Bull. Nov. Comp. Center, Comp. Science, 27 (2008), 105–117

Efficient associative algorithms for implementing the second group of relational algebra operations

A.S. Nepomniaschaya

Abstract. In this paper, we propose an extended version of the associative graph-machine. Then we offer efficient algorithms for implementing the second group of relational algebra operations that consists of operations Product, Join, and Union. The proposed algorithms are represented as the corresponding procedures for the AG-machine. We prove their correctness and evaluate their time complexity.

1. Introduction

Associative (content addressable) parallel processors of the SIMD type with simple processing elements ideally suit for performing fast parallel search operations being used in different applications such as graph theory, computational geometry, relational database processing, image processing, and genome matching. In [14], the search and data selection algorithms for both bit-serial and fully parallel associative processors were described. In [5, 6], an experimental implementation of a multi-comparand multi-search associative processor and some parallel algorithms for the search problems in computational geometry were considered. In [10], a formal model of the associative parallel processors called associative graph machine (AG-machine) and its possible hardware implementation were proposed. It performs bit-serial and fully parallel associative processing of matrices representing graphs as well as some basic set operations on matrices (sets of columns). The AG-machine differs from that of [6] due to the presence of built-in operations designed for associative graph algorithms. In [1, 8, 11, 13], the relational database processing on conventional associative processors and specialized parallel processors were discussed. In [7], an experimental architecture, called the optical content addressable parallel processor for relational database processing, was devised. It supports the parallel relational database processing by the full exploitation of the optics parallelism. In [3], different optical and optoelectronic architectures for image processing and relational database associative processing were reviewed. In [4], the depth search machines (DSMs) and their applications in computational geometry, relational databases, and artificial intelligence were investigated. In particular, different associative algorithms for processing databases with characteristic functions (CF databases) on parallel DSMs were offered. In [12], an efficient associative algorithm for the multi-comparand search and its applications in

representing the first group of relational algebra operations on the AG-machine were proposed.

In this paper, we propose a new version of the AG-machine and show how this model can efficiently support classical operations in relational databases. In particular, we design efficient associative algorithms for implementing the second group of relational algebra operations. These algorithms are given as corresponding procedures for the AG-machine. We prove their correctness and evaluate their time complexity.

2. A model of the associative graph machine

In this section, we propose a model of the SIMD type with simple single-bit processing elements (PEs) called associative graph machine (AG-machine). It carries out both the bit-serial and the bit-parallel processing. To simulate the access data by contents, the AG-machine uses both *typical operations* for associative systems first presented in Staran [2] and some *new operations* to perform bit-parallel processing.

The model consists of the following components:

- a sequential common control unit (CU), where programs and scalar constants are stored;
- an associative processing unit forming a two-dimensional array of single-bit PEs;
- a matrix memory for the associative processing unit.

The CU broadcasts each instruction to all PEs in one unit of time. All active PEs execute it simultaneously while inactive PEs do not perform it. Activation of a PE depends on the data employed.

Input binary data are loaded in the matrix memory in the form of two-dimensional tables, where each data item occupies an individual row and is updated by a dedicated row of PEs. In the matrix memory, rows are numbered from top to bottom and columns — from left to right. Both a row and a column can be easily accessed.

The associative processing unit is represented as a matrix of single-bit PEs that correspond to the input binary data matrix. Each column in the matrix of PEs can be regarded as a vertical register that maintains the entire column of a table. The model runs as follows. Bit columns of tabular data are stored in the registers which perform the necessary bitwise operations.

To simulate data processing in the matrix memory, we use data types **slice** and **word** for the bit column access and the bit row access, respectively, and the type **table** for defining and updating matrices. We assume that any variable of the type **slice** consists of n components. For simplicity, let us call *slice* any variable of the type **slice**.

For variables of the type **slice**, we employ the same operations as in the case of the STAR-machine along with new operations FRST(Y) and CONVERT(Y).

The *new operation* FRST(Y) saves the first (uppermost) component $'1'$ in the slice Y and assigns to $'0'$ its other components.

