Computing and Informatics, Vol. 29, 2010, 467–487

PARALLEL IMPLEMENTATION OF RELATIONAL ALGEBRA OPERATIONS ON A MULTI-COMPARAND ASSOCIATIVE MACHINE

Anna Shmilevna Nepomniaschaya

Institute of Computational Mathematics and Mathematical Geophysics
Siberian Division of the Russian Academy of Sciences
pr. Lavrentieva 6
630090 Novosibirsk, Russia
e-mail: anep@ssd.sscc.ru

Manuscript received 4 October 2008; revised 8 December 2009
Communicated by Ivan Plander

Abstract. In this paper, we propose a new multi-comparand associative machine (MCA-machine) and its application to relational algebra operations. We first offer a new efficient associative algorithm for the multi-comparand parallel search. It generalizes the Falkoff associative algorithm that performs a parallel search in a matrix based on the exact match with a given pattern. Then we apply the new associative algorithm to implement one group of the relational algebra operations on the MCA-machine. Then, we propose efficient associative algorithms for implementing another group of the relational algebra operations. The proposed algorithms are represented as corresponding procedures for the MCA-machine. We prove their correctness and evaluate their time complexity.

Keywords: SIMD architecture, data parallelism, operation of the exact match, associative parallel processor, bit-serial processing, bit-parallel processing, associative parallel algorithm

Mathematics Subject Classification 2000: 68-XX, 68U99

1 INTRODUCTION

Associative (content addressable) parallel processors of the SIMD type with simple processing elements are ideally suited 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 [19], the search and data selection algorithms for both bit-serial and fully parallel associative processors were described. In [5], the depth search machines and their applications to computational geometry, relational databases, and expert systems were investigated. In [6, 7], an experimental implementation of a multi-comparand multi-search associative processor and some parallel algorithms for search problems in computational geometry were considered. In [11], a formal model of associative parallel processors called the 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 in [7] due to the presence of built-in operations designed for associative graph algorithms.

In [2, 9, 12, 18], the relational database processing on conventional associative processors and specialized parallel processors were discussed. In [8], an experimental architecture, called the optical content addressable parallel processor for the relational database processing, was devised. It supports the parallel relational database processing by fully exploiting the parallelism of optics. In [4], different optical and optoelectronic architectures for image processing and relational database processing were reviewed.

In this paper, we propose a new multi-comparand associative machine (MCA-machine) and show how this model can efficiently support classical operations in relational databases. We first propose a new associative algorithm for the multi-comparand search and its implementation on the MCA-machine. It generalizes the Falkoff associative algorithm [1] that simultaneously selects those rows in a given matrix that coincide with a given pattern. Then we consider applications of this algorithm to representing one group of the relational algebra operations whose resulting relation is a subset of the corresponding argument relations. After that we propose efficient associative algorithms for implementing another group of the relational algebra operations, where every operation assembles a new relation. The proposed algorithms are given as corresponding procedures for the MCA-machine. We prove their correctness and evaluate their time complexity.

2 A MODEL OF A MULTI-COMPARAND ASSOCIATIVE MACHINE

In this section, we first explain, why the new model is introduced.

By means of the STAR-machine [10], we have represented both the associative versions of some classical graph algorithms (for example, [14–16]) and classical operations in relational databases [13]. The proposed associative algorithms utilize the following properties of associative systems with vertical processing: data parallelism,

bit-serial processing, and access data by contents. To improve the time complexity of associative graph algorithms, the AG-machine was proposed in [11]. It allows one to use both the bit-serial and the bit-parallel processing. Due to the bit-parallel processing, some parts of a given associative graph algorithm can be performed in parallel [17]. To improve the time complexity of performing the relational algebra operations, we have considered in [12] a modified version of the STAR-machine joined with two hardware supports: the set intersection processor and the λ-processor. The MCA-machine allows to implement the classical relational algebra operations with the same time complexity as in the case of a modified version of the STAR-machine [12].

We define the model as an abstract MCA-machine of the SIMD type with simple single-bit processing elements (PEs). To simulate the access data by contents, the MCA-machine uses both the *typical operations* for associative systems first presented in Staran [3] and a group of *new operations* to perform the 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 passes each instruction to all PEs in one unit of time. All active PEs execute it simultaneously, while inactive PEs do not. Activation of a PE depends on the data employed.

Input binary data are given in the form of two-dimensional tables, where each datum occupies an individual row to be updated by a dedicated row of PEs. In any table, rows are numbered from top to bottom and columns - from left to right.

The associative processing unit is represented as a matrix of single-bit PEs that corresponds to the matrix of input binary data. Each column in the matrix of PEs can be regarded as vertical register that maintains the entire column of a table.

To simulate data processing in the associative processing unit, we use the 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 any variable of the type *slice* to consist of n components. For simplicity, let us call *slice* any variable of the type *slice*.

To perform bit-serial (vertical) processing, the MCA-machine employs the same operations for slices as the STAR-machine [10] along with new operations FRST(Y) and CONVERT(Y).

For the sake of completeness, we recall some elementary operations for slices being used in this paper.

