Bull. Nov. Comp. Center, Comp. Science, 29 (2009), 63–77
© 2009 NCC Publisher

Basic associative parallel algorithms for vertical processing systems

A. S. Nepomniaschaya

Abstract. In this paper, by means of an abstract model of the SIMD type with vertical data processing (the STAR–machine), we present basic associative parallel algorithms. These algorithms are represented as the corresponding procedures for the STAR–machine, whose correctness is justified and the time complexity is evaluated. We also propose a new version of the language STAR.

1. Introduction

Asociative processing is one of the most interesting topics in computer science. Remarkable advances in microelectronics have greatly reduced the implementation cost of practical associative systems and made possible to implement new systems formerly unrealized because of technological restrictions [5]. Asociative processing is a totally different way of storing, manipulating, and retreiving data as compared to traditional computation techniques. Its main feature is implementation of a more intelligent memory. In [21], there is an analysis of different design aspects which influence the configuration, performance, and use of new SIMD systems. However, in spite of a large variety of different SIMD systems which could be implemented, one combination of factors is brought together in fine-grained systems, the other in course-grained systems. Of special interest is the class of associative (content addressable) parallel processors that belong to fine-grained SIMD systems with bit–serial (vertical) processing and simple single–bit processing elements [3]. Associative parallel processors offer distinctive advantages over other parallel systems, such as data parallelism at the base level, the use of two–dimensional tables as the basic data structure, massively parallel search by contents, and processing of unordered data. Moreover, basic operations of search and arithmetical operations take time which is proportional to the number of bit columns in a field, but not to the number of data items being searched [18–20].

In [1], Falkoff first suggested some associative algorithms for the content addressable memory. However, these algorithms were unknown because it was impossible to explain them in terms of sequential computers. In [19], Potter enumerated basic algorithms and functions used in the model ASC. However, they cannot be represented in the ASC language. In [2], basic instructions applied to the associative memory of the associative array processor LUCAS are classified according to the types of operands and the types of results. They also cannot be represented by means of the language Pascal/L for LUCAS.

To present basic associative algorithms, we use a model of the SIMD type with vertical data processing (the STAR–machine) that simulates the run of the associative architecture at the micro level. The main goal of the paper is to present a new version of the language STAR and to propose the basic associative parallel algorithms. We will represent these algorithms as the corresponding procedures for the STAR–machine. On one hand, basic associative algorithms are used for designing different associative algorithms for different applications that employ a two–dimensional topology, such as graph algorithms [10–15], relational databases [16, 18], knowledge bases, expert systems. On the other hand, they can be used both for testing the run of the associative architecture and for specifying new associative systems. Moreover, the basic associative algorithms allow one to better understand why the associative architecture is very attractive.

2. Model of associative parallel machine

Let us recall our model which is based on a Staran–like associative parallel processor [4], [6]. In [9], we have compared different models for vertical processing systems. We define our model as an abstract STAR–machine of the SIMD type with vertical data processing [7]. It consists of the following components (Figure 1):

– a sequential control unit (CU), where programs and scalar constants are stored;

– an associative processing unit consisting of p single–bit processing elements (PEs);

– a matrix memory for the associative processing unit.

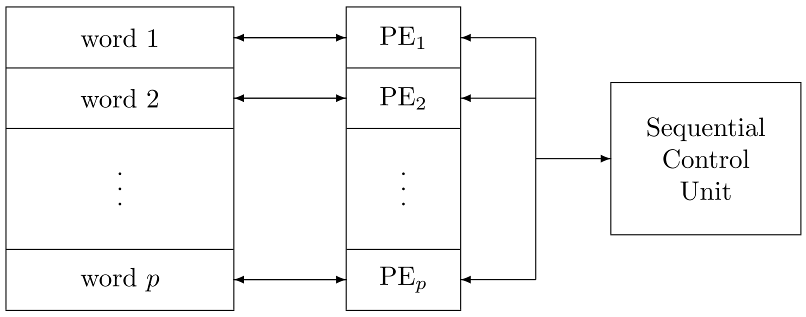

Figure 1. The STAR-machine

The CU passes an instruction to all PEs in one unit of time. All active PEs execute it in parallel while inactive PEs do not. Activation of a PE depends on the data.

