Gaussian block algorithms for solving path problems

ROBERT MANGER*

Abstract. *Path problems are a family of optimization and enumeration problems that reduce to determination or evaluation of paths in a directed graph. In this paper we give a convenient algebraic description of block algorithms for solving path problems. We also develop block versions of two Gaussian algorithms, which are counterparts of the conventional Jordan and escalator method respectively. The correctness of the two considered block algorithms is discussed, and their complexity is analyzed. A parallel implementation of the block Jordan algorithm on a transputer network is presented, and the obtained experimental results are listed.*

Key words: *path problems, path algebras, block algorithms, Gaussian elimination, parallel computing*

Sažetak. **Blok-algoritmi Gaussovog tipa za rješavanje problema putova.** *Problemi putova su porodica optimizacijskih i enumeracijskih problema, koji se svode na određivanje ili vrednovanje putova u usmjerenom grafu. U ovom radu dajemo algebarski opis blok-algoritama za rješavanje problema putova. Također razvijamo blok-verzije dvaju algoritama Gaussovog tipa: riječ je o analogonima standardne Jordanove odnosno eskalator-metode. Raspravlja se o korektnosti i složenosti dvaju razmatranih blok-algoritama. Opisuje se paralelna implementacija blok-Jordanovog algoritma na mreži transputera, te se navode rezultati dobiveni eksperimentiranjem.*

Ključne riječi: *problemi putova, algebre putova, blok-algoritmi, Gaussova eliminacija, paralelno računanje*

AMS subject classifications: 05C85, 68Q25, 68Q22, 65F05

Received January 9, 1998 Accepted March 23, 1998

1. Introduction

Path problems are frequently encountered in operations research. These problems reduce to determination or evaluation of paths in a directed graph. The best known

*Department of Mathematics, University of Zagreb, Bijenička cesta 30, HR-10 000 Zagreb, Croatia, e-mail: manger@cromath.math.hr

example is the shortest path problem, stated as follows: in a graph whose arcs are given lengths determine a shortest path between two given nodes. A similar optimization problem is to find a longest path, or alternatively a most reliable path, or a path with maximum capacity. There are also some enumeration problems, e.g. checking the existence of a path between two given nodes, listing all possible paths, etc.

Each particular path problem can be treated separately, and solved by dedicated algorithms. However, a more economic approach is to establish a general framework for the whole family of problems, and to use general algorithms. The latter can be achieved by introducing a suitable abstract algebraic structure. To solve a particular type of the problem, one should apply a general algorithm to an appropriate concrete instance of the structure.

Few variants of the algebraic approach for solving path problems have been proposed [9]. Our favorite variant from [1] uses a structure whose instances are called "path algebras". Algorithms for solving path problems are developed as counterparts of the well known methods for inverting matrices. More precisely, the derived algorithms are interpreted as procedures working on matrices whose entries belong to a path algebra.

The aim of this paper is to develop Gaussian block algorithms for solving path problems. These are the algorithms that are based on Gaussian elimination and that work on block matrices rather than on ordinary (scalar) matrices. We are interested in general algorithms, which are described within a unifying framework and are applicable to many different types of path problems. Our motivation for considering block algorithms is parallel computing: namely, switching to blocks can drastically reduce otherwise prohibitive communication costs on a parallel computing system.

The paper is organized as follows. *Section 2* reviews the theory from [1] and explains how path algebras are used to generally formulate and solve path problems. *Section 3* extends the theory from [1] in order to cover block matrices and block algorithms. *Section 4* introduces two Gaussian block algorithms for solving path problems - both methods are presented within the extended algebraic framework and are prepared for parallelization. *Section 5* considers the correctness of the two algorithms, and analyzes their computational and communication complexity. *Section 6* describes an actual parallel implementation of one of the algorithms on a transputer network, and lists the obtained testing results. The final *Section 7* gives concluding remarks.

2. Path algebras and path problems

We start our review of the chosen approach to path problems by defining a suitable algebraic structure. A *path algebra* is a set P equipped with two binary operations, $\vee$ and $\circ$, which are called *join* and *multiplication*, respectively. The two operations have the following properties: $\vee$ is idempotent, commutative and associative; $\circ$ is associative, left-distributive and right-distributive over $\vee$; there exist a zero element ϕ and a unit element e such that (for any a) $\phi \vee a = a$, $\phi \circ a = \phi = a \circ \phi$, $e \circ a = a = a \circ e$.

One concrete example of a path algebra will be given in the second part of this section, and two more in *Sections 5* and *6*, respectively. Additional examples can

be found in [1, 4]. When evaluating algebraic expressions over a path algebra P, we will always assume that the operation $\circ$ takes the precedence over $\vee$, unless otherwise regulated by parentheses. Also, we will sometimes interpret P as an ordered structure. Namely, a natural ordering $\preceq$ of P can be defined as follows. For $a, b \in P$, $a \preceq b$ if $a \vee b = b$.

Next we define the notion of stability and closure. Let a be any element of a path algebra P. We consider the powers: $a^0 = e$, $a^1 = a$, $a^2 = a \circ a$, $\ldots$, $a^k = a^{k-1} \circ a$ $(k = 1, 2 \ldots)$. The element 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 expression $a^* = \bigvee_{k=0}^{q} a^k$ is then called the *strong closure* of a, and the expression $\widehat{a} = a^* \circ a = a \circ a^* = \bigvee_{k=1}^{q+1} a^k$ is called the *weak closure* of a. It is easy to check that $a^* = (e \vee a)^r$ if $r \geq q$. This enables efficient evaluation of a^* and $\widehat{a}$ respectively, by successive squaring of $(e \vee a)$.

