Bull. Nov. Comp. Center, Comp. Science, 25 (2006), 37–48
© 2006 NCC Publisher

An efficient associative algorithm for multi-comparand parallel searching and its applications

A.S. Nepomniaschaya

1. Introduction

Associative (content addressable) parallel processors of the SIMD type 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 [12], 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 search problems in computational geometry were considered. In [10], a formal model of 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 [6] due to the presence of built-in operations designed for associative graph algorithms. We will show that this model can efficiently support classical operations in relational databases. In [2, 8, 11], relational database processing on 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 parallel relational database processing by fully exploiting the parallelism of optics. In [4], different optical and optoelectronic architectures for image processing and relational database associative processing were reviewed.

In this paper, we propose a new associative algorithm for multi-comparand searching and its implementation on the AG-machine. Then we consider applications of this algorithm to representing the classical operations of relational algebra. Algorithms are given as the corresponding procedures for the AG-machine. We prove their correctness and evaluate time complexity.

2. 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 the *typical operations* for associative systems first presented in Staran [3] 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, the rows are numbered from top to bottom and the 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 matrix of input binary data. Each column in the matrix of PEs can be regarded as a vertical register that maintains the entire column of a table. Our 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 (the uppermost) component $'1'$ in the slice Y and sets to $'0'$ its other components.

The *new operation* convert(Y) returns a row whose every i-th component (bit) coincides with $Y(i)$. It will be 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, it was not included before in the STAR-machine in view of absence of its application.

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

SET(Y)	sets all components of 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 $'0'$.

In the usual way, we introduce predicates ZERO(Y) and SOME(Y) and the bitwise Boolean operations X *and* Y, X *or* Y, *not* Y, and 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 i-th row of the matrix T;
COL(i, T)	returns the i-th 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 hardware.

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

The operation SCOPY(T, X, v) *simultaneously* writes the given slice X in those columns of the given matrix T which are marked by ones in the given comparand v.

The operation *not*(T, v) *simultaneously* replaces the columns of the given matrix T, marked by ones 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* fulfils the operation FRST for every row of the matrix T and writes the result in T.

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

The operation *or*(row, T) *simultaneously* performs 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 *or*(col, T) *simultaneously* performs 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 determine the *second group* of new operations.

The operation SMERGE(T, F, v) *simultaneously* writes the columns of the given matrix F, that are marked by ones in the comparand v, in the corresponding columns of the result matrix T. If the comparand v consists of ones, 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 by ones 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.

We will employ the basic procedure MATCH(T, X, w, Z) [9]. It determines *positions* of the matrix T rows that coincide with the given pattern w. By means of the slice X, we mark by ones the matrix T rows being used for comparison with w. The procedure returns the slice Z, where $Z(i) =' 1'$ if and only if ROW$(i, T) = w$ and $X(i) =' 1'$.

In [10], we have proposed an efficient implementation of this procedure on the AG-machine that takes $O(1)$ time. Notice that on the STAR-machine, it requires $O(k)$ time [9], where k is the number of columns in the matrix T.

3. Performing multi-comparand searching in parallel

In [1], Falkoff proposed an associative algorithm for selecting rows in the given matrix T that coincide with the 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 [9], we suggested an implementation of this algorithm on the STAR-machine as procedure MATCH(T, X, w, Z).

Here, we generalize Falkoff's 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 the given set of m patterns. Our algorithm uses the following idea: at every i-th step of computation, we store in parallel m groups of the matrix T rows such that each group stores positions of those rows whose first i bits are the initial part of a concrete pattern.

Before presenting the procedure MultiMatch, let us explain the implementation of the procedure MATCH(T, X, w, Z) on the STAR-machine. Initially, the result slice Z coincides with the given global slice X, that is, rows of T, marked by $'1'$ 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 in a slice Y. Then we examine the i-th bit of the pattern w. If $w(i) =' 1'$, we perform the statement $Z := Z\ and\ Y$. Otherwise, to save positions of rows whose i-th bit is $'0'$, we fulfil the statement $Z := Z\ and\ (not\,Y)$.

Now, we propose the following procedure:

```
    procedure MultiMatch(T: table; X: slice(T); F: table;
                         k: integer; var A: table);
    /* Every i-th column of the matrix A saves by '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);
    /* The current column of T is written to the matrix B. */
 5.       Z:=COL(i,F);
 6.       w2:=convert(Z);
 7.       u:= not w2; B:= not(B,u);
    /* The columns of the matrix B marked by '1' in the row u are replaced
       with their negation. */
 8.       A:=and(A,B,w1);
 9.     end;
10. End;
```

Remark 2. The number of columns in matrices A and B is equal to the number of rows in F.