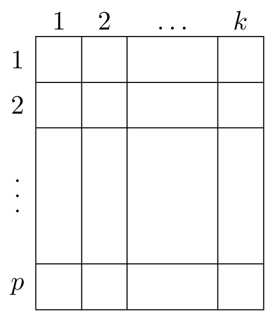

Figure 2. Data array

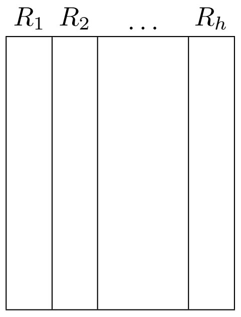

Figure 3. Associative processing unit

Input binary data are loaded to the matrix memory in the form of two–dimensional tables, where each data item occupies an individual row and it is updated by a dedicated processing element (Figure 2). We assume that the number of PEs is no less than the number of rows in an input table. 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. Some tables may be loaded to the matrix memory.

An associative processing unit is represented as h ($h \geq 4$) vertical registers each consisting of p bits (Figure 3). A vertical register can be regarded as a one–column array. The STAR–machine runs as follows. The bit columns of the tabular data are stored in the registers which perform the necessary Boolean operations.

To simulate data processing in the matrix memory, we use data types **word, slice**, and **table**. Constants for the types **slice** and **word** are represented as a sequence of symbols of $\{0, 1\}$ in single quotation marks. The types **slice** and **word** are used for the bit column access and the bit row access, respectively, and the type **table** is used for defining the tabular data. Assume that any variable of the type **slice** consists of p components which belong to $\{0, 1\}$. For simplicity let us call *slice* any variable of the type **slice**.

Let us present the main operations for slices.

Let X, Y be variables of the type **slice** and i be a variable of the type **integer**. We use the following operations:

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

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

$Y(i)$ selects the value of the i-th component of Y;

FND(Y) returns the ordinal number i of the first (the uppermost) bit $'1'$ of Y, $i \geq 0$;

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

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

CONVERT(Y) returns a row, whose every i-th bit coincides with $Y(i)$. It is applied when a row of one matrix is used as a slice for another matrix;

FRST(Y) saves the first (the uppermost) component $'1'$ in the slice Y and sets to $'0'$ its other components. For example, if $Y =' 0010110'$, then $Y =' 0010000'$ after performing the operation FRST(Y);

MASK(Y, i, j) sets components $'1'$ from the i-th through the j-th positions and components $'0'$ in other positions of the slice Y $(i < j)$;

PRESS(X, Y) erases from the slice X all components which are matched with bits $'0'$ in the slice Y, and then compresses the contents of X. We observe that the contents of X does not change if the slice Y has no bits $'0'$.

The operations FND(Y), STEP(Y), NUMB(Y), and CONVERT(Y) are used only as the right part of the assignment statement, while the operation $Y(i)$ is used as both the right part and the left part of the assignment statement.

To carry out data parallelism, we introduce in the usual way the bitwise Boolean operations: $X\, and\, Y$, $X\, or\, Y$, $not\, Y$, $X\, xor\, Y$. We also use the predicate SOME(Y) that results in **true** if there is at least a single bit $'1'$ in the slice Y.

We will employ the following functions that transform the contents of the slice Y.

SHIFT($Y, down, k$) moves the contents of the slice Y by k bits down as follows. Bits $'0'$ are set from the first through the k-th positions and the last k components are shifted out of its edge.

SHIFT(Y, up, k) is defined by analogy with SHIFT($Y, down, k$).

MIRROR(Y) performs the reverse of the contents of the slice Y.