We further consider matrices over a path algebra P. Let $M_n(P)$ denote the set of all $n \times n$ matrices whose entries belong to P. We define the join and the product of matrices, by analogy with the sum and the product in ordinary linear algebra. Thus for $A, B \in M_n(P)$, $A = [a_{ij}]$, $B = [b_{ij}]$, 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. Since $M_n(P)$ is itself a path algebra, one can speak about stable matrices and their strong or weak closure.

Now we are ready to establish a correspondence between matrices and labeled graphs. Let P be a path algebra. It is obvious that to any matrix $A = [a_{ij}]$ of $M_n(P)$ there corresponds a unique directed graph G with the following properties. The nodes of G are $1, 2, \ldots, n$; if $a_{ij} = \phi$, then the arc (i, j) of G does not exist, else (i, j) exists and is labeled with a_{ij}. Conversely, it is clear that to any graph G of the described form corresponds a unique matrix $A \in M_n(P)$, which is called the *adjacency matrix* of G.

By using the above equivalence between labeled graphs and matrices, it has been demonstrated [1, 4, 9] that various *path problems* can be reduced to computing the strong or weak closure of a (stable) adjacency matrix. Thereby, to each particular problem there corresponds a different (suitably constructed) path algebra P.

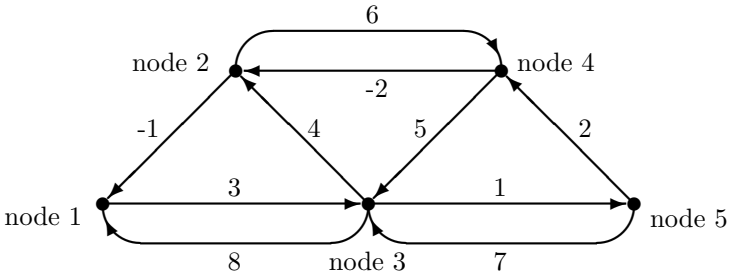

Figure 1. *A directed graph G with given arc lengths*

To illustrate these ideas, let us consider the graph G in *Figure 1* whose arcs are given "lengths". Suppose that we want to solve the shortest distance problem, i.e. we want to determine the length of a shortest path connecting any pair of nodes.

Then the corresponding adjacency matrix A and its closure matrices A^* and $\widehat{A}$, respectively are given by:

$$A = \begin{bmatrix} \infty & \infty & 3 & \infty & \infty \\ -1 & \infty & \infty & 6 & \infty \\ 8 & 4 & \infty & \infty & 1 \\ \infty & -2 & 5 & \infty & \infty \\ \infty & \infty & 7 & 2 & \infty \end{bmatrix}, A^* = \begin{bmatrix} 0 & 4 & 3 & 6 & 4 \\ -1 & 0 & 2 & 5 & 3 \\ 0 & 1 & 0 & 3 & 1 \\ -3 & -2 & 0 & 0 & 1 \\ -1 & 0 & 2 & 2 & 0 \end{bmatrix}, \widehat{A} = \begin{bmatrix} 3 & 4 & 3 & 6 & 4 \\ -1 & 3 & 2 & 5 & 3 \\ 0 & 1 & 3 & 3 & 1 \\ -3 & -2 & 0 & 3 & 1 \\ -1 & 0 & 2 & 2 & 3 \end{bmatrix}.$$

The path algebra P involved here is the set of real numbers augmented by the symbol ∞, with the standard min as $\vee$ and the standard + as $\circ$. It is easy to see that the solution of the shortest distance problem is determined by any of the two closure matrices. Namely, the (i, j)-th entry of both A^* and $\widehat{A}$ is the length of the shortest path between node i and node j. The difference between the two solutions is that A^* assumes the existence of trivial zero-length paths that connect any node to itself.

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, which is a generalization of various concrete problems. The already mentioned idea to evaluate a closure by successive squaring is not very efficient when applied to matrices. Better algorithms are obtained as counterparts of the traditional methods for solving linear systems, e.g. Gauss, Jordan, Jacobi, Gauss-Seidel, ...

Finally, we introduce an important class of graphs and matrices. Let G be a directed graph whose arcs are labeled with elements of a path algebra P. Then G is said to be *absorptive* if the product of its arc labels along any circular path is $\preceq e$. The matrix corresponding to an absorptive graph is also called absorptive. Meaningful path problems usually lead to absorptive matrices. This is for instance true for the problem of *Figure 1*: namely, in that particular example $\preceq$ is equivalent to the conventional $\geq$, e is equal to 0, and G contains no cycle with negative length. It can be shown that an absorptive matrix is always stable.

3. Block matrices and block algorithms

In this section we develop a theoretical framework, which enables a convenient description of block algorithms for solving path problems. Instead of treating block algorithms as modifications of scalar algorithms, we interpret them as applications of scalar algorithms in a modified path algebra. This approach is appealing since it enables both scalar and block algorithms to be considered together. General results concerning path problems and algorithms can directly be applied to block algorithms.