The *new operation* CONVERT(Y) returns a row, whose every ith component (bit) coincides with $Y(i)$. It is used as the right part of the assignment statement.

It should be noted that the operation CONVERT(Y) was implemented in the Russian associative processor ES-27-20. However, earlier it was not included into the STAR-machine in view of the absence of its application.

For the sake of completeness, we recall some elementary operations for slices from [9]:

SET(Y) sets all components of the slice Y to $'1'$;

CLR(Y) sets all components of Y to $'0'$;

FND(Y) returns the ordinal number of the first component $'1'$ of Y;

STEP(Y) returns the same result as FND(Y) and then resets the first $'1'$ found to $'0'$;

NUMB(Y) returns the number of components $'1'$ in the slice Y;

MASK(Y, i, j) sets components $'1'$ from the ith through the jth positions, inclusively, and components $'0'$ in other positions of the slice Y ($1 \le i < j \le n$);

SHIFTDOWN(Y, k) moves the contents of Y by k positions down, placing each component from the position i to the position $i + k$ ($i \ge 1$) and setting components $'0'$ from the first through the k-th position, inclusive.

In the usual way, we introduce the predicates ZERO(Y) and SOME(Y) and the bitwise Boolean operations X *and* Y, X *or* Y, *not* Y, X *xor* Y.

The above-mentioned operations along with the operation FRST(Y) are also used for variables of the type **word**.

For a variable T of the type **table**, we use the following two operations:

ROW(i, T) returns the ith row of the matrix T;

COL(i, T) returns the ith column of the matrix T.

Moreover, we use two groups of *new operations*. One group of such operations is applied to a single matrix, while the other one is used for two matrices of the same size. All new operations are implemented in the hardware.

Now we present the *first group* of new operations.

The operation SCOPY(T, X, v) *simultaneously* writes the given slice X into those columns of the given matrix T which are marked with $'1'$ in the given comparand v.

The operation RCOPY(T, v, X) *simultaneously* writes the given word v into those rows of the given matrix T which are marked with $'1'$ in the given slice X.

The operation $not(T, v)$ *simultaneously* replaces columns of the given matrix T, marked with $'1'$ in the comparand v, with their negation. It will be used as the right part of the assignment statement.

The operation FRST(row, T) *simultaneously* performs the operation FRST(v) for every row of the matrix T and writes the result into T.

The operation FRST(col, T) *simultaneously* performs the operation FRST(Y) for every column of the matrix T and writes the result into T.

The operation or(row, T) *simultaneously* performs the disjunction in every row of the matrix T. It returns a slice, whose every ith component is equal to $'0'$ if and only if ROW(i, T) consists of zeros. It is used as the right part of the assignment statement.

The operation and(row, T) *simultaneously* performs the conjunction in every row of the matrix T. It returns a slice whose every ith component is equal to $'1'$ if and only if ROW(i, T) consists of ones. This operation is used as the right part of the assignment statement.

The operation or(col, T) *simultaneously* performs the disjunction in every column of the matrix T. It returns a row whose every i-th bit is equal to $'0'$ if and only if COL(i, T) consists of zeros. This operation is used as the right part of the assignment statement.

The operation SHIFTDOWN(T, k) *simultaneously* performs the operation SHIFTDOWN(Y, k) for all columns of the given matrix T.

Now we determine the *second group* of new operations.

The operation SMERGE(T, F, v) *simultaneously* writes columns of a given matrix F that are marked with $'1'$ in the comparand v, into the corresponding columns of the result matrix T. If the comparand v consists of $'1'$, the operation SMERGE copies the matrix F into the matrix T.

The operation $op(T, F, v)$, where $op \in \{or, and, xor\}$, is *simultaneously* performed between those columns of the given matrices T and F that are marked with $'1'$ in the given comparand v. This operation is used as the right part of the assignment statement, that is, $R := op(T, F, v)$.

Remark 1. We will assume that each elementary operation of the AG-machine (its microstep) takes one unit of time.