All the operations, the predicate and the function MIRROR for slices can also be performed for the type **word**.

In the new version of the language STAR, we employ the following two operations:

TRIM(i, j, w) cuts the substring of the string w from the i-th through the j-th bits, where $1 \leq i < j \leq \mid w \mid$;

REP(i, j, v, w) replaces the substring $w(i)w(i+1)\ldots w(j)$ of the string w with the string v, where $\mid v \mid = j - i + 1$ and $1 \leq i < j < \mid w \mid$.

For two variables v and w of the type **word** having the same length, we also use the following new functions:

ADD(v, w) performs the addition of binary strings v and w;

SUBT(v, w) performs the subtraction of the binary string w from the binary string v for the case when $v > w$.

Moreover, we utilize the following predicates first proposed by Potter in [19]: EQ(v, w), NOTEQ(v, w), LESS(v, w), LESSEQ(v, w), GREAT(v, w), and GREATEQ(v, w).

For a variable T of the type **table**, we use the following 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;

WITH(X, T) augments the matrix T by X from the left.

We also use the function SIZE(T) which yields the number of columns in the matrix T. Obviously, the value of SIZE(T) is incremented by one after performing the operation WITH(X, T).

Remark 1. Observe that the STAR statements [7] are defined in the same manner as for Pascal. We will use them for presenting our procedures.

Following [4], we assume that each elementary operation of the STAR–machine takes one unit of time. Therefore we will measure the *time complexity* of an algorithm by counting all elementary operations performed in the worst case.

3. Basic associative algorithms for non–numerical computing

In this section, we suggest a group of main associative parallel algorithms used for non–numerical processing. These algorithms are divided into two groups depending on whether we apply them to a single array or to two arrays. Correctness of the corresponding procedures is easily verified by induction on the number of bit columns in the corresponding matrix.

3.1. Basic procedures using a single array

We first consider two basic algorithms which use a given pattern. These algorithms are based on the corresponding Falkoff's algorithms [1]. Then, we analyze other algorithms which are applied to a single array of integers.

The procedure MATCH(T, X, v, Z) uses the given slice X to indicate with bits $'1'$ the row positions used here. It defines *positions* of those rows of the given matrix T which coincide with the given pattern v (Figure 4). The procedure returns the slice Z, where $Z(i) =' 1'$ if and only if ROW$(i, T) = v$ and $X(i) =' 1'$.

```
procedure MATCH(T: table; X: slice; v: word; var Z: slice);
var Y: slice; i,k: integer;
Begin Z:=X; k:=SIZE(T);
  for i:=1 to k do
    begin Y:=COL(i,T);
      if v(i)='1' then Z:=Z and Y
      else Z:=Z and (not Y);
    end;
End;
```

Let us briefly explain the run of this procedure. Initially the rows of T, whose positions are selected with bits $'1'$ in the slice X, are candidates for analysis. At any i-th iteration $(1 \leq i \leq \log v)$, we examine the i-th order

v	1	0	1	1

T				X	Z
1	0	1	0	1	0
0	0	1	1	0	0
1	0	1	1	1	1
1	0	1	1	0	0
1	0	1	1	1	1
1	0	1	0	1	0

Figure 4. Testing $v \in T$

bit of v beginning with its first bit and eliminate from further consideration the candidates disagreeing in this position with the i-th bit of v.