- SET(Y) simultaneously sets all components of the slice Y to $'1'$;
- CLR(Y) simultaneously 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), then resets the first found $'1'$ to $'0'$. For example, let $Y = '0100101'$ and the statement $k :=$STEP(Y) be performed. Then we obtain $k = 2$ and $Y = '0000101'$;
- FRST(Y) saves the first (the uppermost) component $'1'$ in the slice Y and sets to $'0'$ its other components. For example, let $Y = '0100101'$ and the operation FRST(Y) be performed. Then we obtain $Y = '0100000'$;
- NUMB(Y) returns the number of components $'1'$ in the slice Y;
- MASK(Y, i, j) sets components $'1'$ from the i^{th} through the j^{th} positions, inclusively, and components $'0'$ in other positions of the slice Y $(1 \leq i < j \leq n)$;
- SHIFT($Y, down, k$) moves the contents of Y by k positions down, placing each component from the position i to the position $i + k$ $(n - k \geq i \geq 1)$ and setting components $'0'$ from the first through the k^{th} positions, inclusive;
- CONVERT(Y) returns a row whose every i^{th} component (bit) coincides with $Y(i)$. It is applied when a column of one matrix is used as a comparand for another matrix.

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 predicates and operations for slices, except SHIFT, 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 i^{th} row of the matrix T;
- COL(i, T) returns the i^{th} column of T.

To perform the bit-parallel processing, the MCA-machine uses two groups of new elementary operations for variables of the type *table*. One group of such operations is applied to a single binary matrix, while the other one is used for two binary matrices of the same size. In any binary matrix, its rows can be masked by means of a variable of the type *slice*, while its columns can be masked by means of a variable of the type *word*.

Now we present the *first group* of elementary operations for matrices.

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

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

The operation FRST(row, T) simultaneously performs FRST(v) for every row v of the matrix T and writes the result into T (Figure 2).

The operation FRST(col, T) simultaneously performs FRST(Y) for every column Y of the matrix T and writes the result into T (Figure 2).

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

Now, we present a group of logical operations for binary matrices. Every logical operation will be used as the right part of the assignment statement.

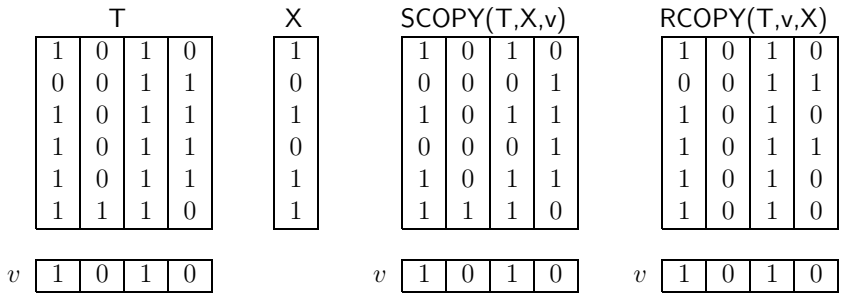

Fig. 1. The use of SCOPY(T, X, v) and RCOPY(T, v, X)

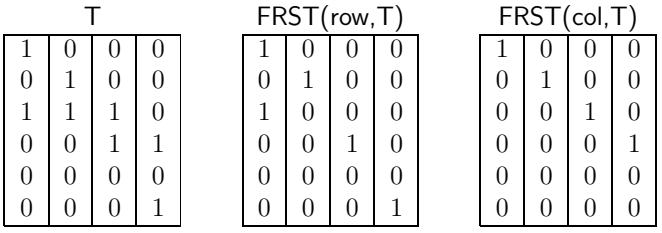

Fig. 2. The use of FRST(row, T) and FRST(col, T)

The operation not (T, v) simultaneously replaces the columns of the given matrix T, marked with $'1'$ in the comparand v, with their negation.

The operation or (row, T) simultaneously performs the disjunction in every row of the matrix T. It returns a slice whose every i^{th} component is equal to $'0'$ if and only if ROW(i, T) consists of zeros.

The operation and (row, T) simultaneously performs the conjunction in every row of the matrix T. It returns a slice whose every i^{th} component is equal to $'1'$ if and only if ROW(i, T) consists of ones.

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.

Now, we present the second group of elementary operations for matrices.

The operation SMERGE(T, F, v) simultaneously writes the columns of the given matrix F that are marked with $'1'$ in the comparand v in the corresponding columns of the resulting matrix T. If the comparand v consists of ones, the operation SMERGE copies the matrix F into the matrix T (Figure 3).

The operation op (T, F, v), where op $\in \{\text{or}, \text{and}, \text{xor}\}$, 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 assign-

ment statement. For example, let a binary matrix R be a result of the operation or (T, F, v). Then we write this as $R :=$ or (T, F, v).