Let P be a path algebra. Then for positive integers $\bar{n}$ and l we can consider the matrix algebras $M_l(P)$ and $M_{\bar{n}}(M_l(P))$. The matrices belonging to $M_{\bar{n}}(M_l(P))$ will be called *block matrices*, and accordingly, the elements of $M_l(P)$ will be called *blocks*.

Let us choose a positive integer l. We further define a function Π_l, which transforms scalar matrices into block matrices with $(l \times l)$-sized blocks. The rule

to construct $\Pi_l(A)$ for a given $A \in M_n(P)$ is as follows. First, A is extended by zero-rows and columns on its right and lower edge (if necessary) so that it becomes a matrix of the size $\lceil n/l \rceil l \times \lceil n/l \rceil l$. Then, the extended A is divided into $(l \times l)$-sized blocks. Each block is interpreted as an element of $M_l(P)$, and the whole extended A is viewed as a matrix belonging to $M_{\lceil n/l \rceil}(M_l(P))$.

As an example illustrating our rule, let us consider again the 5×5 scalar matrix A, which has been used in the previous section. This matrix can be mapped onto a 3×3 block matrix $\Pi_2(A)$ with 2×2 blocks:

$$A = \begin{bmatrix} \infty & \infty & 3 & \infty & \infty \\ -1 & \infty & \infty & 6 & \infty \\ 8 & 4 & \infty & \infty & 1 \\ \infty & -2 & 5 & \infty & \infty \\ \infty & \infty & 7 & 2 & \infty \end{bmatrix}, \quad \Pi_2(A) = \begin{bmatrix} \begin{bmatrix} \infty & \infty \\ -1 & \infty \end{bmatrix} & \begin{bmatrix} 3 & \infty \\ \infty & 6 \end{bmatrix} & \begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix} \\ \begin{bmatrix} 8 & 4 \\ \infty & -2 \end{bmatrix} & \begin{bmatrix} \infty & \infty \\ 5 & \infty \end{bmatrix} & \begin{bmatrix} 1 & \infty \\ \infty & \infty \end{bmatrix} \\ \begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix} & \begin{bmatrix} 7 & 2 \\ \infty & \infty \end{bmatrix} & \begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix} \end{bmatrix}.$$

Remember that the path algebra involved in this example is the one associated with the shortest distance problem.

The next two propositions list some properties of the function Π_l. *Proposition 1* first shows that Π_l is in fact a homomorphism, which embeds the algebra $M_n(P)$ into the algebra $M_{\lceil n/l \rceil}(M_l(P))$. *Proposition 2* then determines the relationship between Π_l and the matrix closure.

Proposition 1. *The function Π_l is injective. Also, for $A, B \in M_n(P)$ it holds:*

$$\Pi_l(A \vee B) = \Pi_l(A) \vee \Pi_l(B), \tag{1}$$

$$\Pi_l(A \circ B) = \Pi_l(A) \circ \Pi_l(B). \tag{2}$$

Proof. The injectivity is obvious, i.e. if the (i, j)-th entries in A and B differ, then they will form different entries in $\Pi_l(A)$ and $\Pi_l(B)$, respectively. To establish (1), it is sufficient to show that (extended $A \vee B$) = (extended A) $\vee$ (extended B). The latter is obvious when taking into account the definition of the matrix join and the fact that $\phi \vee \phi = \phi$. To establish (2), it suffices to show that (extended A$\circ B$) = (extended A) $\circ$ (extended B), and this is easily checked by using the definition of the matrix product and the fact that $\phi \circ x = x \circ \phi = \phi$ for all $x \in P$. □

Proposition 2. *A matrix $A \in M_n(P)$ is stable if and only if $\Pi_l(A) \in M_{\lceil n/l \rceil}(M_l(P))$ is stable. The closures of these matrices then satisfy the following equations:*

$$\Pi_l(A^*) = (\Pi_l(A))^*, \tag{3}$$

$$\Pi_l(\widehat{A}) = (\widehat{\Pi_l(A)}). \tag{4}$$

Proof. If A is stable, then by the definition of stability there exists a non-negative integer q such that $\bigvee_{k=0}^{q} A^k = \bigvee_{k=0}^{q+1} A^k$. Accordingly,

$$\Pi_l(\bigvee_{k=0}^{q} A^k) = \Pi_l(\bigvee_{k=0}^{q+1} A^k).$$

By using (1) and (2), we can transform the above equation into

$$\bigvee_{k=0}^{q} (\Pi_l(A))^k = \bigvee_{k=0}^{q+1} (\Pi_l(A))^k,$$

hence $\Pi_l(A)$ is stable. The whole reasoning can also be followed backwards, since Π_l is injective by *Proposition 1*. As a by-product, the equation (3) is established. The equation (4) is checked similarly. □

Now we are ready to present a general idea how to use already known algorithms for solving path problems in a new way. We have seen that a problem in an algebra P reduces to evaluating the strong or weak closure of a given stable matrix $A \in M_n(P)$. We act as follows.