The procedure GEL(T, w, X, Y) separates the rows (numbers) of the given matrix T into the following classes: 'greater than', 'equal to' and 'less than' the given pattern w. It returns the slices X and Y satisfying the following properties:

– ROW$(i, T) > w$ if and only if $X(i) =' 1'$ and $Y(i) =' 0'$;
– ROW$(i, T) = w$ if and only if $X(i) =' 0'$ and $Y(i) =' 0'$;
– ROW$(i, T) < w$ if and only if $X(i) =' 0'$ and $Y(i) =' 1'$.

```
procedure GEL(T: table; w: word; var X,Y: slice);
var i,k: integer; Z,B: slice;
Begin k:=SIZE(T); CLR(X); CLR(Y); SET(Z);
  for i:=1 to k do
    begin B:=COL(i,T); if w(i)='1' then
      begin B:=Z and (not B); Y:=Y or B
/* In the slice $Y$, we accumulate positions
  of those $i$-th rows for which ROW$(i,T) < w$. */
      end
      else
      begin B:=Z and B; X:=X or B
/* In the slice $X$, we accumulate positions
  of those $i$-th rows for which ROW$(i,T) > w$. */
      end;
    Z:=Z and (not B);
/* Positions of the selected rows are deleted
  from the slice $Z$. */
    end;
End;
```

Let us explain the main idea of this algorithm. At first, the highest order bit of the given pattern w is examined. If $w(1) =' 1'$, then all numbers

w	1	0	1	1

T				X	Y
1	0	1	0	0	1
0	1	1	1	0	1
1	0	1	1	0	0
1	0	1	1	0	0
1	1	0	1	1	0
1	0	1	0	0	1

Figure 5. The use of the procedure GEL

from T, beginning with $'0'$, are necessary smaller than w. Positions of such numbers (rows) are separated and eliminated from further examination. Numbers with the same first bit as w are indeterminate. At any i-th iteration $(i \geq 1)$, we examine the i-th order bit of w and all indeterminate numbers from T. Among them, we separate and eliminate from further consideration those numbers which are necessary smaller or greater than w.

Remark 2. Obviously, using the procedure GEL, it is easy to determine the rows of T, where $\text{ROW}(i,T) \geq w$ and $\text{ROW}(i,T) \leq w$. Moreover, in [8], by means of the procedure GEL, we have designed basic associative parallel algorithms for finding the greatest lower bound, the least upper bound, between limits and outside limits.

The procedure $\text{MIN}(T,X,Z)$ defines *positions* of those rows of the given matrix T, where minimal elements are located. It returns the slice Z, where $Z(i) =' 1'$ if and only if $\text{ROW}(i,T)$ is the minimal element of the matrix T and $X(i) =' 1'$.

```
procedure MIN(T: table; X: slice; var Z: slice);
var Y: slice; i,k: integer;
Begin Z:=X; k:=SIZE(T);
  for i:=1 to k do
    begin Y:=COL(i,T); Y:=Z and ( not Y);
      if SOME(Y) then Z:=Y
    end;
End;
```

Let us explain the run of the procedure MIN. Assume that there exists $j \geq 1$ such that for the first time candidates disagree in the j-th bit. Then at this iteration, all the candidates with $'1'$ in the j-th bit are eliminated from further consideration. We examine successively the lower–order bit positions in the same way, each time eliminating from further consideration those candidates which disagree and have $'1'$ in the same bit position.

In [19], there are standard functions *minval*(T) and *mindex*(T) that define the minimal value in the matrix T and its position. On the STAR–machine, we realize these functions by means of the resulting slice Z of the procedure MIN(T, X, Z) as follows: `k:=FND(Z); w:=ROW(k,T)`.

The procedure MAX(T, X, Z) is defined by analogy with MIN(T, X, Z).

Note that the standard functions $maxval(T)$ and $maxdex(T)$ from [19] are defined on the STAR–machine in the same manner as shown above.