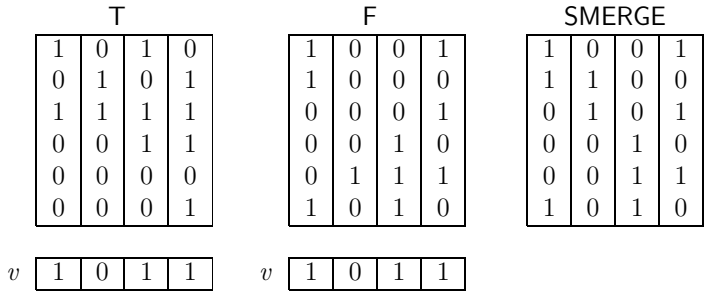

Fig. 3. The use of SMERGE(T, F, v)

Remark 1. Statements of the MCA-machine are defined in the same manner as for Pascal. We will use them later for presenting our procedures.

Following Foster [3], the *time complexity* of an algorithm is measured by counting all elementary operations of the MCA-machine (its *microsteps*) performed in the worst case. It is assumed that each elementary operation for variables of the types *slice*, *word*, and *table* takes one unit of time.

3 IMPLEMENTING THE FIRST GROUP OF RELATIONAL ALGEBRA OPERATIONS ON THE MCA-MACHINE

A relational database is defined as in [20]. Let D_i be a domain, $i = 1, 2, \ldots, k$. Let R denote a relation. It is determined as a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_k$. An element of the relation 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 MCA-machine, any relation is represented as a matrix and each its tuple is allocated to one memory row. Obviously, any relation consists of different tuples. We will assume the entire relation to fit in the hardware matrix of the associative processing unit.

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 operations Product, Join, and Union. These operations assemble a new relation.

To represent the first group of relational algebra operations on the MCA-machine, we first propose a new efficient associative algorithm for the multi-comparand parallel search.

3.1 Performing the Multi-Comparand Search in Parallel

In [1], Falkoff proposed an associative algorithm for selecting rows in a given matrix T that coincide with a given pattern w. This algorithm runs as follows: at every i^{th} step of computation, it saves the matrix T rows whose first i bits are the initial part of the pattern w.

In [15], we presented an implementation of this algorithm on the STAR-machine as procedure MATCH(T, X, w, Z). It determines *positions* of the matrix T rows that coincide with the given pattern w. By means of the slice X, we mark with $'1'$ the matrix T rows that are used for comparison with w. The procedure returns the slice Z, where $Z(i) = '1'$ if and only if ROW$(i, T) = w$ and $X(i) = '1'$ (Figure 4). On the STAR-machine, the procedure MATCH requires $O(k)$ time [15], where k is the number of columns in the matrix T.

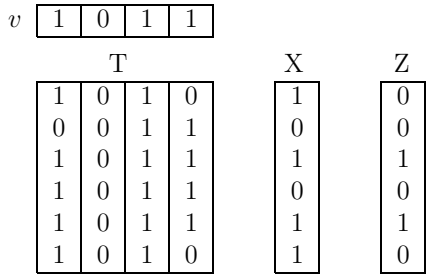

Fig. 4. Testing $v \in T$

Let us explain its implementation on the STAR-machine that performs the bit-serial processing. The procedure MATCH uses an auxiliary slice, say Y. Initially, the resulting slice Z coincides with the given slice X, that is, the rows of T, marked with $'1'$ in the slice X, are candidates for analysis. At every i^{th} step of computation ($i \geq 1$), we first write the i^{th} column of the matrix T into the slice Y. Then we examine the i^{th} bit of the pattern w. If $w(i) = '1'$, we perform the statement $Z := Z\text{and}Y$. Otherwise, to save positions of rows whose i^{th} bit is $'0'$, we fulfil the statement $Z := Z\text{and}(\text{not}Y)$.

Now we consider implementation of the procedure MATCH(T, X, w, Z) on the MCA-machine. It makes use of the following idea. For every $1 \leq i \leq |w|$, the i^{th} bit of every row of the matrix T is simultaneously compared to the i^{th} bit of the 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;
```

```
/* Positions of the bit '0' in the pattern w are marked
with '1' in the row v.  */
2.    SCOPY(A,X,u);
/* Every column of the matrix A saves with '1' positions
   of the matrix T rows being used for comparison with w.  */
3.    B:=not(T,v);
/* We write the negation of every column of the matrix T
   that corresponds to '0' into the given string w.  */
4.    A:=and(A,B,u);
Here, u is a mask for columns of matrices A and B.  */
5.    Z:=and(row,A);
/* We obtain that Z(i) = '1' if ROW(i, A) consists of ones.  */
6.  End;
```

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

Proof. We will 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 our assumption. □

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

Here we generalize the Falkoff algorithm. Let T be a given matrix consisting of n rows and k columns and F be a given matrix of patterns or comparands consisting of m rows and k columns, where $m \leq n$. We will select *in parallel* the matrix T rows that coincide with a given set of m patterns. Our algorithm uses the following idea: at every i^{th} step of computation, by means of ones, we store in parallel m groups of the matrix T rows such that each group stores positions of those rows whose first i bits match the initial part of a concrete pattern.

Now we propose the MultiMatch procedure. It uses the number of bit columns k in the given matrix T and two auxiliary matrices A and B. Note that the number of columns in A and B is equal to the number of patterns m in F.