- For a chosen l, by applying the function Π_l, we map the initial matrix $A \in M_n(P)$ onto the matrix $\Pi_l(A) \in M_{\lceil n/l \rceil}(M_l(P))$.
- By using one of the already known algorithms in the algebra $M_l(P)$, we find the strong or weak closure of the matrix $\Pi_l(A)$. This can be done since $\Pi_l(A)$ is stable by *Proposition 2*.
- Through the inverse function $(\Pi_l)^{-1}$ we put the closure of $\Pi_l(A)$ "back" into the algebra $M_n(P)$, thereby obtaining the corresponding closure of A. This also can be done since Π_l is injective by *Proposition 1*, and due to *Proposition 2* the "returned" matrix is indeed the closure of the initial matrix A.

The procedure is concisely represented by the "commutative" diagram in *Figure 2*.

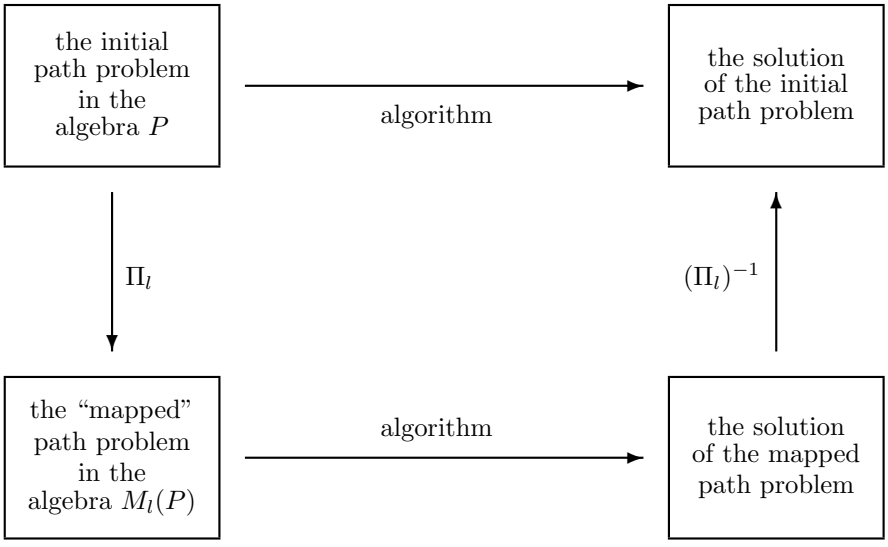

Figure 2. *Transforming an algorithm into a block algorithm.*

The described method for solving a path problem can be regarded as a newly defined *block algorithm*. In fact, a number of block algorithms can be constructed.

Namely, the basic scalar algorithm can be chosen in many ways. Also, various rules for choosing the block size l are possible, e.g. l being a constant, or a specified parameter, or a computed value depending on the matrix size n, ..., etc.

4. Two Gaussian block algorithms

In this section we develop block versions of two already known Gaussian algorithms for solving path problems. According to the ideas from the previous section, these block versions are obtained by using the original (scalar) algorithms in a modified (block) path algebra. Gaussian algorithms for solving path problems are generally based on operations that are analogous to the traditional Gaussian eliminations. The particular two scalar algorithms considered in this section are counterparts of the conventional Jordan and escalator method, respectively [8].

The *Jordan algorithm* for solving path problems has been described in [1]. It is a general path-finding method, whose specialized instances are also known as Floyd's, Warshall's and Murchland's algorithm, respectively [6]. One possible variant is given by the following pseudo-code.

Algorithm 1
(∗ *P is any path algebra. For a given stable matrix $A \in M_n(P)$, the weak closure $\widehat{A}$ is evaluated. Input: the original matrix $A = [a_{ij}]$. Output: the final transformed matrix $[a_{ij}]$.* ∗)

for $k := 1$ **to** n **do**
begin
$c := (a_{kk})^*$;
for all j *such that* $1 \leq j \leq n$ **do**
$a_{kj} := c \circ a_{kj}$;
for all i *such that* $i \neq k$, $1 \leq i \leq n$ **do**
$a_{ik} := a_{ik} \circ c$;
for all (i, j) *such that* $i \neq k$, $j \neq k$, $1 \leq i, j \leq n$ **do**
$a_{ij} := a_{ij} \vee a_{ik} \circ a_{kj}$
end

Algorithm 1 has a simple graph-theoretic interpretation. Namely, in its k-th step the algorithm partially solves the given problem, by considering paths that connect any pair of nodes but use only first k nodes as possible intermediate nodes. Each step adjusts and improves the solution, by introducing one more feasible intermediate node.

Now, there follows the block version of a slightly modified *Algorithm 1*. Thanks to the modification, the same scalar algorithm can also be used to compute the closure of a block within each step of the block algorithm.

Algorithm 2
(∗ *P is any path algebra. For a given stable matrix $A \in M_n(P)$, the weak closure $\widehat{A}$ is evaluated. Input: the original matrix $A = [a_{ij}]$ and the block size l. Output: the final transformed matrix $[a_{ij}]$.* ∗)
(∗ *Let us denote with A_{ij} the block of A consisting of rows $(i-1)l+1$, $(i-1)l+2$, ..., $\min\{il, n\}$ and columns $(j-1)l+1$, $(j-1)l+2$, ..., $\min\{jl, n\}$. Let us denote*

with E_{ij} the corresponding block of the unit matrix $E \in M_n(P)$. $*)$
for $k := 1$ **to** $\lceil n/l \rceil$ **do**
begin
$A_{kk} := \widehat{(A_{kk})}$ ($*$ *by using* Algorithm 1 $*)$;
for all j *such that* $j \neq k$, $1 \leq j \leq \lceil n/l \rceil$ **do** ($*$ *in parallel* $*)$
$A_{kj} := (E_{kk} \vee A_{kk}) \circ A_{kj}$;
for all i *such that* $i \neq k$, $1 \leq i \leq \lceil n/l \rceil$ **do** ($*$ *in parallel* $*)$
$A_{ik} := A_{ik} \circ (E_{kk} \vee A_{kk})$;
for all (i, j) *such that* $i \neq k$, $j \neq k$, $1 \leq i, j \leq \lceil n/l \rceil$ **do** ($*$ *in parallel* $*)$
$A_{ij} := A_{ij} \vee A_{ik} \circ A_{kj}$
end