Now we present the procedure BUBBLE(T, X, P) that sorts the rows (integers) of the matrix T selected with bits $'1'$ in the slice X, in the *increasing order*. This procedure returns a matrix P that has the same number of rows as the given matrix T and the number of its columns is equal to the number of different rows (numbers) in T. The number of columns in the matrix P is determined during the execution of the procedure BUBBLE. Note that every r-th column saves with $'1'$ the *positions* of the r-th value assuming that the smallest number has the *first* value.

```
procedure BUBBLE(T: table; X: slice; var P: table);
var Y,Z: slice; i,k: integer;
Begin i:=0; k:=NUMB(X); Y:=X;
  while SOME(Y) do
    begin i:=i+1; MIN(T,Y,Z);
      COL(i,P):=Z; Y:=Y and (not Z)
    end;
End;
```

The procedure BUBBLE runs as follows. Initially, the rows of T, indicated with bits $'1'$ in the slice X, are candidates for analysis. At any i-th iteration, by means of the procedure MIN, we first determine the positions of the i-th value, then we write them in the i-th column of the matrix P, and after that we delete these positions from further consideration.

In [19], there are standard functions $nthval(i, T)$ and $nthdex(i, T)$ to determine the i-th value in the matrix T and its location. On the STAR–machine, we obtain these functions by means of the resulting matrix P of the procedure BUBBLE as follows: `Y:=COL(i,P); k:=FND(Y); w:=ROW(k,T).`

The following three procedures are very simple. Therefore we present them without explanations.

The procedure WCOPY(v, X, F) writes the binary word v into those rows of the matrix F which are selected with bits $'1'$ in X. Other rows of the matrix F consist of zeros.

```
procedure WCOPY(v: word; X: slice; var F: table);
var Y: slice; i,k: integer;
Begin CLR(Y); k:=SIZE(F);
  for i:=1 to k do
    if v(i)='1' then COL(i,F):=X
    else COL(i,F):=Y
End;
```

The procedure TCOPY1(T, j, h, F) writes h columns from the given matrix T, starting with the $(1+(j-1)h)$-th column, into the resulting matrix F, where $j \geq 1$.

```
procedure TCOPY1(T: table; j,h: integer; var F: table);
var X: slice; i,r: integer;
Begin for i:=1 to h do
  begin r:=i+(j-1)h; X:=COL(r,T);
    COL(i,F):=X
  end;
End;
```

The procedure TCOPY2(R, j, h, T) writes the given matrix R, consisting of h columns, into the resulting matrix T, starting with the $(1+(j-1)h)$-th column, where $j \geq 1$. This procedure is implemented on the STAR–machine by analogy with TCOPY1(T, j, h, F).

3.2. Basic procedures using two arrays

Here we propose a group of basic procedures to perform in parallel both the comparison of the corresponding rows of two arrays and some variants of merging two arrays.

The procedure SETMIN(T, F, X, Z) defines the positions of the matrix T rows that are less than the corresponding rows of the matrix F. It returns the slice Z, where $Z(j) =' 1'$ if and only if ROW$(j, T) <$ ROW(j, F) and $X(j) =' 1'$.

```
procedure SETMIN(T,F: table; var X,Z: slice);
/* The matrices $T$ and $F$ have the same size. */
var B,M,Y: slice; i,k: integer;
Begin k:=SIZE(T); CLR(Z);
  for i:=1 to k do
    begin B:= COL(i,T); Y:=COL(i,F);
      M:=B xor Y; M:=M and X;
/* In the slice $M$, we save positions of the rows
  where ROW$(j,T) \neq$ ROW$(j,F)$ and $X(j) =' 1'$. */
      B:=Y and (not B); B:=B and X;
/* In the slice $B$, we save positions of the rows
  where ROW$(j,T) <$ ROW$(j,F)$ and $X(j) =' 1'$. */
      Z:=Z or B; X:=X and (not M)
    end;
End;
```

The procedure SETMIN runs as follows. Initially, the resulting slice Z consists of zeros. At any i-th iteration, we first save the positions of the corresponding rows of T and F that disagree in the i-th bit. Then we determine the positions j of the corresponding rows of T and F, where the i-th bit of ROW(j, T) is $'0'$ and the i-th bit of ROW(j, F) is $'1'$. We add these row positions to the resulting slice Z. After that, we delete from further consideration the corresponding rows of T and F that disagree in the i-th bit.

The procedure SETMAX(T, F, X, Y) is defined by analogy with the procedure SETMIN.