```
procedure MultiMatch(T: table; X: slice(T); F: table;
  k: integer; var A: table);
/* Every i^th column of the matrix A saves with '1' positions
   of those rows of T that coincide with the i^th pattern of F.  */
var B: table;
  u,w1,w2: word(A);
```

```
   Y: slice(T);
   Z: slice(F);
 1.  Begin SET(w1); SCOPY(A,X,w1);
 2.    for i:=1 to k do
 3.      begin Y:=COL(i,T);
 4.        SCOPY(B,Y,w1);
 5.        Z:=COL(i,F);
 6.        w2:=CONVERT(Z);
 7.        u:=not w2; B:=not (B,u);
 /* The columns of the matrix B marked with '1' in the row u
    are replaced with their negation.  */
 8.        A:=and (A,B,w1);
 9.      end;
10.  End;
```

Proposition 2. Let T be a matrix consisting of n rows and k columns and F be a matrix of patterns consisting of m rows and k columns, where $m \leq n$. Let the selected rows of the matrix T be marked with $'1'$ in the given slice X. Then the procedure MultiMatch(T, X, F, k, A) returns a matrix A consisting of n rows and m columns whose every i^{th} column stores positions of those matrix T rows that coincide with the pattern written in the i^{th} row of the matrix F.

Proof. [Sketch.] We prove this by induction in terms of the number of columns k in the matrix T.

Basis is checked for $k = 1$. Then maximum two patterns $'0'$ and $'1'$ belong to F and $m = 2$. After performing line 2, the given slice X is written into both columns of the matrix A. After fulfilling lines 3–7, we first write the single column of the matrix T into the slice Y. Then we store this slice in both columns of the matrix B. Further, the column of B, that corresponds to the pattern $'0'$, is replaced by $not\, Y$. Therefore after performing line 8, one column of the matrix A stores positions of the matrix T rows that coincide with the pattern $'1'$ and its another column saves positions of rows that coincide with the pattern $'0'$.

Step of induction. Let the assertion be true for $k \geq 1$. We will prove it for $k+1$. To this end, we represent the matrices T and F as $T = T_1T_2$, $F = F_1F_2$, where T_1 consists of the first k columns of T and T_2 is its $(k+1)^{\text{th}}$ column. In the same manner, we determine F_1 and F_2. After performing line 2, the given slice X will be written into $k+1$ columns of the matrix A. By the inductive assumption, the assertion is true for T_1 and F_1, that is, after updating the first k columns of T, every l^{th} column of the matrix A $(1 \leq l \leq m)$ saves with $'1'$ the positions of the matrix T rows whose first k bits match the initial part of the l^{th} pattern. Now, we perform the $(k+1)^{\text{th}}$ iteration. Here, we reason by analogy with the basis. Hence, after performing this iteration, every l^{th} column of the resulting

matrix A saves with $'1'$ the positions of the matrix T rows which coincide with the l^{th} pattern of the matrix F.

□

On the MCA-machine, the MultiMatch procedure takes $O(k)$ time, where k is the number of columns in the matrix T. On the STAR-machine, such an algorithm can be implemented by fulfilling the procedure MATCH for every pattern. Since on the STAR-machine the procedure MATCH takes $O(k)$ time, the procedure MultiMatch requires $O(km)$ time.

Now, we enumerate two properties of the matrix A to be used below.

Property 1. The i^{th} row of the matrix A $(1 \leq i \leq n)$ consists of zeros if and only if the i^{th} row of T does not belong to F.

Property 2. The j^{th} column of the matrix A $(1 \leq j \leq m)$ consists of zeros if and only if the j^{th} pattern from F does not belong to T.

3.2 Applications of the Multi-Comparand Search to the First Group of Relational Algebra Operations

We will consider applications of the multi-comparand search to the following relational algebra operations: Intersection, Difference, Semi-join, Projection, and Division. The resulting relation of these operations is a subset of the argument relations T and F. The corresponding procedures will use a global slice X to select with $'1'$ positions of tuples in the relation T.

The operation Intersection has two argument relations T and F that are drawn from the same domain. The resulting relation of this operation consists of those tuples that belong to T and F.

On the MCA-machine, this operation is implemented as follows.

```
procedure Intersection(T: table; X: slice(T); F: table; k: integer;
  var Y: slice(T));
var A: table;
1.  Begin MultiMatch(T,X,F,k,A);
2.    Y:= or (row,A);
3.    Y:= Y and X;
4.  End;
```

The correctness of this procedure is checked as follows. Since T and F are relations, there is at most a single bit $'1'$ both in every column and in every row of the matrix A (line 1). Therefore, after performing lines 2–3, $Y(i) = '1'$ if and only if the i^{th} row of T is a tuple of the relation F.

Consider the operation Difference of relations T and F. The resulting relation consists of those tuples of T that do not belong to F.

```
procedure Difference(T: table; X: slice(T); F: table; k: integer;
  var Y: slice(T));
var Z: slice(T);
Begin Intersection(T,X,F,k,Z);
  Y:= X and (not Z);
End;
```

Consider the operation Semi-join of relations $T(T1, T2)$ and F. We assume that the attribute $T2$ of the relation T and the relation F are drawn from the same domain. The resulting relation of the operation Semi-join consists of those tuples $\mathrm{ROW}(i, T)$, for which there exists such j that $\mathrm{ROW}(i, T2) = \mathrm{ROW}(j, F)$. Positions of the resulting tuples are marked with $'1'$ in the slice Y.