3. Implementation of the second group of relational algebra operations on the AG-machine

A relational database is defined by analogy with [15]. Let D_i be a domain, $i = 1, 2, \ldots, k$. The relation R is determined as a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_k$. An element of R is called *tuple* and has the form $v = (v_1, v_2, \ldots, v_k)$, where $v_i \in D_i$. Let A_i be the name of the domain D_i which is called *attribute*. Let $R(A_1, A_2, \ldots, A_k)$ denote a *scheme* of the relation R.

On the AG-machine, any relation is represented as a matrix and each its tuple is allocated to one memory row. Note that any relation consists of *different* tuples.

The relational algebra operations are divided into two groups. The *first group* consists of the following operations: Intersection, Difference, Semi-join, Projection, and Division. The resulting relation for these operations is a subset of the argument relations T and F. The *second group* consists of the operations Product, Join, and Union. These operations assemble a new relation.

To implement the operation Join on the AG-machine, we will use the basic procedure MATCH(T, X, w, Z). Therefore we first propose an efficient associative algorithm for implementing the procedure MATCH on the AG-machine.

The procedure MATCH determines *positions* of the matrix T rows that coincide with a given pattern w. By means of the slice X, we mark with $'1'$ the matrix T rows used for comparison with w. It returns the slice Z, where $Z(i) =' 1'$ if and only if ROW$(i, T) = w$ and $X(i) =' 1'$. Notice that on the STAR-machine, the procedure MATCH requires $O(k)$ time, where k is the number of columns in the matrix T.

Let us consider implementation of the procedure MATCH on the AG-machine. This makes use of the following idea. For every $1 \leq i \leq |w|$, we *simultaneously* compare the ith bit of every row of the matrix T with the ith bit of a given pattern w.

```
   procedure MATCH(T: table; X: slice(T); w: word;
                   var Z: slice(T));
   var A,B: table; u,v: word(T);
1. Begin SET(u); v:= not w;
2.   SCOPY(A,X,u);
   /* We write the slice X in all columns of the matrix A. */
3.   B:=not(T,v);
   /* We write the negation of every column of the matrix T that corresponds to '0'
      in the given string w. */
4.   A:=and(A,B,u);
```

```
5.   Z:=and(row,A);
```

/* We obtain that $Z(i) =' 1'$ if ROW(i, A) consists of ones, that is, ROW$(i, T) = w$. */

```
6. End;
```

Proposition 1. *Let T be a matrix, X be a slice, where positions of the matrix T rows to be analyzed are marked with $'1'$, and w be a pattern. Then the procedure* MATCH(T, X, w, Z) *returns the slice Z, where $Z(i) = w$ if and only if* ROW$(i, T) = w$ *and $X(i) = '1'$.*

Proof. We prove this by contradiction. Assume that ROW$(j, T) = w$, $X(j) = '1'$ but $Z(j) =' 0'$. Let us analyze the execution of the procedure MATCH. After performing lines 1–2, ROW(j, A) consists of ones because $X(j) =' 1'$. After performing line 3, ROW(j, B) also consists of ones because we write the negation of every bit of ROW(j, T) that corresponds to $'0'$ in the given string w. Therefore after performing lines 4–5, ROW(j, A) consists of ones and $Z(j) =' 1'$. This contradicts to our assumption. □

Obviously, on the AG-machine, the procedure MATCH takes $O(1)$ time.

3.1. Implementation of the operation Product on the AG-machine. Here, we consider the operation Product. Its resulting relation $R(R1, R2)$ is obtained as concatenation of all combinations of the argument relations T and F.