The procedure HIT(T, F, X, Z) defines the positions of the corresponding identical rows of the given matrices T and F. It returns a slice Z, where $Z(i) =' 1'$ if and only if ROW(i, T)=ROW(i, F) and $X(i) =' 1'$.

```
procedure HIT(T,F: table; X: slice; var Z: slice);
var i,k: integer; B,Y: slice(T);
Begin Z:=X; k:=SIZE(T);
  for i:=1 to k do
    begin B:=COL(i,T); Y:=COL(i,F);
      Y:=B xor Y;
      Z:=Z and ( not Y)
    end;
End;
```

The procedure HIT runs as follows. Initially, the resulting slice Z coincides with the given slice X. At any i-th iteration ($i \geq 1$), we first determine the positions of the corresponding rows of the matrices T and F that disagree in the i-th bit. Then we delete these positions from the slice Z.

The procedure TMERGE(T, X, F) writes the rows of the matrix T, indicated with bits $'1'$ in the slice X, into the matrix F. Other rows of the matrix F do not change.

```
procedure TMERGE(T: table; X: slice; var F: table);
/* The matrices T and F have the same size. */
var Y,Z: slice; i,k: integer;
Begin k:=SIZE(T);
  for i:=1 to k do
    begin Y:=COL(i,T); Y:=Y and X;
      Z:=COL(i,F); Z:=Z and ( not X);
      Z:=Z or Y; COL(i,F):=Z
    end;
End;
```

This procedure runs as follows. At any i-th iteration, by means of two slices, we save both the positions of the matrix T rows, selected with bits $'1'$

in the slice X, and the positions of the matrix F rows, selected with bits $'0'$ in the slice X. The disjunction of these slices is stored in the i-th column of the matrix F.

The procedure WMERGE(w, X, F) writes the binary word w into the rows of the resulting matrix F selected with bits $'1'$ in the slice X. Other rows of the matrix F do not change.

```
procedure WMERGE(w: word; X: slice; var F: table);
var Z: slice; i,k: integer;
Begin k:=SIZE(F);
  for i:=1 to k do
    begin Z:=COL(i,F); Z:=Z and ( not X);
/* In the slice $Z$, we save bit positions in the $i$-th
  column of the matrix $F$ which do not change. */
        if w(i)='0' then COL(i,F):=Z else
          begin Z:=Z or X; COL(i,F):=Z
          end;
    end;
End;
```

This procedure runs as follows. At any i-th iteration, by means of the slice Z, we save the i-th bits of the matrix F rows, selected with $'0'$ in the slice X. If the i-th bit of w is $'0'$, we store the slice Z into the i-th column of the matrix F. Otherwise, we have to write the bit $'1'$ into the positions of the i-th column of F indicated with $'1'$ in the slice X. Therefore we store the disjunction of the slices Z and X into the i-th column of F.

4. Basic associative algorithms for numerical computing

In this section, we propose associative parallel algorithms to perform in parallel the addition and subtraction of the corresponding rows of two matrices, whose positions are selected with bits $'1'$ in the given slice X. These algorithms use tables 5.1 and 5.2 from [4]. Correctness of the corresponding procedures for numerical computing can be verified by induction in terms of the number of bit columns in the corresponding matrix.

The procedure ADDV(T, F, X, R) writes the result of adding the matrices T and F into the matrix R. Let us explain the main idea of this procedure. At any i-th iteration, by means of two slices, we save separately the carry digit both at this iteration and at the previous one. Initially, the carry digit at the previous iteration is zero.

```
procedure ADDV(T,F: table; X: slice; var R: table);
var B,Y,Z,M: slice; var i,k: integer;
/* We use the slice $B$ to save the carry digit
```

```
    at the current iteration and the slice $M$ for
    the carry digit at the previous one.*/
  Begin CLR(M); k:=SIZE(T);
    for i:=k downto 1 do
      begin Y:=COL(i,T); Y:=Y and X;
        Z:=COL(i,F); Z:=Z and X;
        B:=Y and Z; Z:=Y xor Z;
        COL(i,R):=Z xor M;
  /* We obtain the carry digit at the current step when
    $Y(i) = Z(i) =' 1'$ and when $Y(i) \neq Z(i)$ and $M(i) =' 1'$.*/
        Y:=Z and M; B:=B or Y; M:=B
      end;
        if SOME(B) then WITH(B,R)
  End;
```

