quite acceptable when the number of processors is small.

We believe that our parallel algorithms are suitable in the same situations as the corresponding sequential algorithms (Carre 1979), i.e. when "single destination" problems are solved on sparse graphs. The algorithms can be implemented so as to exploit sparsity. For instance, a graph can be represented by lists of arcs emanating from a particular node.

References

R. H. BARLOW, D. J. EVANS (1982) Parallel algorithms for the iterative solution to linear systems. *Computer Journal*, **25**, 56-60.

D. P. BERTSEKAS, J. N. TSITSIKLIS (1989) *Parallel and Distributed Computation - Numerical Methods.* Prentice-Hall, Englewood Cliffs NJ.

B. CARRÉ (1979) *Graphs and Networks.* Oxford University Press, Oxford.

N. DEO ET AL. (1980) Two parallel algorithms for shortest path problems. In *Proceedings of the 1980 International Conference on Parallel Processing, August*, pp. 244-253. IEEE, New York.

S. DEY, P. K. SRIMANI (1989) Fast parallel algorithm for all-pairs shortest path problem and its VLSI implementation. *IEE Proceedings*, **136-E**, 85-89.

T. GAYRAUD, G. AUTHIE (1992) A parallel algorithm for the all pairs shortest path problem. In *Parallel Computing '91* (D. J. Evans, G. R. Joubert, H. Liddell, Eds.), pp. 107-114. Advances in Parallel Computing 4, North-Holland, Amsterdam.

A. GOTTLIEB ET AL. (1983) The NYU ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, **C-32**, 175-189.

V. KONRAD, Y. WALLACH (1977) Iterative solution of linear equations on a parallel processor system. *IEEE Transactions on Computers*, **C-26**, 838-847.

S-Y. KUNG ET AL. (1987) Optimal systolic design for the transitive closure and the shortest path problems. *IEEE Transactions on Computers*, **C-36**, 603-614.

R. MANGER (1990) New examples of the path algebra and corresponding graph theoretic path problems. In *Proceedings of the VII Conference on Applied Mathematics, Osijek, Croatia, September 1989* (R. Scitovski, Ed.), pp. 119-128. University of Osijek, Croatia.

R. MANGER (1993) Implementing parallel "for" loops on multiprocessors. In *Proceedings of the 15th International Conference ITI, Pula, Croatia, June 1993*, (V. Čerić, V. Dobrić, Eds.), pp. 491-496. University Computing Centre, Zagreb, Croatia.

J. A. MCHUGH (1990) *Algorithmic Graph Theory.* Prentice-Hall, Englewood Cliffs NJ.

J. M. ORTEGA, R. G. VOIGT (1985) Solution of partial differential equations on vector and parallel computers. *SIAM Review*, **27**, 149-239.

M. J. QUINN (1987) *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, New York.

G. ROTE (1990) Path problems in graphs. *Computing Supplement*, **7**, 155-189.

B. P. SINHA ET AL. (1986) A parallel algorithm to compute the shortest paths and diameter of a graph and its VLSI implementation. *IEEE Transactions on Computers*, **C-35**, 1000-1004.

T. TAKAOKA (1989) An efficient parallel algorithm for the all pairs shortest path problem. In *Graph-Theoretic Concepts in Computer Science* (J. Van Leeuwen, Ed.), pp. 276-287. Lecture Notes in Computer Science 344, Springer Verlag, Berlin.

U. ZIMMERMANN (1981) *Linear and Combinatorial Optimization in Ordered Algebraic Structures.* Annals of Discrete Mathematics 10, North-Holland, Amsterdam.

Received: November 1992
Accepted: August 1993

Contact address:
Robert Manger
Prirodoslovno-matematički fakultet
Bijenička cesta 30
41000 Zagreb, Croatia
Phone: +385 41 432-459/119
E-mail: manger@math.hr

Robert Manger received the BSc (1979), MSc (1982), and PhD (1990) degrees in mathematics, all from the University of Zagreb. For more than ten years he worked in industry, and in this way he obtained practical experience in programming, computing, and designing information systems. Dr Manger is presently a lecturer in the Department of Mathematics at the University of Zagreb. His current research interests include parallel algorithms and neural networks.