Bull. Nov. Comp. Center, Comp. Science, 23 (2005), 71–83
© 2005 NCC Publisher

Efficient update of tree paths on associative systems with bit-parallel processing[*]

A. Sh. Nepomniaschaya

Abstract. In this paper, we propose an efficient associative parallel algorithm for updating tree paths after every change in the underlying graph. Such a problem arises when we perform dynamic graph algorithms. To speed up time complexity of the algorithm, we use a new associative model called associative graph machine (AG–machine). It carries out bit–serial and fully parallel associative processing of matrices representing graphs as well as some basic operations on matrices. On the AG–machine, the algorithm is implemented as procedure TreePaths. We prove its correctness and evaluate time complexity.

1. Introduction

Updating a minimum spanning tree (MST) after changes in network topology is a fundamental problem. Let $G = (V, E_G)$ be an undirected graph with n vertices and m edges and $T = (V, E)$ be its MST. Let one of the following changes be performed in G: deletion or insertion of an edge, or deletion or insertion of a vertex along with its incident edges. We want to compute a new MST for the altered graph by performing changes in the given T. Dynamic graph algorithms are designed to handle graph changes. They maintain some property of a changing graph more efficiently than recomputation of the entire graph with a static algorithm after every change.

In this paper, we study updating tree paths after every change in the underlying graph. In particular, such a problem arises when we perform dynamic edge update of an MST.

To solve update problems, different techniques are used. In [9], Tarjan proposes a special technique, path compression on balanced trees, to compute functions defined on paths in trees under various assumptions. In [3], Frederickson suggests a graph decomposition and data structures techniques to deal with the edge update problem. In [1], a general technique, called sparsification, for designing dynamic graph algorithms is provided. In [8], Pawagi and Ramakrishnan propose a technique for updating tree paths on parallel random access machines. Their technique is based on representing an MST in the form of an inverted tree. In [5], we suggest a new technique for updating tree paths on a model of associative (content addressable) sys-

[*] This work was supported in part by the Russian Foundation for Basic Research under Grant 03-01-00399.

tems with bit–serial (vertical) processing (the STAR–machine). It simulates the run of fine–grained massively parallel SIMD architectures.

Here, we propose a modified version of the associative parallel algorithm for updating tree paths [6] that can be also used for the vertex update problem. To speed up time complexity of this algorithm, we use a new associative model called associative graph machine (AG–machine) that performs bit–serial and fully–parallel associative processing of matrices representing graphs as well as some basic operations on matrices [7]. On one hand, this model generalizes the STAR–machine. On the other hand, an explanation of its possible hardware implementation is proposed in [7]. On the AG–machine, the algorithm is implemented as the procedure TreePaths. We justify correctness of this procedure and evaluate its time complexity. We obtain that it takes $O(h)$ time, where h is the number of vertices in the tree path joining end–points of the deleted and the inserted edges in the current MST. It is assumed that each elementary operation of the AG–machine (its microstep) takes one unit of time.

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 [2] and some *new operations* to perform bit–parallel processing.

The model consists of the following components:

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

– an associative processing unit forming a two–dimensional array of single–bit PEs;

– a matrix memory for the associative processing unit.

The CU broadcasts each instruction to all PEs in unit 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 the new operation FRST(Y).

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

For completeness, we recall some elementary operations for slices from [4] being used in the paper.

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

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

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

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

The above–mentioned operations 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.

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 $not\,(A, v)$ *simultaneously* replaces the columns of the given matrix A, marked by ones in the comparand v, with their negation. It will be used as the right part of the assignment statement.

Remark 1. It should be noted that the above presented two operations use the column parallelism, while the next two operations of this group will use the row parallelism.

The operation FRST(A) *simultaneously* replaces each i-th row of the given matrix A with FRST(ROW(i, A)).

The operation $or\,(A, Y)$ *simultaneously* performs disjunction in every row of the given matrix A and saves the result in the slice Y, that is, $\forall i$ $Y(i) =' 0'$ if and only if ROW(i, A) consists of zeros.

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

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

The operation $op\,(A,B,v)$, where $op \in \{or, and, xor\}$, is *simultaneously* performed between those columns of the given matrices A and B that are marked by ones in the given comparand v. This operation is used as the right part of the assignment statement, that is, $C := op\,(A,B,v)$.