The procedure ADDC(T, X, v, F) adds the binary word v to those rows of the matrix T which are selected with bits $'1'$ in the given slice X, and writes down the result into the corresponding rows of the matrix F. Other rows of the matrix F consist of zeros.

```
  procedure ADDC(T: table; X: slice; v: word; varF: table);
  var R: table;
  Begin WCOPY(v,X,R);
  /* We write the word $v$ in the rows of the matrix $R$
    selected with bits $'1'$ in the slice $X$.*/
    ADDV(T,R,X,F);
  End;
```

The procedure SUBTV(T, F, Z, R) writes the result of subtracting the matrix F from the matrix T into the matrix R. Let us explain the main idea of this algorithm. At any i-th iteration, by means of two slices, we save both the borrow digit at this iteration and the borrow digit from the previous one. Initially, the borrow digit from the previous iteration is zero.

```
  procedure SUBTV(T,F: table; Z: slice; var R: table);
  var B,M,P,X,Y: slice i,k: integer;
  Begin k:=SIZE(T); CLR(M);
  /* We use the slice $M$ for saving the borrow digit from
    the previous iteration and the slice $B$ for
    the borrow digit at the current one.*/
    for i:=k downto 1 do
      begin X:=COL(i,T); X:=X and Z;
        Y:=COL(i,F); Y:=Y and Z;
        P:=Y and (not X); B:=P and (not M);
  /* At the current iteration, we save the borrow digit
    in the slice $B$ if $Z(i)='0'$ and $Y(i)='1'$ and $M(i)='0'$. */
        P:=X and (not Y); P:=P and M; Y:=Y xor X;
        Y:=Y xor M; COL(i,R):=Y; M:=M and (not P);
  /* If at the current iteration $X(i)='1'$ and $Y(i)='0'$
    and $M(i)='1'$, we write $M(i)='0'$ in the corresponding
    positions of the slice $M$. */
        M:=M or B
      end;
  End;
```

The procedure SUBTC(T, X, v, F) subtracts the binary word v from the rows of the matrix T selected with bits $'1'$ in the slice X. It is implemented on the STAR–machine by analogy with the procedure ADDC(T, X, v, F).

Let us evaluate the time complexity of the basic procedures. Obviously, the basic procedures, which use only elementary operations of the STAR–machine, take $O(k)$ time each, where k is the number of columns in the corresponding matrix. The procedure BUBBLE takes $O(kn)$ time since the procedure MIN takes $O(k)$ time and it repeats n times, where n is the number of different rows in the given matrix T.

5. Conclusions

In this paper, we have proposed a new version of the language STAR. For slices, it includes the new operations CONVERT and FRST. The specific functions for slices, such as ROTATE and SHUFFLE from the first version, have not been included into the new one. For variables of the type **word**, we have proposed new operations TRIM and REP, new functions ADD and SUBT, and a group of predicates for comparing two binary strings of the same length. We have also presented associative parallel algorithms for the basic procedures included into the library of the language STAR. We have obtained that the basic procedure BUBBLE takes $O(kn)$ time. Other basic procedures take $O(k)$ time each, where k is the number of bit columns in the corresponding matrix. Moreover, we have shown how to implement on the STAR–machine the basic functions of the language ASC [19].

We are planning to apply the library of the language STAR for seismic data processing.