Correctness of the MultiMatch procedure is established by means of the following

Theorem. *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 by $'1'$ 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. We prove this by induction on 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 1, the given slice X is written in both columns of the matrix A. After fulfilling lines 3–7, we first write the single column of the matrix T in the slice Y. Then we store this slice in both columns of the matrix B. Further, the column of B, that correspond 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 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 $(k+1)$-th column. In the same manner, we determine F_1 and F_2. After performing line 1, the given slice X will be written in $k + 1$ columns of the matrix A. By 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 by $'1'$ positions of the matrix T rows which have the first k bits of the l-th pattern as their initial part. Now, we perform the $(k+1)$-th iteration. Here, we reason by analogy with the basis. Hence, after performing this iteration, every l-th column of the result matrix A saves by $'1'$ positions of the matrix T rows which coincide with the l-th pattern of F.
□

Let us evaluate time complexity of the procedure MultiMatch. We obtain that on the AG-machine, it 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 in turn. Since the procedure MATCH takes $O(k)$ time on the STAR-machine, the procedure MultiMatch requires $O(km)$ time.

Now, we enumerate two properties of the matrix A being 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.

Remark 3. In the procedure MultiMatch, all patterns of the matrix F are analyzed. In a general case, a global slice (say, L) for the matrix F may be used. To take into account this case, we perform the following statements immediately after line 8:

```
    v:=convert(L); u:= not v; CLR(Z); SCOPY(A,Z,u).
```

Therefore the rows of F marked by $'0'$ in the slice L will correspond to columns consisting of zeros in the result matrix A.

The procedure MultiMatch can be generalized as follows. Let T and F be two matrices. Let k be the number of columns in T, r be the number of columns in F, and $j = k - r$. For every row v in F, we want to check whether there exists such a row w in T that v is the tail of w, that is, $v = w(j+1)w(j+2)\dots w(k)$. Such a checking can be done *in parallel* for all rows from F by means of the procedure MultiSelect. The procedure head has the following form:

```
procedure MultiSelect(T: table; X: slice(T); F: table;
                      j,k: integer; var A: table);
```

Every i-th column of the matrix A will store positions of those matrix T rows whose tail coincides with the i-th row of the matrix F.

The procedure MultiSelect runs by analogy with MultiMatch. We have only to change the following two lines in MultiMatch:

Line 2: `for i:=j+1 to k do`
Line 5: `Z:=COL(i-j,F);`

4. Applications of multi-comparand searching to relational algebra

A relational database is defined as in [13]. 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.

We will consider applications of multi-comparand searching to the following relational algebra operations: Intersection, Difference, Semi-join, Projection, and Division. The result relation for these operations is a subset of the argument relations T and F. The corresponding procedures will use a global slice X to select by ones the *positions* of tuples in the relation T.

The Intersection operation has two argument relations T and F. The result relation consists of those tuples that belong to T and F.

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

```
procedure Inters(T: table; X: slice(T); F: table; k: integer;
                 var Y: slice(T));
var A: table;
Begin MultiMatch(T,X,F,k,A);
  Y:=or(row,A);
  Y:=Y and X;
End;
```

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 result relation consists of those tuples of T that do not belong to F.

```
procedure Differ(T: table; X: slice(T); F: table; k: integer;
                 var Y: slice(T));
var Z: slice(T);
Begin Inters(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 result relation consists of those tuples ROW(i, T) for which there exists such j that ROW$(i, T2)$ = ROW(j, F). Positions of the result tuples are marked by $'1'$ 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 Inters(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 result tuples form a relation as a subset of T.

Correctness of procedures Differ and Semi-join is evident.

Consider the operation Project2 of the relation T having two attributes $T1$ and $T2$. The result relation consists of the tuples from the relation T having only different values of the second attribute. The operation Projection1 is determined in the same manner.

```
procedure Project2(T(T1,T2): table; X: slice(T); k: integer;
                   var Y: slice(T));
/* Here, k is the number of columns in the attribute T2. */
1. Begin MultiMatch(T2,X,T2,k,A);
2.   FRST(col,A);
3.   Y:=or(row,A);
4. End;
```

Let us justify correctness of this procedure. After performing line 1, every i-th column of the matrix A $(1 \le i \le k)$ stores by $'1'$ positions of rows of $T2$ that coincide with the pattern ROW$(i, T2)$. Note that $T2$ is not a relation in 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 positions of tuples from the relation T having different values of the attribute $T2$.

Now, we consider the Division operation. Let the relation T be a dividend having attributes $T1$ and $T2$ and the relation F be a divisor. Let the values of $T2$ and F are drawn from the same domain. Then $T\ div\ F = \{u \in T1 \mid \forall v \in F, uv \in T\}$. On the AG-machine, this operation is implemented by analogy with the STAR-machine [9].