Remark 2. We will assume that each elementary operation of the AG–machine (its microstep) takes one unit of time.

Now, we provide an implementation on the associative graph machine of a few basic procedures which will be used in this paper.

The procedure WCOPY(w,X,F) writes the binary word w in the rows of the result matrix F that correspond to ones in the given slice X. Other rows of F will consist of zeros. On the AG–machine, it is implemented as follows.

```
procedure WCOPY(w: word; X: slice; var F: table);
var Y: slice; v: word;
Begin CLR(Y); v:= not w;
  SCOPY(F,X,w);
  SCOPY(F,Y,v);
End;
```

This procedure runs as follows. The slice X is simultaneously written in the matrix F columns that correspond to $'1'$ in the given w, while the slice Y is simultaneously written in other columns of F.

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

```
procedure HIT(T,F: table; X: slice; var Y: slice);
var A: table; v: word;
Begin SET(v);
  A:= xor (T,F,v);
  or (A,Y);
/* $Y(j) =' 0'$ iff ROW$(j,A)$ consists of zeros. */
  Y:=X and ( not Y);
End;
```

This procedure runs as follows. First, the corresponding columns in the given matrices T and F are simultaneously compared and the result is saved in the matrix A. Then the disjunction is simultaneously performed in all rows of the matrix A. Finally, we take into account the corresponding rows of matrices T and F selected by $'1'$ in X.

The procedure MATCH(T, X, w, Y) defines positions of rows in the given matrix T that coincide with the given pattern w. It returns the slice Y, where $Y(i) =' 1'$ if and only if ROW$(i, T) = w$ and $X(i) =' 1'$.

```
procedure MATCH(T: table; X: slice; w: word; var Y: slice);
var A: table;
Begin WCOPY(w,X,A);
  HIT(T,A,X,Y);
End;
```

This procedure runs as follows. First, the given pattern w is written in the matrix A rows marked by $'1'$ in the given slice X. Then the procedure HIT is applied to matrices T and A.

On the AG–machine, these procedures take $O(1)$ time each, while on the STAR–machine, they require $O(k)$ time, where k is the number of columns in the corresponding matrix [4].

3. Updating tree paths

Let $G = (V, E)$ denote an *undirected graph*, where V is the set of vertices and E is the set of edges. Let $wt(e)$ denote the weight of the edge e. We assume that $V = \{1, 2, \ldots, n\}$, $|V| = n$, and $|E| = m$.

A *connected component* is a maximal connected subgraph of G.

A *minimum spanning tree* $T = (V, E')$ is a connected acyclic subgraph of G, where $E' \subseteq E$ and the sum of weights of the corresponding edges is minimum.

A *path* from v_1 to v_k in the graph G is a sequence $v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k$ of alternating vertices and edges such that $e_i \in E$ and $e_i = (v_i, v_{i+1})$ for $1 \leq i < k$.

We will deal with paths in the MST T. It means that edges from any *tree path* will belong to the set E'.

Let every edge (u, v) be matched with the triple $(u, v, wt(u, v))$. Note that vertices and weights are written in binary code. On the AG–machine, a graph is represented as association of matrices *left*, *right*, and *weight*, where every triple $(u, v, wt(u, v))$ occupies an individual row, and $u \in left$, $v \in right$, and $wt(u, v) \in weight$. A minimum spanning tree is represented as a slice, where *positions* of edges belonging to it are marked by $'1'$.

We also use a matrix of tree paths M consisting of m rows and n columns. Its every i-th column saves the tree path from the root v_1 to vertex v_i. Initially, it is obtained along with the MST [5].

Now, we illustrate the representation of the given graph G, its MST T, and the corresponding matrix of tree paths M on the AG–machine.