Let us explain the main idea of implementing the operation Product on the AG-machine. Let the relation T consist of p tuples and the relation F consist of s tuples. We first build the attribute $R1$, where s copies of every tuple from T are written. We do this with the use of operations MASK, RCOPY, and SHIFTDOWN. To build the attribute $R2$, we first mark with $'1'$ in a slice, say $Z2$, the positions of rows in $R2$, where p copies of the first tuple from F are written. Then by means of operations RCOPY and SHIFTDOWN, we write the tuples from the relation F into the corresponding rows of $R2$.

```
procedure Product(T: table; F: table; var X: slice(T);
                  var Y: slice(F); var R(R1,R2): table);
```

/* The rows of T are marked with $'1'$ in the slice X, and the rows of F are marked with $'1'$ in the slice Y. */

```
var i,j,p,s: integer; Z1,Z2,Z: slice(R);
    v: word(T); v1: word(F);
1. Begin p:=NUMB(X); s:=NUMB(Y);
2.   MASK(Z,1,s);
```

/* We set $'1'$ in the first s bits of the slice Z. */

```
3.   while SOME(X) do
4.     begin i:=STEP(X); v:=ROW(i,T);
5.       RCOPY(R1,v,Z);
   /* We copy the string v in the rows of R1 marked with '1' in the slice Z. */
6.       SHIFTDOWN(Z,s);
7.     end;
8.   CLR(Z1); Z1(1):='1';
9.   CLR(Z2); Z2(1):='1';
10.  for j:=1 to p-1 do
11.    begin SHIFTDOWN(Z1,s);
12.      Z2:=Z2 or Z1;
13.    end;
   /* We mark with '1' in Z2 positions of the matrix R2 rows, where p copies of
      its first string are written. */
14.  while SOME(Y) do
15.    begin i:=STEP(Y);
16.      v1:=ROW(i,F);
17.      RCOPY(R2,v1,Z2);
   /* We write the string v1 in the rows of R2 marked with '1' in Z2. */
18.      SHIFTDOWN(Z2,1);
19.    end;
20. End;
```

Proposition 2. *Let a relation T have p tuples whose positions are marked with $'1'$ in the slice X. Let a relation F have s tuples whose positions are marked with $'1'$ in the slice Y. Then the procedure* Product *returns the relation $R(R1, R2)$, where s copies of any tuple of T are created in the attribute $R1$ and p copies of the relation F tuples are created in the attribute $R2$.*

Proof. (Sketch.) We prove this by contradiction. Let there be such a tuple v_1 in the relation T and a tuple v_2 in the relation F that v_1v_2 does not belong to the resulting relation R. We will prove this to contradict to executing the procedure Product.

Really, after performing lines 1–2, the variables p and s save the number of tuples in the relations T and F, respectively, and the first s bits of the slice Z are equal to $'1'$. After fulfilling lines 3–6, we select the first tuple in the relation T and simultaneously write it into the rows of the attribute $R1$ marked with $'1'$ in the slice Z. After that, we shift the contents of the slice Z down by s bits. Hence, the slice Z will save positions of rows in $R1$, where s copies of the next tuple of T will be written. After execution of the cycle `while SOME(X) do` (lines 3–7), s copies of *every* tuple from T will be written into $R1$. Hence, s copies of v_1 are also written into the

attribute $R1$. It is easy to check that after performing lines 8–13, the slice $Z2$ saves positions of rows in the attribute $R2$, where p copies of the first tuple from F are to be written. After execution of the cycle `while SOME(Y) do` (lines 14–19), p copies of a group of s tuples from F are written into the attribute $R2$. Since the tuple v_2 belongs to the relation F, it belongs to *every* copy of the group of s tuples in the attribute $R2$. Let v_1 be the kth tuple in T and v_2 be the ith tuple in F. Then the tuple v_1v_2 has been written into the $(i + (k - 1)s)$-th row of the resulting relation R. This contradicts to our assumption. □

Obviously, the time complexity of the procedure Product on the AG-machine is $O(p + s)$.

In [11], we proposed implementation of the operation Product on the STAR-machine. This implementation uses an auxiliary matrix G obtained by compaction of the relation F. We have also shown how to implement this operation using a modified version of the STAR-machine joined with a special hardware support called λ-processor. This processor allows one to execute the matrix compaction by means of the vertical processing. As shown above, the efficient implementation of the operation Product on the AG-machine does not use the compaction of the relation F. We avoid the compaction of a matrix due to the use of operations SHIFTDOWN and RCOPY.