The *escalator algorithm* for solving path problems has been outlined in [1]. This is again a general method, whose one particular instance is also known as Dantzig's algorithm. The solution is again produced in n steps. However, in its k-th step the method operates only on the submatrix consisting of the first k rows and columns of the original matrix. The details are specified by the following pseudo-code.

Algorithm 3
($*$ *P is any path algebra. For a given stable matrix $A \in M_n(P)$, the strong closure $C = A^*$ is evaluated. Input: the original matrix $A = [a_{ij}]$. Output: the closure matrix $C = [c_{ij}]$.* $*)$
$c_{11} := (a_{11})^*$;
for $k := 2$ **to** n **do**
begin
$c_{kk} := (\bigvee_{i,j=1}^{k-1} a_{ki} \circ c_{ij} \circ a_{jk} \vee a_{kk})^*$;
for all i *such that* $1 \leq i \leq k-1$ **do**
$c_{ik} := (\bigvee_{j=1}^{k-1} c_{ij} \circ a_{jk}) \circ c_{kk}$;
for all j *such that* $1 \leq j \leq k-1$ **do**
$c_{kj} := c_{kk} \circ (\bigvee_{i=1}^{k-1} a_{ki} \circ c_{ij})$;
for all (i, j) *such that* $1 \leq i, j \leq k-1$ **do**
$c_{ij} := c_{ij} \vee c_{ik} \circ c_{kj}$
end

Algorithm 3 has also a plausible graph-theoretic interpretation. Namely, the k-th step of the algorithm solves the given problem on the subgraph consisting of the first k nodes. Each step extends the subgraph by one more node, and adjusts the solution accordingly.

Next there follows the block version of a slightly improved *Algorithm 3*. Some common subexpressions are now computed only once, and for this purpose the auxiliary matrix B has been introduced. The improved scalar algorithm is also used as a subroutine to compute the closure of a block.

Algorithm 4
($*$ *P is any path algebra. For a given stable matrix $A \in M_n(P)$, the strong closure $C = A^*$ is evaluated. $B \in M_n(P)$ is an auxiliary matrix. Input: the original matrix $A = [a_{ij}]$ and the block size l. Output: the closure matrix $C = [c_{ij}]$.* $*)$
($*$ *Let us denote with A_{ij} the block of A consisting of rows $(i-1)l+1$, $(i-1)l+2$,*

$\ldots$, $\min\{il, n\}$ *and columns* $(j-1)l+1$, $(j-1)l+2$, $\ldots$, $\min\{jl, n\}$. *Let us denote with* B_{ij} *and* C_{ij} *the corresponding blocks of* B *and* C *respectively.* $*$)
$C_{11} := (A_{11})^*$ ($*$ *by using* Algorithm 3 $*$) ;
for $k := 2$ **to** $\lceil n/l \rceil$ **do**
begin
for all i *such that* $1 \leq i \leq k-1$ **do** ($*$ *in parallel* $*$)
$B_{ik} := \bigvee_{j=1}^{k-1} C_{ij} \circ A_{jk}$;
for all j *such that* $1 \leq j \leq k-1$ **do** ($*$ *in parallel* $*$)
$B_{kj} := \bigvee_{i=1}^{k-1} A_{ki} \circ C_{ij}$;
$C_{kk} := (\bigvee_{i=1}^{k-1} A_{ki} \circ B_{ik} \vee A_{kk})^*$ ($*$ *by using* Algorithm 3 $*$) ;
for all i *such that* $1 \leq i \leq k-1$ **do** ($*$ *in parallel* $*$)
$C_{ik} := B_{ik} \circ C_{kk}$;
for all j *such that* $1 \leq j \leq k-1$ **do** ($*$ *in parallel* $*$)
$C_{kj} := C_{kk} \circ B_{kj}$;
for all (i, j) *such that* $1 \leq i, j \leq k-1$ **do** ($*$ *in parallel* $*$)
$C_{ij} := C_{ij} \vee C_{ik} \circ C_{kj}$
end

Remember that we are interested in parallel implementations of block algorithms. The comments within previous pseudo-codes indicate which **for** loops can be scheduled for parallel execution. Even more, the tasks from the first and second parallel loop in *Algorithm 2* can be put together into one parallel loop. Similarly, the first and second and respectively the third and fourth loop in *Algorithm 4* can be merged together.

5. Analysis of correctness and complexity

The first part of this section is concerned with correctness issues. Up to this point, it has not been quite clear whether our block algorithms are correct. Moreover, there has been no guarantee that the proposed methods can even be conducted - for instance we are not sure whether the blocks in *Algorithm 2* and *4* whose closures are required are indeed stable. Fortunately, correctness can be proved for most of the important path problems, as stated by the following theorem.