```
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 L,M,Q: slice(T); P: slice(F);
    w: word(F); i: integer;
Begin L:=X; P:=Y;
  while SOME(P) do
    begin i:=STEP(P); w:=ROW(i,F);
      MATCH(T2,L,w,Q);
      Inters(T1,Q,T1,L,k,M);
      L:=M;
    end;
  Z:=M;
End;
```

Correctness of the procedure Division is checked by induction on the number of tuples in the relation F.

Let us evaluate time complexity of the considered procedures. We obtain that on the AG-machine, procedures Inters, Differ, Semi-join, and Project2 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 [9], 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 AG-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 [9], where n is the number of tuples in the relation T and parameters m and k have been determined above.

It should be noted that more complicated operations of the relational algebra Product and Join assemble a new relation. However, these operations do not use the multi-comparand searching.

5. Application of multi-comparand searching to finding the set inclusion

Here, we propose an efficient implementation of finding the set inclusion on the AG-machine. In [5], a specification of this task for a specialized multi-comparand associative processor was presented.

Let two sets of integers $T = \{x_1, x_2, \ldots, x_n\}$ and $F = \{y_1, y_2, \ldots, y_m\}$, where $m \leq n$, be given. We have to check whether for every element $x_i \in T$ there exists an element $y_j \in F$ such that $x_i = y_j$, that is, whether $T \subseteq F$.

To this end, we propose the following procedure:

```
procedure Subset(T: table; F: table; k: integer;
                 var res: boolean);
var A: table; X,Y,Z: slice(T);
1. Begin SET(X); res:=true;
2.   MultiMatch(T,X,F,k,A);
/* Every i-th row of the matrix A stores positions of the matrix T
   rows that coincide with the i-th pattern of F. */
3.   Y:=or(row,A);
/* Y(i) = '0' if and only if the i-th row of T does not belong to F. */
4.   Z:=not Y;
/* We check whether there is at least one bit '0' in the slice Y. */
5.   if SOME(Z) then res:=false;
6. End;
```

Correctness of this procedure is checked by contradiction.

Let the procedure Subset return the **true** value, that is, $T \subseteq F$. However, let there exist such an l-th row of T that doesn't belong to F. We will prove that this assumption cotradicts to performing our procedure. Really, in this case, the l-th row of the matrix A consists of zeros in view of Property 1. Therefore after performing lines 3–5, the procedure Subset returns the value **false**. This contradicts to the given condition.

Remark 4. In the same manner, we can also check whether $F \subseteq T$. In this case, we will use two variables v and w of the type **word** instead of slices Y and Z. In view of Property 2, we have to check whether there is a column in the matrix A that consists of zeros.

It is obvious that the procedure Subset takes $O(k)$ time.

6. Conclusions

In this paper, we have shown how to efficiently implement the classical operations of the relational algebra using the associative graph machine. To this end, we first propose a new associative algorithm for performing multi-comparand searching in parallel. On the AG-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 T. Note that on the bit-serial associative processor such a procedure takes $O(km)$ time, where m is the number of patterns in the relation F. We also propose applications of multi-comparand searching to perform the following classical operations of the relational algebra: Intersection, Difference, Semi-join, Projection, and Division. These operations are represented as the corresponding procedures on the AG-machine 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. Moreover, we have proposed an efficient implementation on the AG-machine of finding the set inclusion.

We are planning to study an application of the AG-machine to performing the image processing.

References

[1] Falkoff A.D. Algorithms for parallel-search memories // J. ACM. — 1962. — Vol. 9, No. 10. — P. 448–510.

[2] Fernstrom C., Kruzela J., Svensson B. LUCAS Associative Array Processor. Design, programming and Application Studies. — Berlin: Springer, 1986. — (Lect. Notes in Comp. Sci.; 216).

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

[4] 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.

[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 multi-search associative processor // Proc. 12-th Int. Conf. FPL'2002. — Berlin: Springer, 2002. — P. 826–835. — (Lect. Notes in Comp. Sci.; 2438).

[7] Louri A., Hatch Jr. 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.Sh. A language STAR for associative and bit–serial processors and its application to relational algebra. // Bull. Novosibirsk Comp. Center. Ser. Computer Science. — Novosibirsk, 1993. — Iss. 1. — P. 23–36.

[10] Nepomniaschaya A.S., Kokosiński Z. Associative graph processor and its properties // Proc. of the Intern. Conf. PARELEC'2004. Dresden, Germany / 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] 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.

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