```
procedure Semi-join(T(T1,T2): table; X: slice(T); F: table;
  k: integer; var Y: slice(T));
/* Here, k is the number of columns in the relation F.  */
Begin Intersection(T2,X,F,k,Y);
End;
```

It should be noted that the attribute $T2$ is not a relation. Therefore a tuple of F may coincide with a few rows of $T2$. However, the resulting tuples form a relation as a subset of T.

The correctness of the procedures Difference and Semi-join is evident.

Let the relation T have two attributes $T1$ and $T2$. Consider the operation Projection2. The resulting relation of this operation consists of the tuples from the relation T that have only different values of the second attribute. The operation Projection1 is determined in the same manner.

```
procedure Projection2(T(T1,T2): table; X: slice(T); k: integer;
  var Y: slice(T));
/* Here, k is the number of columns in the attribute T2.  */
var A: table;
1.  Begin MultiMatch(T2,X,T2,k,A);
2.    FRST(col,A);
3.    Y:= or (row,A);
4.  End;
```

Let us justify the correctness of this procedure. After performing line 1, every i^{th} column of the matrix A $(1 \leq i \leq k)$ stores with $'1'$ positions of rows of $T2$ that coincide with the pattern $\mathrm{ROW}(i, T2)$. Note that $T2$ is not a relation in the general case. However, after performing line 2, in every i^{th} column of A, a single representative is saved. Notice that after performing line 2 the matrix A may include some identical columns. Nevertheless, after performing line 3, the slice Y saves the positions of tuples from the relation T having different values of the attribute $T2$.

Now, we consider the operation Division. Let the relation $T = (T1, T2)$ and the relation F be given. Let the relation T be a divident and the relation F be

a divisor. Let the values of $T2$ and F be drawn from the same domain. Then $T \div F = \{u \in T1 \,/\, \forall v \in F,\, uv \in T\}$.

The implementation of the operation Division is complicated [18]. However, for the considered model, we can propose a clear implementation of this operation.

We first explain the main idea of implementing the operation Division on the MCA-machine. Let $F = \{v_1, v_2, \ldots, v_k\}$. The procedure Division constructs the following sequence of embedded sets:

$$E_1 = \{\alpha \in T1 \,/\, \alpha v_1 \in T\};$$

$$E_2 = \{\beta \in E_1 \,/\, \beta v_2 \in T\};$$

$$\vdots$$

$$E_k = \{\delta \in E_{k-1} \,/\, \delta v_k \in T\}.$$

It can be easily seen that $E_k = T \div F$ by construction.

On the MCA-machine, the procedure Division returns a slice Z, where the positions of rows from the attribute $T1$ belonging to the result are marked with $'1'$.

```
procedure Division(T(T1,T2):table; X:slice(T); F:table;
  Y:slice(F); k:integer; var Z:slice(T));
/* Here, k is the number of columns in the attribute T1.  */
var M,Q:slice(T);
  P:slice(F);
  w:word(F);
  v1,v2:word(T1);
  C:table;
  i:integer;
1.  Begin P:=Y; M:=X;
2.    SET(v1); CLR(v2);
3.    SMERGE(T1,C,v1);
/* The matrix C is a copy of the attribute T1.  */
4.    RCOPY(C,v2,not X);
/* We write v2 into the rows of C marked with '0' in the slice X.  */
5.    while SOME(P) do
6.      begin i:=STEP(P); w:=ROW(i,F);
7.        MATCH(T2,M,w,Q);
8.        Intersection(T1,Q,C,k,M);
9.        RCOPY(C,v2,not M);
/* We write v2 into the rows of C marked with '0' in the slice M.  */
10.     end;
11.     Z:=M;
12.  End;
```

Remark 2. The procedure Intersection does not use a slice for the second relation. To obtain a sequence of the embedded sets, we perform line 9 after every application of the procedure Intersection.

The correctness of the procedure Division is checked by induction in terms of the number of tuples in the relation F.

Let us evaluate the time complexity of the considered procedures. On the MCA-machine, procedures Intersection, Difference, Semi-join, and Projection2 take $O(k)$ time each, where k is the number of columns in the corresponding relation. On the STAR-machine, these procedures take $O(kn)$ time each [13], where n is the number of tuples in the relation T and k is the number of columns in T or in F.

On the MCA-machine, the procedure Division takes $O(km)$ time, where m is the number of tuples in the relation F and k is the number of columns in the attribute $T1$. On the STAR-machine, this procedure takes $O(kmn)$ time [13], where n is the number of tuples in the relation T and the parameters m and k were determined above.

It should be noted that the implementation of every operation from the first group on the MCA-machine and on a modified version of the STAR-machine joined with the hardware support called the set intersection processor [12] takes the same time.

4 IMPLEMENTING THE SECOND GROUP OF RELATIONAL ALGEBRA OPERATIONS ON THE MCA-MACHINE

In this section, we consider the implementation of the operations Product, Join, and Union. We first propose a new associative algorithm for implementing the operation Product. Then we consider the implementation of the operation Join that is based on the operation Product. Finally, we consider the implementation of the operation Union.

4.1 Implementing the Operation Product

The operation Product is defined as follows. Let T and F be argument relations for the operation Product. The resulting relation $R(R1, R2)$ is obtained as concatenation of all combinations of the relations T and F.

Let us explain the main idea of implementing the operation Product on the MCA-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 the operations MASK, RCOPY, and SHIFT. 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 the operations RCOPY and SHIFT, 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.        SHIFT(Z,down,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 SHIFT(Z1,down,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.       SHIFT(Z2,down,1);
19.     end;
20. End;
```