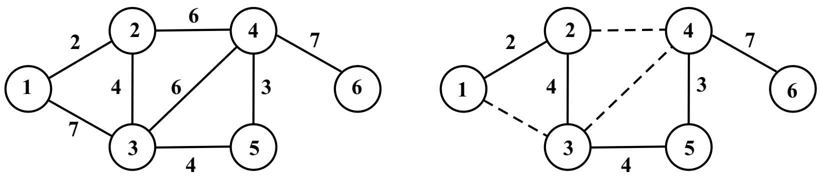

Figure 1. Graph G and its MST

	Tables			Slices	
	left	right	weight	S	T
1	001	010	010	1	1
2	001	011	111	1	0
3	010	011	100	1	1
4	010	100	110	1	0
5	011	100	101	1	0
6	100	101	011	1	1
7	011	101	100	1	1
8	100	110	111	1	1

	Table M					
	1	2	3	4	5	6
1	0	1	1	1	1	1
2	0	0	0	0	0	0
3	0	0	1	1	1	1
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	1	0	1
7	0	0	0	1	1	1
8	0	0	0	0	0	1

In [6], we have proposed two associative parallel algorithms for dynamic edge update of an MST. These algorithms employ the matrix of tree paths M. After finding a new MST, the matrix of tree paths changes by means of the auxiliary procedure TreePaths.

Let us agree that $p_{i,j}$ $(1 < i, j \leq n)$ denotes a tree path from the MST joining vertices v_i and v_j and p_i denotes a tree path from vertex v_1 to vertex v_i. Obviously, every i-th column of the matrix M saves the tree path p_i, that is, *positions* of edges from the MST joining vertices v_1 and v_i are marked by $'1'$ as shown in our example.

The matrix M has the following two properties:

(1) $\forall i \neq j$, $p_{i,j} = p_i \, xor \, p_j$, where $1 < i, j \leq n$.

(2) Let an edge γ from the i-th row of the graph representation be deleted from the MST. Then vertices marked by $'1'$ in the i-th row of M form a separate connected component.

Really, let in our example the edge $(3, 5)$ be deleted from the MST T. Then vertices $\{4, 5, 6\}$ marked by $'1'$ in the 7-th row of the matrix M form a separate connected component.

Remark 3. If a tree path p_i is the initial part of the tree path p_j, then obviously $p_{i,j} = p_j \, and \, (\, not \, p_i)$.

Let a new MST be obtained from the underlying one by deleting an edge (say, γ) located in the l-th position and inserting an edge (say, δ) located in the k-th position. Let C be a connected component of G obtained after deleting γ. The algorithm for updating tree paths will determine *new* tree paths for all vertices from C.

Let v_{del} and v_{ins} be end–points of the corresponding edges γ and δ that belong to C. Let $M1$ be a copy of the matrix of tree paths M. The matrix $M1$ will save tree paths $before$ updating the current MST. Let a slice P save *positions* of tree edges joining v_{ins} and v_{del}. Obviously, directions of edges on the path $[v_{del} \rightarrow v_{ins}]$ will be reversed in the new MST.

Let p_i $(1 < i \leq n)$ denote a tree path $before$ updating the MST and p'_i denote a tree path $after$ updating the MST.

The associative parallel algorithm starts at vertex v_{ins}. Note that p'_{ins} is known. The algorithm carries out the following stages.

At the $first$ stage, copy the matrix M into the matrix $M1$. Then save the l-th row of the matrix $M1$ by means of a variable (say, *comp*). Further, write the given p'_{ins} in the corresponding column of M. Finally, fulfil the statement $r := ins$.

While P is a non-empty slice, repeat stages 2 and 3.

At the *second* stage, determine vertices not belonging to P that form a subtree of the MST with root v_r if any. For every $v_j \neq v_r$ from this subtree, compute p'_j as follows:

$$p'_j := \ (p_j \, and \, (\, not \, p_r)) \, or \, p'_r. \tag{1}$$

Write p'_j in the corresponding column of M. Then mark vertices from this subtree by $'0'$ in *comp*.

At the *third* stage, select position i of an edge from P incident on vertex v_r. Then define its end–point (say, v_q) being adjacent with v_r. Further, determine the new tree path p'_q and write it in the corresponding column of M. Now, mark the edge position i by $'0'$ in the slice P. Finally, perform the statement $r := q$.

At the $fourth$ stage, since P is an empty slice, the vertices marked by $'1'$ in *comp* form a subtree of the MST with root v_r determined just now. For every $v_j \neq v_r$ from this subtree, define p'_j using formula (1). Write p'_j in the corresponding column of M. Then mark vertices from this subtree by $'0'$ in *comp*.