3.2. Implementation of the operation Join on the AG-machine. Recall the operation Join. Let $A(A1, A2)$ and $B(B1, B2)$ be two relations. Let attributes $A2$ and $B2$ be drawn from the same domain. The operation Join performs concatenation in every group of attributes $A1$ and $B1$ for which the corresponding values of attributes $A2$ and $B2$ are equal.

Explain the main idea of implementing the operation Join on the AG-machine. Let $C(C1, C2)$ be the result relation of the operation Join. Initially, we set zeros in the attributes $C1$ and $C2$. Then we select the current tuple v in the attribute $A2$ and determine all its occurrences both in $A2$ and in $B2$. If v belongs to $B2$, we fulfil the procedure Product between the corresponding rows of attributes $A1$ and $B1$ and include the result in the corresponding rows of $C1$ and $C2$. Otherwise, we analyze the next tuple in $A2$. We do this with the use of basic operations of the AG-machine and procedures MATCH and Product.

```
  procedure Join(A(A1,A2): table; B(B1,B2): table; Y: slice(B);
                 var X: slice(A); var C(C1,C2): table);
  var X1: slice(A); Y1: slice(B); Z: slice(C);
      v: word(A2); v1: word(A1); v2: word(B1);
      i,s1,r1,t: integer; E1,E2: table;
1. Begin t:=0; CLR(Z);
```

```
2.   SET(v1); SET(v2);
3.   SCOPY(C1,Z,v1);
4.   SCOPY(C2,Z,v2);
   /* We set zeros in the attributes C1 and C2. */
5.   while SOME(X) do
6.     begin i:=FND(X); v:=ROW(i,A2);
7.       MATCH(A2,X,v,X1);
   /* Positions of the attribute A2 rows that coincide with v are marked with '1'
      in the slice X1. */
8.       X:=X and (not X1);
   /* We mark with '0' in the slice X positions of the attribute A2 rows
      that coincide with v. */
9.       MATCH(B2,Y,v,Y1);
   /* We mark with '1' in the slice Y1 positions of the attribute B2 rows
      that coincide with v. */
10.      if SOME(Y1) then
11.        begin r1:=NUMB(X1); s1:=NUMB(Y1);
12.          Product(A1,B1,X1,Y1,E1,E2);
13.          SHIFTDOWN(E1,t);
14.          SHIFTDOWN(E2,t);
15.          C1:=or(C1,E1,v1);
16.          C2:=or(C2,E2,v2);
   /* We include the result of shifting the matrix E1 (respectively, E2)
      in the attribute C1 (respectively, C2). */
17.          t:=t+r1*s1;
18.        end;
19.    end;
20. End;
```

Proposition 3. *Let two argument relations $A(A1, A2)$ and $B(B1, B2)$ be given. Let a slice X save positions of tuples that belong to A, and a slice Y save positions of tuples that belong to B. Let the attributes $A2$ and $B2$ be drawn from the same domain. Then the procedure* Join *returns the concatenation of those rows from the attributes $A1$ and $B1$, for which the corresponding values of $A2$ and $B2$ are equal.*

Proof. (Sketch.) We prove this by induction on the number of different tuples l that belong to the attributes $A2$ and $B2$.

Basis is checked for $l = 1$, that is, only a single tuple belongs to $A2$ and $B2$. After performing lines 1–4, we set zeros in the attributes $C1$ and $C2$. After performing lines 6–9, we select the first tuple v in the attribute $A2$. Then with the slice $X1$, we save positions of all occurrences of v in $A2$ and

delete them from the slice X. Without loss of generality we assume that v is a single tuple that belongs to $A2$ and $B2$. Therefore after performing lines 9–10, with the slice $Y1$, we save positions of all occurrences of v in the attribute $B2$. Since $Y1 \neq \Theta$*, we determine the number of tuples $r1$ in the attribute $A1$ and the number of tuples $s1$ in the attribute $B1$ (line 11). Since A is a relation, the rows in the attribute $A1$ that correspond to the same tuple v in $A2$ are different. The same we have for the relation B. Moreover, the attributes $A1$ and $B1$ are drawn from different domains. Therefore we apply the procedure Product (line 12). Since initially $t = 0$, after performing lines 13–16, we obtain the attributes $C1$ and $C2$. After that we determine a new value for t (line 17) and terminate the conditional statement from line 10.