Proposition 3. 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 execution of 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 the 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 in $R1$. Hence, s copies of v_1 are also written in the attribute $R1$. It is easy to check that after performing lines 8–13, the slice $Z2$ saves the positions of rows in the attribute $R2$, where p copies of the first tuple from F must 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 in 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 k^{th} tuple in T and v_2 be the i^{th} tuple in F. Then the tuple v_1v_2 has been written in the $(i + (k-1)s)^{\text{th}}$ row of the resulting relation R. This cotradicts our assumption. □

Let us evaluate the time complexity of the procedure Product. Obviously, on the MCA-machine, it takes $O(p + s)$ time.

In [12], we proposed an 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 to execute the matrix compaction by means of the vertical processing. As shown above, the efficient implementation of the operation Product on the MCA-machine does not use the compaction of the relation F. We avoid the compaction of a matrix due to the use of the operations SHIFT and RCOPY.

4.2 Implementing the Operation Join

There are different versions of the Join operations. We will consider the case of the operation Join [2], where the result does not contain the joining attributes and is obtained by concatenating the tuples of two argument relations that satisfy some specified condition.

Let $T(T1, T2)$ and $F(F1, F2)$ be two argument relations for the operation Join. Let the attributes $T2$ and $F2$ be drawn from the same domain. We assume that the joining attributes are $T2$ and $F2$ and the condition for joining is their equality. More precisely, the operation Join performs concatenation in every group of the attributes $T1$ and $F1$, for which the corresponding values of the attributes $T2$ and $F2$ are equal.

Let us explain the main idea of implementing the operation Join on the MCA-machine. Let $C(C1, C2)$ be the resulting relation of the operation Join. Let the relation T consist of p tuples and the relation F consist of s tuples. Although the

cardinality of the relation C is a priori unknown, it is not greater than ps, that is, the cardinality of the resulting relation of the operation Product. Therefore, the attributes $C1$ and $C2$ will consist of ps rows each. Initially, we set zeros in the attributes $C1$ and $C2$. For every current tuple v in the attribute $T2$, we determine all its occurrences both in $T2$ and in $F2$. If v belongs to $F2$, we carry out the procedure Product between the corresponding rows of the attributes $T1$ and $F1$ and include the result into the corresponding rows of $C1$ and $C2$. Otherwise, we analyze the next tuple in $T2$. We do this with the use of basic operations of the MCA-machine and the procedures MATCH and Product.

```
procedure Join(T(T1,T2): table; F(F1,F2): table; Y: slice(F);
  var X: slice(T); var C(C1,C2): table);
var X1: slice(T);
  Y1: slice(F);
  Z: slice(C);
  v: word(T2);
  v1: word(T1);
  v2: word(F1);
  i,s1,r1,t: integer;
  E(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,T2);
7.        MATCH(T2,X,v,X1);
/* Positions of the attribute T2 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 the positions of the attribute
   T2 rows that coincide with v. */
9.        MATCH(F2,Y,v,Y1);
/* We mark with '1' in the slice Y1 the positions
   of the attribute F2 rows that coincide with v. */
10.       if SOME(Y1) then
11.         begin r1:=NUMB(X1); s1:=NUMB(Y1);
12.           Product(T1,F1,X1,Y1,E(E1,E2));
13.           SHIFT(E1,down,t);
14.           SHIFT(E2,down,t);
15.           C1:= or (C1,E1,v1);
16.           C2:= or (C2,E2,v2);
/* We include the result of shifting the matrix E1
```

```
    (respectively, E2) into the attribute C1 (respectively, C2). */
17.              t := t + r1s1;
18.           end;
19.       end;
20.   End;
```

Proposition 4. Let two argument relations $T(T1, T2)$ and $F(F1, F2)$ be given. Let a slice X save the positions of tuples that belong to T, and a slice Y save the positions of tuples that belong to F. Let the attributes $T2$ and $F2$ be drawn from the same domain. Then the procedure Join returns the concatenation of those rows from the attributes $T1$ and $F1$, for which the corresponding values of $T2$ and $F2$ are equal.

Proof. [Sketch.] We prove this by induction in terms of the number of different tuples l that belong to the attributes $T2$ and $F2$.