The algorithm terminates when the slice P and the variable *comp* consist of zeros.

Let us illustrate the run of this algorithm.

In Figure 2, the connected component C consists of vertices $v_8, v_9, \ldots, v_{18}$; *del* $= 8$ and *ins* $= 14$. The algorithm starts at vertex v_{14}. Then the new tree paths are recomputed for vertices v_{15}, v_{16}, v_{17}, and v_{18} from the subtree rooted at v_{14}. Further, a new tree path is first defined for v_{13} and then for v_8. Finally, new tree paths are recomputed for vertices v_9, v_{10}, v_{11}, and v_{12} from the subtree rooted at v_8.

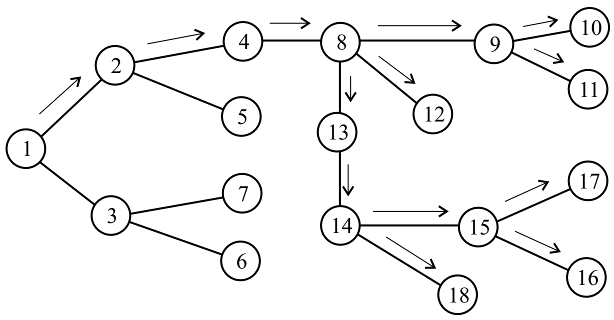

Figure 2. MST before deleting the edge (4,8)

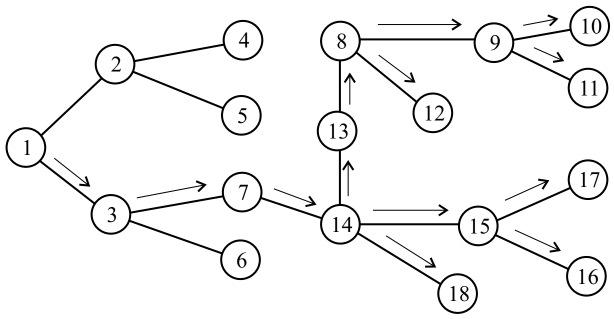

Figure 3. MST after inserting the edge (7,14)

Remark 4. It should be noted that according to formula (1), we first determine the *old* tree path between vertices v_r and v_j. Then we connect this path and the *new* tree path to vertex v_r.

On the AG–machine, this algorithm is implemented as procedure TreePaths which uses the following input parameters: matrices $left$, $right$, and *code*, vertices v_{ins} and v_{del}, and the position l of the deleted edge. It returns the matrix M for the new MST and slices W and P.

Initially, the slice W saves the *new* tree path from v_1 to v_{ins}, the slice P saves *positions* of edges from the tree path joining v_{ins} and v_{del}, and the slice C saves *vertices* whose tree paths will be recomputed.

Remark 5. In [6], the connected component C is given as a slice that saves vertices marked by $'1'$ in the l-th row of the matrix M. However, in the case of the AG–machine, it is convenient to immediately use the l-th row of M, where the deleted edge is located.

We first propose the auxiliary procedure Update. Using formula (1), it computes *in parallel* new tree paths for any subtree with root v_r whose vertices do not belong to the path from P. In this procedure, vertices of the subtree are marked by $'1'$ in *node*1, the slice W saves p'_r and the slice Z saves p_r.

```
procedure Update(M1: table; W,Z: slice; var node1: word;
  var M: table);
var A,B: table;
/* Recall that $p'_j := (p_j \, and \, (not \, p_r)) \, or \, p'_r$ */
Begin SCOPY(A,Z,node1);
/* The old tree path from $v_1$ to $v_r$ is written in $A$. */
  SCOPY(B,W,node1);
/* The new tree path from $v_1$ to $v_r$ is written in $B$. */
  A:= not (A,node1);
  A:= and (M1,A,node1);
  B:= or (B,A,node1);
/* The matrix $B$ saves new tree paths for all $v_j$. */
  SMERGE(M,B,node1);
End;
```

Before presenting the procedure TreePaths, we explain how to determine a subtree with root v_r. To this end, we first determine the *position* i of an edge from P incident on v_r. Then all vertices reachable from v_r will be marked by $'1'$ in the i-th row of the matrix $M1$. Among them, we exclude the vertices updated before.