If $X \neq \Theta$, we select by $'1'$ positions of all occurrences of the next tuple in $A2$ and delete them from X as shown above. We continue this process while $X \neq \Theta$. After that we go to the procedure end.

Step of induction. Let the assertion be true when $l \geq 1$ different tuples belong both to the attribute $A2$ and the attribute $B2$. We prove the assertion for the case when $l + 1$ different tuples belong to $A2$ and $B2$. By the inductive assumption, after selecting the first l different tuples, their positions are selected by $'1'$ in the slice X. After that, these positions are deleted from the slice X (line 8). Each time when a selected group of the same tuple belongs to $B2$, we perform the procedure Product which is applied to the corresponding attributes $A1$ and $B1$. Since a new value for t is computed after every execution of the procedure Product (line 17), a new result of this procedure is written into the corresponding rows of the attributes $C1$ and $C2$. Further we reason by analogy with the basis when a single tuple belongs to $A2$ and $B2$. As soon as we select positions of all occurrences of this tuple (lines 6–10), we perform the procedure Product for the corresponding rows of the attributes $A1$ and $B1$ and write the result into the matrices $E1$ and $E2$. After performing lines 13–16, the result of shifting the contents of the matrices $E1$ and $E2$ are written into the attributes $C1$ and $C2$. As soon as the slice $X = \Theta$, we run to the procedure end. □

Let k be the number of different tuples that belong both to $A2$ and to $B2$. Let us denote by p_i and s_i the number of different occurrences of the i-th tuple that belong to $A2$ and $B2$, respectively. Then the procedure Join takes $O\big(\sum_{i=1}^{k}(p_i + s_i)\big)$ time.

3.3. Implementation of the operation Union on the AG-machine. The operation Union is applied to the argument relations T and F with the same number of bit columns k. The resulting relation P is assembled from the relation T and those tuples of the relation F, which do not belong to T.

*The notation $Y1 \neq \Theta$ denotes that there is at least one component $'1'$ in the slice $Y1$.

To implement the operation Union on the AG-machine, we use the procedure Differ$(F, Y, T, k, Y1)$ from [12]. It returns a slice $Y1$ that saves positions of the relation F tuples not belonging to the relation T. In [12], we have shown that the procedure Differ takes $O(k)$ time.

Explain the main idea of implementing the operation Union on the AG-machine. We first copy the relation T into the relation P. Then we perform the procedure Differ and save *positions* of the relation F tuples that do not belong to the relation T. Further we include into P every tuple v from the relation F that does not belong to the relation T. Moreover, the *position* of the tuple v in the relation P is marked with $'1'$ in the slice Z.

```
procedure Union(T: table; F: table; X: slice(T); Y: slice(F);
                k: integer; var P: table; var Z: slice(P));
var Z1: slice(P); Y1: slice(F); v,w: word(T); i,j: integer;
Begin SET(v); Z:=X;
  SMERGE(T,P,v);
/* We copy the relation T into P. */
  Differ(F,Y,T,k,Y1);
/* The slice Y1 saves positions of the relation F rows that do not belong to T. */
  while SOME(Y1) do
    begin i:=STEP(Y1); w:=ROW(i,F);
      Z1:= not Z; j:=FND(Z1);
      ROW(j,P):=w; Z(j):='1';
    end;
End;
```

Proposition 4. *Let two argument relations T and F be given. Let a slice X save positions of tuples that belong to T, and a slice Y save positions of tuples that belong to F. Then the procedure* Union *returns the relation P that consists of different tuples from relations T and F, and a slice Z that saves positions of the tuples from P.*

The correctness of the procedure Union is proved by induction on the number of tuples l of the relation F that do not belong to the relation T.