Basis is checked for $l = 1$, that is, only a single tuple belongs to $T2$ and $F2$. 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 $T2$. Then by means of the slice $X1$, we save the positions of all occurrences of v in $T2$ and delete them from the slice X. Without loss of generality, we assume v to be the single tuple that belongs to $T2$ and $F2$. Therefore after performing lines 9–10, by means of the slice $Y1$, we save the positions of all occurrences of v in the attribute $F2$. Since $Y1 \neq \emptyset$[1], we determine the number of tuples $r1$ in the attribute $T1$ and the number of tuples $s1$ in the attribute $F1$ (line 11). Since T is a relation, the rows in the attribute $T1$ that correspond to the same tuple v in $T2$ are different. The same we have for the relation F. Moreover, the attributes $T1$ and $F1$ 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 $X \neq \emptyset$, we select with $'1'$ the positions of all occurrences of the next tuple in $T2$ and delete them from X as shown above. We continue this process while $X \neq \emptyset$. After that we go to the end of the procedure.

Step of induction. Let the assertion be true when l ($l \geq 1$) different tuples belong both to the attribute $T2$ and the attribute $F2$. We prove the assertion for the case when $l + 1$ different tuples belong to $T2$ and $F2$. By the inductive assumption after selecting the first l different tuples, their positions are selected with $'1'$ 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 $F2$, we carry out the procedure Product which is applied to the corresponding attributes $T1$ and $F1$. Since a new value for t is computed after every execution of the

[1] The notation $Y1 \neq \emptyset$ denotes that there is at least a single component $'1'$ in the slice $Y1$.

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 $T2$ and $F2$. As soon as we select the positions of all occurrences of this tuple (lines 6–10), we perform the procedure Product for the corresponding rows of the attributes $T1$ and $F1$ 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 in the attributes $C1$ and $C2$. As soon as the slice $X = \emptyset$, we run to the end of the procedure.

□

Let k be the number of different tuples that belong both to $T2$ and to $F2$. Let p_i and s_i denote the number of different occurrences of the i^{th} tuple that belongs to the attributes $T2$ and $F2$, respectively. Then the procedure Join takes $O(\sum_{i=1}^{k}(p_i + s_i))$ time.

4.3 Implementing the Operation Union

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.

To implement the operation Union on the MCA-machine, we use the procedure Difference$(F, Y, T, k, Y1)$. It returns a slice $Y1$ that saves the positions of the relation F tuples not belonging to the relation T. As shown in the previous section, the procedure Difference takes $O(k)$ time.

Let us explain the main idea of implementing the operation Union on the MCA-machine. We first copy the relation T into the relation P. Then we perform the procedure Difference and save the positions of the relation F tuples that do not belong to the relation T. Further we include into P every tuple w from the relation F that does not belong to the relation T. Moreover, the *position* of the tuple w 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;
1.    Begin SET(v); Z:=X;
2.      SMERGE(T,P,v);
  /* We copy the relation T into P. */
3.      Difference(F,Y,T,k,Y1);
  /* The slice Y1 saves the positions of the relation F rows
     that do not belong to T. */
4.      while SOME(Y1) do
```