Now, we propose the procedure TreePaths.

```
procedure TreePaths(left,right: table; code: table;
  l,ins,del: integer; var M: table; var P,W: slice(left));
/* New tree paths for vertices from the connected component $C$
  will be written in the matrix $M$. */
var M1: table;
  N1,N2,X,Z: slice(left);
  S,Y: slice(code);
  comp,current,node1,prev: word(M);
  node: word(code);
  i,q,r: integer;
/* Initialization. */
1. Begin CLR(prev); SET(Y); SET(node1);
/* The first stage. */
2.  SMERGE(M1,M,node1);
/* The matrix $M1$ is a copy of the matrix $M$. */
3.  comp:=ROW(l,M1);
4.  Z:= COL(ins,M1); COL(ins,M):= W;
/* A new path from $v_1$ to $v_{ins}$ is written
  in the corresponding column of $M$. */
5.  r:= ins; node:= ROW(r,code);
/* The second stage. */
6.  while SOME(P) do
```

```
 7.    begin MATCH(left,P,node,N1);
 8.      MATCH(right,P,node,N2);
 9.      X:=N1 or N2; i:=FND(X);
 /* We define the position $i$ of an edge from $P$ incident on $v_r$.  */
10.      node1:=ROW(i,M1);
 /* Vertices whose tree paths include the edge from the $i$-th
    position are marked by '1' in $node1$.  */
11.      comp:=comp and (not node1);
12.      current:=node1;
13.      node1:=node1 and (not prev);
14.      prev:=current;
 /* By means of $prev$, we save the updated vertices.  */
15.      node1(r):='0';
 /* Here, $v_r$ is a subtree root.  */
16.      if SOME(node1) then Update(M1,W,Z,node1,M);
 /* The third stage.  */
17.      if N1(i)='1' then node:=ROW(i,right)
18.      else node:=ROW(i,left);
 /* The binary code of a new subtree root is saved in $node$.  */
19.      MATCH(code,Y,node,S);
20.      q:=FND(S);
 /* Here, $v_q$ is a new subtree root.  */
21.      W(i):='1'; COL(q,M):=W;
 /* A new tree path from $v_1$ to $v_q$ is written
    in the corresponding column of $M$.  */
22.      Z:=COL(q,M1); P(i):='0';
23.      r:=q;
24.    end;
 /* The fourth stage.  */
25.    comp(r):='0';
26.    if SOME(comp) then Update(M1,W,Z,comp,M);
27. End;
```

Correctness of this procedure is established by means of the following theorem.

Theorem. *Let a graph G be given as association of matrices left and right and the matrix code save binary representations of vertices. Let an edge from the l-th position be deleted from the MST and comp be the corresponding connected component. Let del be end–point of the deleted edge and ins be end–point of the inserted edge that belong to comp. Then the procedure TreePaths returns the updated matrix M and slices P and W.*

Proof. (Sketch) We prove this by the induction on the number of edges k belonging to the slice P.

Basis is checked for $k = 1$. On performing lines 1–5, the matrix $M1$ is a copy of M, the variable *comp* saves the connected component obtained after deleting the edge from the l-th row in the graph representation, the variable *prev* consists of zeros, the slice Z saves p_{ins} and p'_{ins} is written in the corresponding column of M, the current vertex v_r coincides with v_{ins}, and the variable *node* saves its binary code.

Further, on fulfilling lines 7–10, we first determine the position i of an edge from P incident on v_{ins}. Then, we determine all vertices whose tree paths include this edge and save them by $'1'$ in the variable $node1$. On performing lines 11–15, we first mark by $'0'$ those vertices of the connected component *comp* that belong to $node1$. Then the variable $node1$ will save a subtree with root v_{ins} whose vertices do not belong to the tree path from P. If $node1$ is non–empty, we determine *in parallel* the new tree paths for all vertices from this subtree (line 16). Further, we execute the next stage.