Theorem 1. *Suppose that the matrix* A *is absorptive. Then both* Algorithms 2 *and* 4 *can be conducted. Also,* Algorithm 2 *correctly evaluates the weak closure* $\widehat{A}$*, and* Algorithm 4 *correctly evaluates the strong closure* A^*.

Proof. The correctness of the sequential Jordan algorithm for an absorptive matrix A has been proved in [1]. The proof is general enough to directly cover *Algorithm 2* in the sequential case. On the other hand, parallel computing cannot spoil correctness, since all inner **for** loops in *Algorithm 2* consist of mutually independent tasks. The claim for *Algorithm 4* can be proved by direct algebraic reasoning. Another way is to compare the escalator method with the Jordan method. By using graph-theoretic arguments, it can be shown that the values computed by both algorithms are essentially the same, although the order of operations is different. So the correctness of *Algorithm 4* follows from the correctness of *Algorithm 2*. □

The matrix A being absorptive is a sufficient condition that guarantees the correctness of our Gaussian methods. Still, the question remains whether it is possible to construct a path algebra P and a stable matrix A over P so that the considered algorithms do not work properly. This question has remained open in [1]. Now, we give an example showing that the answer to the above question is positive. Or, differently speaking, *Algorithms* 1-4 are *not generally correct*.

Let P be the set of real numbers augmented by the symbols $-\infty$ and ∞. Let us use the standard min and $+$ as $\vee$ and $\circ$ respectively, with $(-\infty)+\infty=\infty$. Then it is easy to check that P with $\vee$ and $\circ$ constitutes a path algebra. Consider the graph G shown in *Figure 3*, whose arcs are labeled with elements of P. The corresponding matrix $A \in M_2(P)$ and its powers A^2, A^3 are given by

$$A = \begin{bmatrix} -1 & -\infty \\ -\infty & -1 \end{bmatrix}, \quad A^2 = \begin{bmatrix} -\infty & -\infty \\ -\infty & -\infty \end{bmatrix}, \quad A^3 = \begin{bmatrix} -\infty & -\infty \\ -\infty & -\infty \end{bmatrix} = A^2.$$

A is obviously stable in $M_2(P)$ since $\bigvee_{k=0}^{2} A^k = \bigvee_{k=0}^{3} A^k$. On the other hand, (-1) is not stable in P, since $\min\{0,-1,-2,-3,\ldots\}$ does not exist. It follows that *Algorithms 1* and *3* (or *Algorithms 2* and *4* with the block size $l=1$) are not able to work with A. Namely, the algorithms remain blocked, effortlessly trying to evaluate $(-1)^*$. Note that "pivoting", i.e. renumbering graph nodes would not help since both diagonal entries in A are (-1).

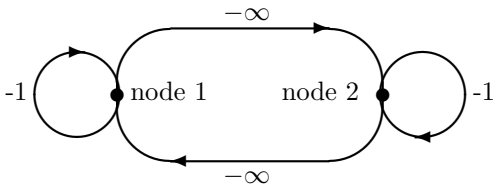

Figure 3. *An example showing that algorithms 1-4 are not generally correct*

The remaining part of this section is devoted to complexity analysis of our block methods. We introduce two measures of complexity, which roughly correspond to the effort spent on computing and communicating respectively. First, let us consider a single algebraic expression. The *computational complexity* is then defined as the number of scalar operations that occur in the expression (i.e. $\vee$, $\circ$). Namely, each of these operations should be executed by a processor. The *communication complexity* is further defined as the number of input/output scalar values that are used or produced by the expression. Namely, these values should be read from or written to a main memory, while intermediate values could temporarily be kept in registers. Finally, the computational (communication) complexity of a whole algorithm is obtained by summing the appropriate complexities of all expressions evaluated by that algorithm.

In our complexity analysis, we will again restrict to path problems defined by absorptive matrices. Namely, more general situations are intractable since there is, for instance, no bound on the number of scalar operations needed to evaluate one scalar closure.

Theorem 2. *Suppose that the matrix A is absorptive. Then the computational complexity of* Algorithm 2 *is $O(n^3)$, and the communication complexity is $O(n^3/l)$. The same estimates are also valid for* Algorithm 4.

Proof. We assume that the block size l divides the matrix size n (otherwise let us switch to the next divisible n - this could increase the complexity only by a constant factor). We also take into account that, thanks to absorptiveness, each of the required scalar closures within the two algorithms must be equal to the unit element e (this fact is not quite obvious but it is visible from the full correctness proof in [1]). With these assumptions and facts in mind, and by counting scalar operations and data transfers within the algorithms, we obtain the following estimates. *Algorithm 2*: the computational complexity is $2n(n^2 - n - l + 1)$, and the communication complexity is $\frac{4n^3}{l} - 2n^2$. *Algorithm 4*: the computational complexity is $2n(n^2 - n - l + 1)$, and the communication complexity is $\frac{8n^3}{3l} + n^2 - \frac{5ln}{3}$. The claimed orders of magnitude follow from these expressions. It must also be noted that l lies between 1 and n. □

The presented estimates give us an idea how switching to block algorithms influences complexity. Namely, the scalar *Algorithms 1* and 3 are only special cases of the corresponding block *Algorithms 2* and 4 respectively. Thus our *Theorem 2* is also applicable to *Algorithm 1* and *3* respectively. By comparing the same estimate for $l = 1$ and for $l > 1$ we see that introduction of blocks reduces the communication complexity. Moreover, the communication complexity becomes smaller as the block size increases. For instance, if we choose l proportional to n, then switching to blocks decreases the communication complexity by an order of magnitude. At the same time, the computational complexity stays virtually the same.