References

[1] Falkoff A. D. Algorithms for Parallel–Search Memories // J. of the ACM. – 1962. – Vol. 9, N 10. – P. 488–510.

[2] Fernstrom C., Kruzela J., Svensson B. LUCAS Associative Array Processor // Design, Programming and Application Studies. – Lect. Notes Comput. Sci. – Berlin: Springer–Verlag,1986. – Vol. 216.

[3] Fet Y. I. Parallel Processing in Cellular Arrays. – Tauton: Research Studies Press, UK, 1995.

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

[5] Grosspietsch K. E., Reetz R. The associative processing system CAPRA: architecture and applications // Associative processing and processors / A. Krikelis, C.C. Weems (eds.). – IEEE Computer Society, 1997. – P. 72–81.

[6] Mirenkov N. The Siberian Approach for an Open-system High-performance Computing Architecture // Computing and Control Engineering J. – 1992. – Vol. 3, N 3. – P. 137–142.

[7] 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.

[8] Nepomniaschaya A. S. Investigation of Associative Search Algorithms in Vertical Processing Systems // Proc. of the Intern. Conf. "Parallel Computing Technologies". Obninsk, Russia, 1993. – P. 631–641.

[9] Nepomniaschaya A. S., Vladyko M. A. Comparison of Models for Associative Parallel Computations // Programming. – Moscow: Nauka, 1997. – N 6. – P. 41–50 (in Russian).

[10] Nepomniaschaya A. S. Solution of path problems using associative parallel processors // Proc. Int. Conf. on Parallel and Distributed Systems ICPADS'97. – IEEE Computer Society, 1997. – P. 610–617.

[11] Nepomniaschaya A. S., Dvoskina M. A. A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors // Fundamenta Informaticae. - Amsterdam: IOS Press, 2000. – Vol. 43. – P. 227–243.

[12] Nepomniaschaya A. S. Associative Parallel Algorithms for Computing Functions Defined on Paths in Trees // Proc. of the Intern. Conf. on Parallel Computing in Electrical Engineering. – Los Alamitos: IEEE Computer Society, 2002. – P. 399–404.

[13] Nepomniaschaya A. S. Associative Parallel Algorithms for Dynamic Edge Update of Minimum Spanning Trees // Proc. of the 7th Int. Conf. PaCT 2003. – Lect. Notes Comp. Sci. – Berlin: Springer–Verlag, 2003. – Vol. 2763. – P. 141–150.

[14] Nepomniaschaya A. S. Efficient Implementation of the Italiano Algorithms for Updating the Transitive Closure on Associative Parallel Processors // Fundamenta Informaticae. – Amsterdam: IOS Press, 2008. – Vol. 89. – P. 313–329.

[15] Nepomniaschaya A. S. Associative version of the Ramalingam decremental algorithm for dynamic updating the single-sink shortest-paths subgraph //

Proc. of the 10-th Intern. Conf. “Parallel Computing Technologies” (PaCT-09), Novosibirsk, Russia, 2009. – Lect. Notes Comput. Sci. – Berlin: Springer–Verlag, 2009. – Vol. 5698. – P. 257–268.

[16] 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: IEEE Computer Society, 1995. – P. 308–314.

[17] Otrubova B., Sykora O. Orthogonal Computer and its Application to Some Graph Problems // Parcella'86. – Berlin: Academie Verlag, 1986. – P. 259–266.

[18] Ozkarahan E. Database Machines and Database Management. – Prentice–Hall, 1986.

[19] Potter J. L. Associative Computing: A Programming Paradigm for Massively Parallel Computers. – Kent State University, New York and London: Plenum Press, 1992.

[20] Potter J., Baker J., Bansal A. et al. ASC – An Associative Computing Paradigm // Computer: Special Issue on Associative Processing. – 1994. – Vol. 27, N 11. – P. 19–24.

[21] Sima D., Fountain T., Kacsuk P. Advanced Computer Architectures, A Design Space Approach. – Addison-Wesley, 1997.