At the third stage, on performing lines 17–20, the variable *node* saves the binary code of v_{del}. Then on fulfilling lines 21–23, we first write a new tree path to vertex v_{del} in the corresponding column of M. After that, we save p_{del} in the slice Z, mark by $'0'$ the edge position i in the slice P, and perform the statement $r := del$.

Since P is an empty slice, we perform the fourth stage. Here, the variable *comp* saves the subtree rooted at vertex v_{del}. If *comp* becomes empty after deleting root v_{del}, we jump to the procedure end. Otherwise, we determine *in parallel* new tree paths for all vertices of the subtree with root v_{del}. Since *comp* becomes empty after performing Update, we go to the end.

Step of induction. Let the assertion be true for $k \geq 1$. We will prove this for $k + 1$.

Let the slice P save positions of $k + 1$ edges from the tree path $[v_{del} \to v_{ins}]$. Let the edge (v_{del}, v_t) belong to this path. Then we represent the tree path $[v_{del} \to v_{ins}]$ as $(v_{del}, v_t)[v_t \to v_{ins}]$, where the path $[v_t \to v_{ins}]$ consists of k edges. By the induction hypothesis, after updating the tree path $[v_t \to v_{ins}]$, the new tree paths for vertices from subtrees rooted at vertices $v_{ins}, \dots, v_t$ from P are written in the corresponding columns of M, the variable *prev* saves the subtree rooted at v_t, the variable q saves vertex v_t, the slice Z saves p_t while W saves p'_t, and the slice P saves position of the edge (v_t, v_{del}). Since P is non–empty, we perform the $(k + 1)$-th iteration.

Further, we reason in the same manner as in the basis.

Hence, after executing the procedure TreePaths, the matrix M saves new tree paths for all vertices of the updated MST. □

Let us evaluate time complexity of the procedure TreePaths. Since the basic procedure MATCH and the auxiliary procedure Update take $O(1)$ time each, we obtain that the procedure TreePaths takes $O(h)$ time, where h is the number of vertices in the tree path joining end–points of the deleted and the inserted edges in the current MST. On the STAR–machine, such a procedure takes $O(k \log n)$ time [5], where k is the number of *all* vertices in the connected component obtained after deleting an edge from the MST.

4. Conclusions

In this paper, we have described an efficient implementation of the associative parallel algorithm for updating tree paths on the AG–machine. This model carries out both the bit–serial and the bit–parallel processing. The algorithm is given as procedure TreePaths whose correctness is proved. We have obtained that it takes $O(h)$ time, where h is the number of vertices in the tree path joining end–points of the deleted and the inserted edges in the current MST. The proposed implementation of this procedure can be used to design an associative parallel algorithm for dynamic vertex update of an MST on the AG–machine.

References

[1] Eppstein D., Galil Z., Italiano G.F., Nissenzweig A. Sparsification – A technique for speeding up dynamic graph algorithms // J. of the ACM. – 1997. – Vol. 44, N 5. – P. 669–696.

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

[3] Frederickson G.N. Data structures for on-line updating of minimum spanning trees, with applications // SIAM J. Comput. – 1985. – Vol. 14. – P. 781–798.

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

[5] Nepomniaschaya A. S. A new technique for updating tree paths on associative parallel processors // Bull. of the Novosibirsk Computing Center. Ser.: Computer Science. – 2004. – Is. 21. – P. 85–97.

[6] Nepomniaschaya A.S. Associative parallel algorithms for dynamic edge update of minimum spanning trees // Proc. 7th Int. Conf. PaCT 2003. – Lect. Notes in Comp. Sci. – 2003. – Vol. 2763. – P. 141–150.

[7] Nepomniaschaya A., Kokosinski Z. Associative Graph Processor and its Properties // Proc. of the International Conference on Parallel Computing in Electrical Engineering (PARELEC 2004), Dresden, Germany. – 2004. – P. 297–302.

[8] Pawagi S., Ramakrishnan I.V. An $O(\log n)$ algorithm for parallel update of minimum spanning trees // Inform. Process. Letters. – 1986. – Vol. 22. – P. 223–229.

[9] Tarjan R.E. Applications of path compression on balanced trees // J. of the ACM. – 1979. – Vol. 26, N 4. – P. 690–715.