```
5.        begin i:=STEP(Y1); w:=ROW(i,F);
6.          Z1:=not Z; j:=FND(Z1);
7.          ROW(j,P):=w; Z(j):='1';
8.        end;
9.    End;
```

Proposition 5. Let two argument relations T and F be given. Let a slice X save the positions of tuples that belong to T, and a slice Y save the positions of tuples that belong to F. Then the procedure Union returns the relation P that consists of different tuples from the relations T and F, and a slice Z that saves the positions of the tuples from P.

The correctness of the procedure Union is proved by induction in terms of 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 MCA-machine takes one unit of time and the procedure Difference takes $O(k)$ time. 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.

5 CONCLUSIONS

In this paper, we have proposed a multi-comparand associative machine and its application to parallel implementation of classical relational algebra operations. We first consider an efficient associative algorithm for performing the multi-comparand search in parallel. On the MCA-machine, this algorithm is implemented as procedure MultiMatch whose correctness is proved. We have shown that this procedure takes $O(k)$ time, where k is the number of columns in the given matrix. On the STAR-machine, such an algorithm takes $O(km)$ time, where m is the number of patterns. We have also proposed applications of the multi-comparand search to carry out the first group of relational algebra operations: Intersection, Difference, Semi-join, Projection, and Division. On the MCA-machine, these operations are represented as corresponding procedures and their correctness is justified. We have shown that the procedure Division takes $O(km)$ time, where m is the number of tuples in the divisor and k is the number of columns in the attribute $T1$ of the divident $T(T1, T2)$. Other procedures take $O(k)$ time each, where k is the number of columns in the corresponding relation. We have shown that all these estimations are optimal. We have also proposed efficient associative algorithms for implementing the second group of relational algebra operations. These algorithms are given as corresponding procedures for the MCA-machine. We have proved their correctness and evaluated time complexity. We have obtained clear and simple implementations of the proposed algorithms due to the access data by contents and the use of data parallelism. We achieve the fastest processing because the structure of input data and applied

algorithms correspond in the best way to the structure of the model. It might be well to use the MCA-machine for simulating and specifying different properties of new associative systems. The efficient non-trivial procedure Multi-Match may be of advantage when different associative algorithms use the fast parallel search.

We are planning to study some applications of the MCA-machine to the image processing.

REFERENCES

[1] Falkoff, A. D.: Algorithms for Parallel-Search Memories. J. of the ACM, Vol. 9, 1962, pp. 448–510.

[2] Fernstrom, C.—Kruzela, J.—Svensson, B.: LUCAS Associative Array Processor. Design, Programming and Application Studies, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 216, 1986.

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

[4] Irakliotis, L. J.–Betzos, G. A.–Mitkas, P. A.: Optical associative processing. In: A. Krikelis, C. C. Weems (Eds.): Associative Processing and Processors, IEEE Computer Society, 1997, pp. 155–178.

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

[6] Kokosiński, Z.: An Associative Processor for Multi-Comparand Parallel Searching and its Selected Applications. In: Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA'97, Las Vegas, USA, 1997, pp. 1434–1442.

[7] Kokosiński, Z.—Sikora, W.: An FPGA Implementation of Multi-Comparand Multi-Search Associative Processor. In: Proc. 12th Int. Conf. FPŁ2002, Lect. Notes in Comp. Sci., Springer-Verlag, Berlin, Vol. 2438, 2002, pp. 826–835.

[8] Louri, A.—Hatch, J. A.: An Optical Associative Parallel Processor for High-Speed Database Processing. Computer, Vol. 27, 1994, pp. 65–72.

[9] Muraszkiewicz, M. R.: Cellular Array Architecture for Relational Database Implementation. Future Generations Computer Systems, Vol. 4, 1988, pp. 31–38.

[10] Nepomniaschaya, A. S.: Language STAR for Associative and Parallel Computation with Vertical Data Processing. In: Proc. of the Intern. Conf. on Parallel Computing Technologies, World Scientiic, Singapore, 1991, pp. 258–265.

[11] Nepomniaschaya, A. S.–Kokosiński, Z.: Associative Graph Processor and its Properties. In: Proc. of the Intern. Conf. PARELEC'2004, IEEE Computer Society, Dresden, Germany, 2004, pp. 297–302.

[12] Nepomniaschaya, A. S.–Fet, Y. L.: Investigation of Some Hardware Accelerators for Relational Algebra Operations. In: Proc. of the First Aizu Intern. Symp. on Parallel Algorithms/Architecture Synthesis, IEEE Computer Society, Aizu-Wakamatsu, Fukushima, Japan, 1995, pp. 308–314.

[13] Nepomniaschaya, A. S.: A Language STAR for Associative and Bit-Serial Processors and Its Application to Relational Algebra. In: Bulletin of the Novosibirsk Computing Center, Series: Computer Science, Issue 1, NCC Publisher, 1993, pp. 23–36.

[14] Nepomniaschaya, A. S.: An Associative Version of the Bellman-Ford Algorithm for Finding the Shortest Paths in Directed Graphs. In: Proceedings of the 6th Intern. Conf. PaCT-2001. Lect. Notes in Comp. Sci., Springer-Verlag, Berlin, 2001, Vol. 2127, pp. 285–292.

[15] Nepomniaschaya, A. S.—Dvoskina, M. A.: A Simple Implementation of Dijkstra's Shortest Path Algorithm on Associative Parallel Processors. Fundamenta Informaticae, IOS Press, Amsterdam, Vol. 43, 2000, pp. 227–243.

[16] Nepomniaschaya, A. S.: Efficient Implementation of Edmonds' Algorithm for Finding Optimum Branchings on Associative Parallel Processors. In: Proc. of the Eighth Intern. Conf. on Parallel and Distributed Systems (ICPADS '01), IEEE Computer Society, KyongJu City, Korea, 2001, pp. 3–8.

[17] Nepomniaschaya, A. S.: Efficient Update of Tree Paths on Associative Systems with Bit-Parallel Processing. In: Bulletin of the Novosibirsk Computing Center, Series: Computer Science, Issue 23, NCC Publisher, 2005, pp. 71–83.

[18] Ozkarahan, E.: Database Machines and Database Management. Prentice-Hall, Inc. 1986.

[19] Parhami, B.: Search and Data Selection Algorithms for Associative Processors. In: A. Krikelis, C. C. Weems (Eds.), Associative Processing and Processors, IEEE Computer Society, 1997, pp. 10–25.

[20] Ullman, J. D.: Principles of Database Systems. Computer Science Press, 1980.

Anna Shmilevna Nepomniaschaya graduated from the Chernovitsky State University in 1967 and worked in the Novosibirsk Institute of Mathematics (Siberian Division of the USSR Academy of Sciences). In 1981, she received her Ph. D. degree in computer science from the Novosibirsk Computing Center (the new name is Institute of Computational Mathematics and Mathematical Geophysics, Siberian Division of the Russian Academy of Sciences). Now, she is a Senior Researcher in the Laboratory of Parallel Algorithms and Structures of the Institute of Computational Mathematics and Mathematical Geophysics. She published 108 papers in automata theory, theory of formal grammars and languages, theory of parallel algorithms and parallel processing. Her current research interests include associative processing in fine-grained parallel systems for such applications as graph algorithms and relational databases, and techniques for specification and analysis of associative parallel processors.