6. Implementation on a transputer network

In this section we describe a parallel implementation of *Algorithm 2* only. Namely, among the two considered block methods, the chosen algorithm seems to be better suited for parallel computing. More precisely, *Algorithm 2* requires the same amount of work in each iteration of its main loop - this enables uniform load distribution among the available processors during the whole process. On the other hand, *Algorithm 4* would not be able to employ many processors initially, but it would require more and more processors at later stages.

Our experimental program has been developed using the Par.C compiler [7], on the MicroWay Quadputer 2 expansion board [3] attached to a standard PC. In accordance with the specification of *Algorithm 2*, most of the code has been written as to work with an abstract (unspecified) path algebra. The definition of the algebra has been isolated in a separate module (i.e. an abstract data type). By changing that module and by recompilation, one can make the same program solve various types of path problems.

The Quadputer 2 board is a network consisting of four INMOS T800 processors, so-called *transputers* [3]. Each pair of transputers is directly connected by a bi-directional communication link. In addition, one of the transputers, called the *root*, is connected through an interface to the PC. Each transputer has its private memory, but there is no common memory. Still, the whole network can emulate a tightly coupled multiprocessor with three processors, as illustrated in *Figure 4*.

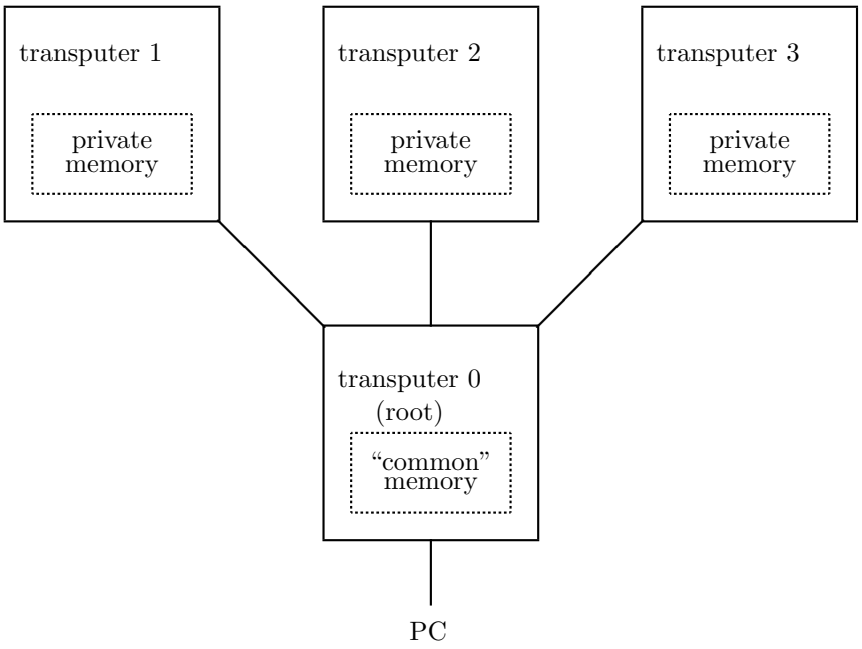

Figure 4. *Emulating a multiprocessor by a transputer network*

In accordance with the above assignment of roles to transputers, our program consists of two executable *tasks*. The first task is a *common memory server* and it runs on the root. The second task is a *client*, which performs computations using data from the common memory. Three copies of the client task run on the three remaining transputers. The operations involving the common memory are realized by message passing between a client and the server. Concurrent access to the common memory is controlled by semaphores. The server also measures the performance time of the whole computation. As input data, the program takes: the block size l, the number of active working processors m (≤ 3), the size n of the matrix A, and the matrix A itself. As output, the program prints the weak closure $\widehat{A}$ and the performance time.

The program was tested on various examples of path problems, expressed in different path algebras. Among other things, testing enabled to monitor how the block size l influences the performance time for a chosen matrix A (with a size n) and a chosen number of processors m. It was observed that as l increases, the performance time initially drops and then again rises. This behaviour can easily be explained. The initial drop of performance time is due to the fact that bigger blocks imply smaller communication costs. The final rise of performance time occurs when the blocks become too big, so that there are too few parallel tasks left to employ the available processors efficiently. It was observed that the optimal value of l depends not only on n and m but also on the path algebra involved. More precisely, it depends on the ratio of the respective times needed for one communication and one computational operation in that algebra.

l	$m=1$	$m=2$	$m=3$
1	9419391	5474994	4707749
2	2198283	1340681	1035446
5	809658	487116	322293
10	583206	338706	252459
15	527140	288687	210563
20	492105	278502	226408
25	475470	283857	203025
30	475555	274698	228114
35	460006	266304	267667
40	460413	280428	248433
45	461315	303171	301598
50	443426	350053	365601
100	456764	463416	471884

Table 1. *Performance time, first example*

Table 1 shows the performance time for a matrix A of size $n = 100$, with the number of processors $m = 1$, 2 or 3. The involved path algebra P is the Boolean algebra, which corresponds to the path existence problem, and which consists of the values 0 and 1 with the operations max and min as $\vee$ and $\circ$, respectively. The time is expressed in transputer clock ticks. We see that, indeed, for a fixed m and with l increasing, the performance time initially drops and then again rises. Small discrepancies from this general trend can be explained by two factors. The first is the influence of divisibility of the integers l, m, n. The second factor is occasional improvement gained by a kind of "cache" memory that has been installed in the program.