Let l be the number of tuples of the relation F that do not belong to the relation T. Then the procedure Union takes $O(k + l)$ time because every operation of the AG-machine takes one unit of time and the procedure Differ from [12] takes $O(k)$ time. Notice that on the STAR-machine, this procedure takes $O(kr)$ time, where r is the number of tuples in the relation F, because for every tuple from the relation F one has to check whether it belongs to the relation T.

4. Conclusions

In this paper, we propose a new version of the associative graph machine. We have also offered efficient associative algorithms for implementing the relational algebra operations Product, Join, and Union on this model. On the AG-machine, these algorithms are represented as the corresponding procedures and their correctness is justified. We have also compared implementations of these operations on the AG-machine and on the STAR-machine. We have shown that efficient implementations of operations Product and Join on the AG-machine do not use the compaction of a relation. Therefore there is no need of using a special hardware support for compaction of a relation as in the case of the STAR-machine. We have shown that the estimations obtained for the AG-machine are optimal.

We are planning to study an application of the AG-machine to performing the genome matching.

References

[1] Fernstrom C., Kruzela J., Svensson B. LUCAS associative array processor. Design, programming and application studies. — Berlin: Springer, 1986. — (Lect. Notes Comp. Sci.; 216).

[2] Foster C.C. Content Addressable Parallel Processors. — New York: Van Nostrand Reinhold Company, 1976.

[3] Irakliotis L.J., Betzos G.A., Mitkas P.A. Optical associative processing // Associative Processing and Processors / A. Krikelis, C.C. Weems (eds). — IEEE Computer Society Press, 1997. — P. 155–178.

[4] Kapralski A. Sequential and Parallel Processing in Depth Search Machines. — Singapore: World Scientific, 1994.

[5] Kokosiński Z. An associative processor for multi-comparand parallel searching and its selected applications // Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA'97. Las Vegas, USA. — 1997. — P. 1434–1442.

[6] Kokosiński Z., Sikora W. An FPGA implementation of multi-comparand multisearch associative processor // Proc. 12th Int. Conf. FPL'2002. — Berlin: Springer, 2002. — P. 826–835. — (Lect. Notes Comp. Sci.; 2438).

[7] Louri A., Hatch J.A. An optical associative parallel processor for high-speed database processing // Computer. — 1994. — Vol. 27, No. 11. — P. 65–72.

[8] Muraszkiewicz M.R. Cellular array architecture for relational database implementation // Future Generations Computer Systems. — 1988. — Vol. 4, No. 1. — P. 31–38.

[9] Nepomniaschaya A.S. Language STAR for associative and parallel computation with vertical data processing // Proc. of the Intern. Conf. "Parallel Computing Technologies". — Singapore: World Scientific, 1991. — P. 258–265.

[10] Nepomniaschaya A.S., Kokosiński Z. Associative graph processor and its properties // Proc. of the Intern. Conf. PARELEC'2004. — Dresden: IEEE Computer Society Press, 2004. — P. 297–302.

[11] Nepomniaschaya A.S., Fet Y.I. Investigation of some hardware accelerators for relational algebra operations // Proc. of the First Aizu Intern. Symp. on Parallel Algorithms / Architecture Synthesis, Aizu-Wakamatsu, Fukushima, Japan. — IEEE Computer Society Press, 1995. — P. 308–314.

[12] Nepomniaschaya A.S. An efficient associative algorithm for multi-comparand parallel searching and its applications // Bull. Novosibirsk Comp. Center. Ser. Computer Science. — Novosibirsk, 2006. — Iss. 25. — P. 38–49.

[13] Ozkarahan E. Database Machines and Database Management. — Prentice-Hall, Inc., 1986.

[14] Parhami B. Search and data selection algorithms for associative processors // Associative Processing and Processors / A. Krikelis, C.C. Weems (eds). — IEEE Computer Society Press, 1997. — P. 10–25.

[15] Ullman J.D. Principles of Database Systems. — Computer Science Press, 1980.