In the example of *Table 1*, the average time to transfer an element of P from one transputer to another is 2.925, and a single computational operation (i.e. $\vee$ or $\circ$) with elements of P takes 0.2 on average. As we see, the ratio of the respective times required for one communication and one computational operation is approximately $15 : 1$. Therefore, the optimal performance is attained with a considerably big block. E.g. for $m = 3$, the optimal l is 25. Note that when $l = 25$, the matrix A becomes a 4×4 block matrix; with this number of blocks it is still possible to employ three working processors quite efficiently. By further increasing l, we soon obtain a 3×3 block matrix, and then the processors cannot be evenly loaded any more.

Table 2 presents the performance time for an example which is almost the same as the one of *Table 1*. The only difference is that the execution of computational operations is artificially prolongued, so that the time ratio between a communication and a computational operation is changed to $1 : 4$ approximately. The optimal l decreases accordingly, e.g. for $m = 3$ it drops to 10.

l	$m=1$	$m=2$	$m=3$
1	43745004	22385504	14754103
2	31266306	15829092	10521327
5	26479825	13380264	8943119
10	25087449	12822962	8557807
15	24677584	12626759	8709865
20	24403048	12683125	9754548
25	24263528	13637297	9077744
30	24259407	13323217	10611497
35	24119826	13408201	12826785
40	24116561	14051913	11746907
45	24109259	15339037	14380017
50	23962524	17975139	17990048
100	23774181	23780836	23789316

Table 2. *Performance time, second example*

7. Conclusions

Our newly developed framework extends the algebraic approach from [1], in order to cover block algorithms for solving path problems. We have presented a procedure to turn any scalar algorithm into a block algorithm. This procedure has been applied to the Jordan and escalator method, so that the corresponding block versions have been obtained. The same procedure could as well be applied to some other Gaussian methods or to successive matrix-squaring.

The considered block algorithms work correctly for absorptive matrices. On the other hand, we have found an example, which shows that the algorithms still do not work for any stable matrix over any path algebra, even if "pivoting" is allowed. Our example is quite general, so that it can be applied to any other Gaussian algorithm working in a path algebra. The example in fact shows that the analogy between path problems and classical linear systems is not so extensive as one would wish.

For both the Jordan and escalator method, it holds that switching to a block version will decrease the communication complexity, while the computational complexity will stay virtually the same. This is favorable if the algorithms are to be executed on a multiprocessor, where communication usually costs much more than computing.

Among the two considered block algorithms, the block Jordan method seems to be slightly more suitable for parallelization. Our experimental parallel implementation of the algorithm works on a network of transputers. Contrarily to some other similar works, c.f. [2], our program reflects the generality of the algebraic approach, i.e. the program is able to solve different types of path problems, not only shortest paths. Our experiments indicate that important properties of the algorithm, such as the optimal block size, depend heavily on the path algebra involved.

As we have seen, the Jordan, escalator and similar methods always evaluate the whole closure of an adjacency matrix. However, in many applications only few elements of the closure matrix are really needed. Then, those particular elements can be computed more efficiently by counterparts of the classical iterative algorithms,

e.g. Jacobi, Gauss-Seidel, etc. [8]. Iterative methods also allow efficient sequential and parallel implementations, as shown in [5]. Our theoretical framework for describing block algorithms can easily be extended to cover the mentioned iterative algorithms. Still, this extension is not very interesting from the practical point of view. Namely, it turns out that block versions of iterative algorithms have a similar communication complexity as the corresponding scalar versions.

References

[1] B. CARRÉ, *Graphs and Networks*, Oxford University Press, Oxford, 1979.

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

[3] I. GRAHAM, T. KING, *The Transputer Handbook*, Prentice Hall, Hemel Hempstead UK, 1990.

[4] R. MANGER, *New examples of the path algebra and corresponding graph theoretic path problems*, in: Proceedings of the 7th Seminar on Applied Mathematics, (R. Scitovski, Ed.), Osijek, 1990, 119–128.

[5] R. MANGER, *A comparison of two parallel iterative algorithms for solving path problems*, Journal of Computing and Information Technology - CIT **4**(1996), 75–85.

[6] J. A. MCHUGH, *Algorithmic Graph Theory*, Prentice-Hall, Englewood Cliffs NJ, 1990.

[7] *Par.C System, User's Manual and Library Reference, Version 1.31*, Parsec Developments, Leiden, 1990.

[8] W. H. PRESS, S. A. TEUKOLSKY, W. T. VETTERLING, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd Edition, Cambridge University Press, Cambridge, 1993.

[9] G. ROTE, *Path problems in graphs*, Computing, Supplement **7**(1990), 155–189.